박 사 학 위 논 문
Ph.D. Dissertation

실제적인 유닛 컨텍스트 합성으로 거짓 경보를 줄인 자동화된 유닛 테스트 생성

Automated Unit Test Generation with Realistic Unit Context Synthesis for Low False Alarms

2017

김 윤 호 (金 潤 浩 Kim, Yunho)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

실제적인 유닛 컨텍스트 합성으로 거짓 경보를 줄인 자동화된 유닛 테스트 생성

2017

김 윤 호

한 국 과 학 기 술 원

전산학부

실제적인 유닛 컨텍스트 합성으로 거짓 경보를 줄인 자동화된 유닛 테스트 생성

김 윤 호

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2016년 11월 10일

Automated Unit Test Generation with Realistic Unit Context Synthesis for Low False Alarms

Yunho Kim

Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
November 10, 2016

Approved by

Moonzoo Kim
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 20097016

김윤호. 실제적인 유닛 컨텍스트 합성으로 거짓 경보를 줄인 자동화된 유닛 테스트 생성. 전산학부 . 2017년. 53+iv 쪽. 지도교수: 김문주. (영문 논문)

Yunho Kim. Automated Unit Test Generation with Realistic Unit Context Synthesis for Low False Alarms. School of Computing . 2017. 53+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

소프트웨어 개발 과정에서 대부분 테스트 케이스를 수작업으로 만들고 있기 때문에 소프트웨어 테스팅이 효율적, 효과적이지 못하다. 또한, 소프트웨어가 크고 복잡해짐에 따라 제한된 개발 시간동안 소프트웨어의 다양한 행동을 충분히 살펴보는 것이 어렵다. 이와 같은 문제를 해결하기 위해 소프트웨어 각 소프트웨어 유닛의 테스트 드라이버/스텁 함수와 유닛 테스트 케이스를 자동으로 생성하는 자동화된 유닛 테스트 생성 기법이 개발되었다. 하지만 기존의 자동화된 유닛 테스트 생성 기법은 타겟 유닛의 부정확한 컨텍스트로 인해 실제 프로그램 수행 과정에서는 탐색할 수 없는 실행 경로를 탐색하여 많은 수의 거짓 경보를 생성하는 문제가 있다.

본 논문은 실제적인 유닛 컨텍스트를 합성하여 거짓 경보를 줄이고 버그를 자동으로 탐지하는 자동화된 유닛 테스트 생성 기법을 제안한다. 본 논문은 우선 세계 최초로 산업체의 대규모 C 프로그램에 적용 가능한 자동화된 유닛 테스트 생성 기법 CONBOL을 제안한다. CONBOL은 자동으로 심볼릭 유닛 테스트 드라이버/스텁 함수를 생성하고 부정확한 드라이버/스텁 함수로 생성되는 거짓 경보를 줄이기 위한 거짓 경보 제거 휴리스틱을 적용한다. 400만 줄 규모의 대규모 산업체 C 프로그램을 대상으로 CONBOL을 적용한 결과 24개의 새로운 버그를 발견하여 버그 탐지 능력이 우수함을 입증하였다. 본 논문은 두 번째로 실제적인 유닛 컨텍스트를 합성하여 거짓 경보를 크게 줄인 자동화된 유닛 테스트 생성 기법 CONCERT를 제안한다. CONCERT는 테스트 대상 함수 f의 테스트 드라이버/스텁 함수를 생성할 때 f와 "밀접하게 연관된" 다른 함수의 코드를 활용함으로써 실제적인 유닛 컨텍스트를 생성한다. 테스트 대상 함수 f와 다른 함수 g의 "연관도"는 테스트 대상 소프트웨어가 실행되는 동안 f와 g가 얼마나 많이 같이 실행되었는지를 기준으로 정의된다. 15개 실제 C 프로그램(평균 55KLOC)의 67개 크래시 버그를 대상으로 적용한 결과 CONCERT 는 83.6%의 버그를 탐지하여 높은 버그 탐지 능력을 보이고, 1개의 진짜 경보당 2.4개의 거짓 경보를 보고하여 낮은 거짓/진짜 경보 비율을 보였다.

향후 연구로는 함수 연관도 측정 방법을 더 정확하게 개선하여 거짓 경보를 줄이고 유닛 테스트과정에서 얻은 정보를 활용하여 시스템 테스트 케이스를 생성하고자 한다. 또한, 함수 연관도를 리팩토링, 변화 분석 등 다양한 분석 기법에 적용하여 기존 분석 기법을 개선하고자 한다.

핵 심 낱 말 자동화된 유닛 테스팅, 실제적인 유닛 컨텍스트, 거짓 경보 제거, 함수 연관도, 동적 분석, 콘콜릭 테스팅

Abstract

Current testing practice in industry is often ineffective and inefficient to detect bugs since most test cases are created manually. In addition, the execution space of a complex target program is too large to explore in limited testing time. As a solution for these problems, automated unit test techniques automatically generate drivers/stubs for each unit of a target program and test cases to explore the execution space of each target unit separately. However, these techniques suffer a large number of false alarms due to approximated inaccurate unit contexts which allow infeasible executions of a target unit.

In this dissertation, I present an automated unit test generation framework that synthesizes realistic unit context to automatically detect bugs with low false alarms. The first part of this dissertation presents CONBOL which is the world's first automated unit testing framework for large industrial C programs. CONBOL generates symbolic unit testing drivers/stubs automatically and applies heuristics to reduce false alarms caused by the imprecise drivers/stubs. CONBOL demonstrated its bug detection effectiveness by detecting 24 new crash bugs in a four million lines long industrial embedded program. The second part of this dissertation presents the world's most accurate automated unit testing framework CONCERT which reduces a large number of false alarms by automatically synthesizing realistic unit contexts. CONCERT synthesizes test drivers/stubs to represent realistic contexts of a target unit f by utilizing the code of the other units which are "closely relevant" to f. The "relevance" of other unit g to f is measured based on how many times g and f are executed together in system executions. In the experiments on the 67 crash bugs of the 15 real-world C programs (55KLOC on average), CONCERT demonstrates both high bug detection ability (i.e., 83.6% of the target bugs detected) and low false/true alarm ratio (i.e., 2.4 false alarms per one true alarm).

As future work, I will improve the function correlation metric for reducing false alarms further. Also, I plan to develop a framework to build system test cases based on automatically generated unit test cases. Furthermore, I will utilize unit relevance information for other purposes such as impact analysis.

Keywords Automated unit testing, Realistic unit context, False alarm reduction, Function correlation, Dynamic analysis, Concolic testing

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Challenges in Software Testing

As software is pervasively used in our daily life, the reliability of software becomes important. Software is used not only for non-critical purposes such as entertainment or word-processing, but also for the safety-critical systems such as automotive controllers and avionics. As software plays an important role in such systems, the reliability of software is closely related with our modern society. For example, NIST [66] reported in 2002 that software bugs cost about 59.5 billion dollars to the US economy every year. For another example, Toyota's unintended acceleration caused by software bug killed four persons in a family [70] and Toyota was fined 1.2 billion dollars in US.

Software testing is a de-facto standard method to detect software bugs and increase the reliability of software. Software testing runs software-under-test with a set of inputs (i.e., test cases) and checks if outputs of the software satisfy the expected conditions (i.e., test oracles). As different inputs explore different behaviors of a target program, a sufficient number of quality test cases is a key to the success of software testing. Companies spends significant effort on software testing to improve the reliability of their commercial software. According to the 2016 world quality report [5] by the Capgemini group, industry spends 31% of IT budget for QA and testing in 2016.

Current testing practice in industry, however, often fail to detect bugs in programs because test cases are usually created manually in industry. The world quality report [5] also says that the 41% of responders in industry point out reliance on manual testing is a challenge in software development because it is almost impossible for human developers to test a huge number of various behaviors of complex target software. For example, the avionics system in F-22 Raptor consists of about 1.7 million lines of code and the radio and navigation system in S-class Mercedes-Benz has more than 20 million lines of code [10].

To address aforementioned challenges, *automated test generation techniques* have been developed to systematically and automatically generate test cases to explore (all) possible behaviors of software.

1.2 Background and Limitations on Automated Test Generation Techniques

1.2.1 Background on Automated Test Generation Techniques

Automated test generation techniques operate in two different levels: a system-level and a unit-level. Automated test generation in system-level takes a whole program as a target-under-test and generates test cases for the whole system. Bugs detected by automated test generation in system-level are usually real bugs since a program is assumed to take any inputs and sanity checking of inputs is the duty of the program. Automated test generation in unit-level generates test cases for each individual unit (i.e., functions or methods) separately. To test each unit in isolation with other units of the target program, unit test drivers and stubs are generated and used. Drivers and stubs simulate the interface between the target unit and the other units in a simple manner. Automated unit testing can examine various behaviors of individual units throughly because the behaviors of the unit can be directly controlled

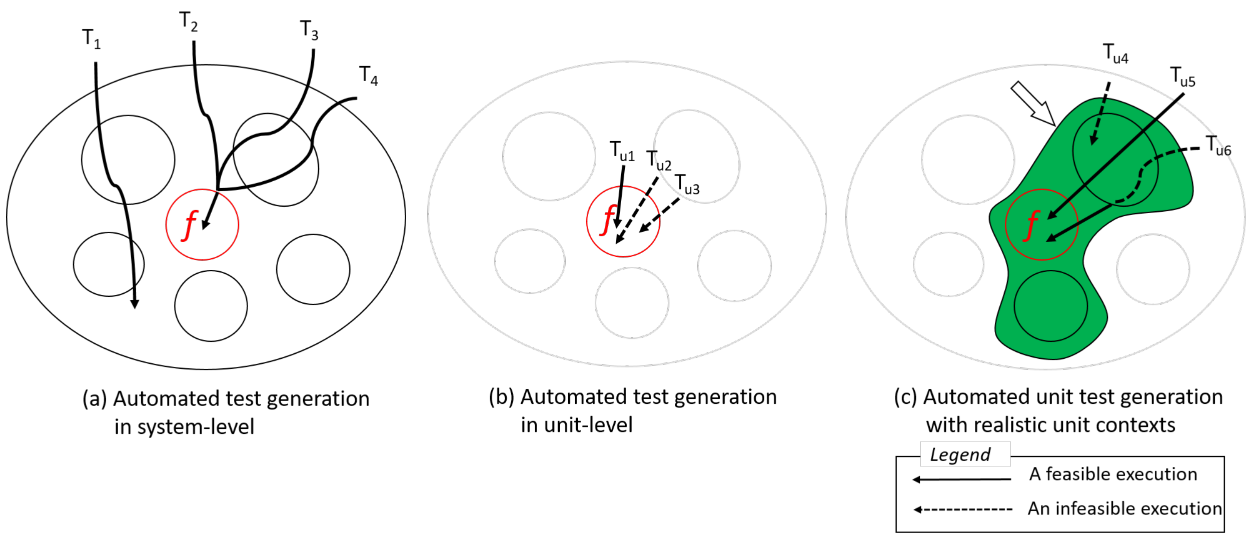

T1
T2
T3
T4
f
(a) Automated test generation in system-level
Tu1
Tu2
Tu3
f
(b) Automated test generation in unit-level
Tu4
Tu5
Tu6
f
(c) Automated unit test generation with realistic unit contexts
Legend
A feasible execution
An infeasible execution

Figure 1.1: Comparison of automated test generation in system-level and unit-level, and automated unit test generation with realistic unit contexts

through drivers and stubs. For more detail on how to automatically generate unit drivers and stubs, see Section 3.1.3.

1.2.2 Limitations on Automated Test Generation Techniques

Although automated test generation techniques can effectively detect bugs, the techniques have several limitations.

Limitation of Automated System-level Testing

System tests often do not explore diverse behaviors of a whole target program, but explore only tiny portion of possible behaviors in a limited testing time because the execution space of a whole target program is extremely large and each execution is relatively long. In addition, it is very difficult to generate system-level test cases to explore diverse behaviors of target programs since internal components of a target program is very difficult to control in fine-granularity through system level test cases (i.e., *controllability* of a target unit is low with system-level tests).

Figure 1.1 (a) illustrates automated test generation in system-level. A largest circle indicates a whole program and the six smaller circles indicate the inner components of the program. The red circle in the middle is a target function f. Suppose that we want to examine the various behaviors of f and generate four different system test cases T_1 to T_4. The four different system tests can explore only one behavior of the target component. Even T_1 does not execute the target function f, and the three different system tests T_2 to T_4 explore the same behavior of f.

Limitation of Automated Unit-level Testing

Automated test generation in unit-level suffers from false alarms, which are serious obstacles for field engineers to adopt this effective and efficient technique. Creating accurate unit test drivers/stubs is difficult because each unit has various kinds of inputs (i.e., array, nested struct, etc) on which there exist implicit constraints (e.g., an input array should be sorted for correct operation of a target unit).

Note that such implicit constraints on inputs to each unit are rarely documented in practice. Unit tests that violate such implicit constraints are infeasible executions and may raise false alarms.

Figure 1.1 (b) illustrates automated test generation in unit-level. A unit test `Tu1` executes a feasible execution path of the target function f, but `Tu2` and `Tu3` execute infeasible execution paths and may raise false alarms (i.e., these two execution paths cannot be explored in any system executions) because these two tests do not satisfy the implicit constraints of f.

1.3 Approach: Automated Unit Test Generation with Realistic Unit Context Synthesis for Low False Alarms

To overcome the limitations of automated test generation, I have developed an automated unit test generation framework with realistic unit context synthesis to reduce false alarm. My thesis statement is as follows:

Automated unit test framework that synthesizes realistic unit context can automatically detect bugs in complex real-world programs with a low false alarm rate.

To validate the thesis statement, I have developed and evaluated an automated unit-test generation framework (calling it CONCERT) which synthesizes *realistic unit contexts* to reduce false alarms.

Figure 1.1 (c) illustrates the proposed approach. The figure shows explored behaviors of automatically generated unit tests with realistic contexts. The green area which includes the target function f and two closely relevant function to f shows a realistic unit context of f. The realistic unit context filters out tests that may explore infeasible execution paths of the target function. For example, the realistic unit context filters out `Tu4` which is an infeasible test case to the context and, thus, avoids a possible false alarm caused by `Tu4`. `Tu5` is a feasible test case to the context and it explores f. `Tu6` is an infeasible test case to the context of f, but the context still refines/guides it to explore a feasible execution path of f.

CONCERT synthesizes test drivers/stubs to build realistic contexts of a target function f by utilizing the code of the other functions closely relevant to f (see Chapter 4). Then, CONCERT explores diverse and realistic unit execution scenarios by enforcing various contexts through dynamic symbolic execution of f and f's context (i.e., the closely relevant function code of f which composes the context of f). To evaluate CONCERT, I have applied CONCERT and related techniques on the 67 bugs of the 15 real-world C programs (see Section 5.2) and approved my thesis statement by demonstrating that CONCERT achieves both high bug detection ability (i.e., 83.6% of the target bug detected) and relatively low false/true alarm ratio (i.e., 2.4 false alarms per one true alarm).

1.4 Outline of Thesis

The remainder of the dissertation is structured as follows. First, in Chapter 2, I present related work for automated unit test generation techniques. Chapter 3 presents an automated unit testing framework CONBOL for large industrial embedded software. Through a case study on four million lines long embedded software, I showed that CONBOL detected 24 crash bugs which had not been detected by developers nor static analysis tools. In Chapter 4, I present a refined automated unit test generation framework CONCERT. To reduce false alarms of automated unit test generation, CONCERT synthesizes realist unit contexts by calling functions closely relevant to a target function f during unit testing. Chapter 5 presents empirical evaluation of CONCERT on 67 crash bugs in 15 real-world C programs (55KLOC on average). The experiment results show that CONCERT has high bug detection ability and low false/true alarm ratio. I conclude this dissertation in Chapter 6 with future work.

Chapter 2. Related Work

This chapter presents related work on automated unit test generation techniques, concolic testing techniques, and metrics on relevance between functions.

2.1 Automated Unit Test Generation Techniques

Automated unit test generation techniques analyzes a given unit-under-test and generates input values of units (i.e., values of parameters, global variables, and class variables in object-oriented languages) automatically. The automated unit test generation techniques can be classified into three groups according to how the inputs of units are generated: direct function input generation, method sequence generation, system execution capture.

Table 2.1 shows a list of related work on automated unit test generation techniques. Most of the related work show their bug detection ability by demonstrating how many new bugs were detected by the techniques. TestFul, OCAT, and GenUTest did not report the number of bugs detected, but report only coverage. Unlike the related work, I scientifically demonstrated the bug detection ability by showing the recall of the crash bugs (i.e., $\frac{\text{a number of detected crash bugs}}{\text{a number of total crash bugs}}$) in 15 open-source programs.

2.1.1 Direct Function Input Generation based Unit Testing Techniques

Direct function input generation based unit testing techniques generates input values of units directly. The input values are set by the assignment which are generated by the automated test generation techniques such as concolic testing [63] and the target units are tested with the input values.

Godefroid et al. [26] developed DART to automatically generate unit test drivers (but not stubs) and test inputs for C programs by using the combination of concrete execution and symbolic execution. Sen et al. [63] developed CUTE to automatically generate test inputs. Tillmann et al. [67] developed Pex for testing .NET programs. Kim et al. [36] developed CONBOL which automatically generates unit test drivers/stubs and test inputs targeting large-scale embedded software. Note that these techniques synthesize test drivers and stubs without utilizing the contexts of a target function in the target program, which may suffer false alarms due to infeasible test executions. Thus, Godefroid et al. [26] targeted public API functions of libraries to avoid the false alarm problem because public API functions usually work with all possible test inputs (i.e., no false alarm caused by infeasible unit executions). In contrast, CONCERT effectively resolves the false alarm problem by utilizing the highly correlated functions of a target function in unit test drivers/stubs.

Pasareanu et al. [52] developed Symbolic Java PathFinder (JPF) which concretely executes a target program until a target method f is reached and then starts symbolic execution on f with user-specified symbolic input setting for f. Although Symbolic JPF alleviates the false alarm problem by providing realistic contexts to f using concrete system executions, it still requires human expertise to setup symbolic input setting of f. In contrast, CONCERT generates unit test drivers/stubs and test inputs in a fully automatic way. Chakrabarti and Godefroid [9] developed a unit testing technique which statically generates partitions of functions based on a static call graph and tests each partition as a unit through symbolic execution. In this technique, functions in the same partition have the same test drivers/stubs,

Table 2.1: Related work of automated unit test generation techniques

Related work	Target language	Test generation techniques	Bug detection ability
DART [26]	C	Concolic testing	Found a bug in oSIP 2.0.9
CUTE [63]	C	Concolic testing	Found two bugs in SGLIB 1.0.1
Pex [67]	.NET	Concolic testing	Found bugs in .NET libraries
Symbolic JPF [52]	Java	Symbolic execution	Found a bug in the Onboard Abort Executive system
Chakrabarti and Godefroid [9]	C	Concolic testing	Found no bug
UC-KLEE [56]	LLVM IR	Symbolic execution	Found 67 bugs in BIND, OpenSSL, and Linux Kernel
Randoop [50]	Java	Random testing	Found 210 errors in JDK 1.5, Jakarta Commons, and .NET framework
EvoSuite [22, 23]	Java	Search-based testing	Found 1694 faults in 100 SourceForge projects
TestFul [2]	Java	Search-based testing	N/A
Garg et al. [24]	C++	Random testing and concolic testing	Found 9 bugs in gnuchess
Elbaum et al. [18]	Java	Capturing system tests	Found 7 seeded bugs in Siena
OCAT [29]	Java	Capturing system tests	N/A
GenUTest [54]	Java	Capturing system tests	N/A
CONCERT	C	Concolic testing	Found 86.7% (=56/67) of bugs in the SIR and SPEC2006 benchmarks

which may fail to generate realistic contexts for each target function f in the partition. In contrast, CONCERT generates different realistic unit test drivers/stubs for each target function based on dynamic function correlation observed at runtime. Recently, Ramos and Engler [56] developed UC-KLEE to directly start symbolic execution from the target function using lazy initialization [30]. Through the manual analysis of the thousands of alarms, they reported that UC-KLEE detected 67 new bugs and the false/true alarm ratio of crash alarms was 15.4 in `bind`, `openssl`, and Linux kernel components.

Most of the aforementioned papers just reported assert violations or bugs detected on a few target programs without scientifically evaluating bug detection ability using recall and bug detection accuracy of their techniques. In contrast, this paper rigorously evaluates both bug detection ability with recall and bug detection accuracy of CONCERT on 15 open source target programs.

2.1.2 Method Sequence Generation based Unit Testing Techniques

Instead of directly set the input values of units, method sequence generation based unit testing techniques indirectly set the input values by calling a sequence of methods. In object-oriented languages, member field variables are usually not directly set by the assignment, but manipulated through their public interface methods. The input values are constructed by calling a sequence of those public methods. The false alarm problem can be reduced because the public methods can generate unit contexts.

Pacheco et al. [50] developed Randoop to create a sequence of method calls randomly. Garg et al. [24] improved Randoop approach by generating input test cases of the generated method sequence using concolic testing. Fraser et al. developed EvoSuite [22, 23] for testing Java programs using search-based strategies. To improve the branch coverage, EvoSuite adopted symbolic execution to complement search-based testing. Baresi et al. [2] developed TestFul for testing Java programs. TestFul combined genetic algorithm and a local search (e.g., a greedy search) to improve the speed of test generation. These techniques also suffer from false alarms.

These techniques may also suffer false alarms due to infeasible test inputs/method sequences generated. For example, Garg et al. [24] reported that the false/true alarm ratio of testing `gnuchess` was 0.97. Fraser et al. [23] reported that 0.56 to 4.24 false/true alarm ratios in their experiments. CONCERT reports that 2.4 false/true alarm ratio on average over the 15 target programs.

However, it is not straight-forward to directly compare these techniques with CONCERT. This is because, in contrast to this paper which studies both bug detection accuracy and bug detection ability based on the previous bug-fix commits of the target programs, the aforementioned papers did not report bug detection ability in terms of recall, but only bug detection accuracy (i.e., false alarm ratio) of their techniques. Although the false alarm ratios of these techniques may be low, they can still miss many bugs to decrease false alarm ratio.

2.1.3 System Execution Capture based Unit Testing Techniques

Automated unit testing techniques in this category capture system execution information (e.g., method sequences or argument/return values of function calls) and generate unit tests by utilizing the captured information. Elbaum et al. [18] proposed a technique to generate unit tests from the system tests. To generate unit test inputs and oracles for a target method f, the technique captures program states before and after an invocation of f. Jaygarl et al. [29] developed OCAT which captures object instances during system execution and generates unit tests from the captured object instances. To increase test coverage, OCAT mutates object instances to cover uncovered branches. OCAT uses the captured and mutated object instances as seed objects of Randoop [50] to generate new unit test cases. Pasternak et al. developed GenUTest [54] to automatically generate unit tests and mock objects using captured method sequences during system testing.

The aforementioned techniques can be effective for regression testing of evolving software, but not effective for a single version of software. This is because the executions of the generated unit tests just replay the same behaviors [18, 54] (or similar behaviors [29]) of the target unit in already performed system testing. In contrast, CONCERT is effective for even a single version of a program because it enforces various and realistic runtime contexts to a target function f through dynamic symbolic execution of not only f but also f's drivers/stubs containing the code of closely relevant functions of f.

2.2 Concolic Testing Techniques

2.2.1 Concolic Testing Technique Research

Concolic testing is also known as dynamic symbolic execution, since it utilizes both dynamic testing and symbolic execution [37]. The core idea of concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers. Various concolic testing tools have been implemented to realize this core idea [53, 61, 8]. We can classify this work into the following three categories,, based on how they extract symbolic path formulas from concrete executions:

1. *Static instrumentation of target programs*
   The concolic testing tools in this group instrument a target source program to insert probes to extract symbolic path formulas from concrete executions at run-time . Many concolic testing tools adopt this approach, since the approach is relatively simple to implement and, consequently, convenient when attempting to apply new ideas in tools. In addition, it is easier to analyze the performance and internal behavior of the tools compared to the other approaches. In this group, CUTE [63], DART [26], CREST [6] and PathCrawler [71] target C programs, while jCUTE [62] targets Java programs.

2. *Dynamic instrumentation of target programs*
   The concolic testing tools in this group instrument a binary target program when it is loaded into memory (i.e., through a dynamic binary instrumentation technique [49]). Thus, even when the source code of a target program is not available, the target binary program can be automatically tested. In addition, this approach can detect low-level failures caused by a compiler, a linker, or a loader. SAGE [27] and Triton [60] are concolic testing tools that use this approach to detect security bugs in x86-binary programs.

3. *Instrumentation of virtual machines*
   The concolic testing tools in this group are implemented as modified virtual machines, on which target programs execute. One advantage of this approach is that the tools can exploit all execution information at run-time, since the virtual machine possesses all necessary information. Pex[68] targets C# programs that are compiled into Microsoft .Net binaries. KLEE[7] targets LLVM [41] binaries. jFuzz [28] targets Java bytecode on top of Java PathFinder [69]. BitBlaze [64] and S2E [13] use a modified version of x86/x64 virtual machine QEMU [3] to perform concolic testing on x86/x64 binary programs.

2.2.2 Case Studies of the Concolic Testing Techniques

Lakhotia et al. [40] applied CUTE (and AUSTIN [39], which is a test case generation tool based on genetic algorithms [46]) to the 387 functions of four C programs (`libogg`, `plot2d`, `time`, and `zile`) and reported testing results. Marri et al. [45] used PEX to test a client program of CodePlex (a source control management system) with an intelligent mock file system. Kim et al. [31, 34, 32] report results of applying CREST to a flash file system. Botella et al. [4] discuss limitations of current concolic techniques in practice and suggests several solutions.

For automated testing techniques, it is important to employ a proper context when generating test cases for a given target program. Otherwise, invalid test cases (which violate preconditions/assumptions of a target program) might be given to a target program and test results cannot be trusted. In other words, a test engineer should build a testing environment that can feed only valid test cases to a target program. For this purpose, concolic testing employs “mock objects” [45]/“environment model” [31] to generate test cases that satisfy “data structure invariants” [63]/“symbolic grammar” [44]/ “grammar-based constraints” [25]/“preconditions” [4] of a given target program.

2.3 Metrics on Relevance between Functions

There exist several metrics between functions to quantify relevance between functions. Chidamber and Kemerer [11] proposed a response set for classes to measure coupling metric between classes and methods. Li and Henry [43] proposed a message passing coupling metric which measures the number of method invocations in a class. Chidamber and Kemerer [12] proposed a coupling metric of two classes using the number of accesses of field variables and invocations of the methods of another class. Lee et al. [42] uses the number of method invocations of another class weighted by the number of arguments of the invoked methods.

However, these metrics have limitations to apply to reduce false alarms in automated unit testing. For example, the metric using the number of accesses to the common class field variables [12] does not capture the relation constructed by passing arguments. Lee et al. [42] considered the number of arguments passed but the number of arguments is often not a good weight because one pointer argument can pass large data structure. In addition, these metrics are static ones and reports provide too imprecise coupling value to select functions to use in the realistic unit test drivers of a target function. In contrast, CONCERT uses a dynamic function correlation metric (Section 4.2.2) which effectively reflects dependency between two functions by observing system executions of a target program.

Chapter 3. Automated Unit Test Generation for Large Scale C Programs

I, with Samsung Electronics, have developed CONcrete and symBOLic (CONBOL) testing framework targeting large industrial C programs. CONBOL automatically generates symbolic unit testing drivers/stubs and performs concolic testing on the generated unit testing drivers/stubs including target units. In addition, CONBOL implements two false alarm reduction heuristics to remove false alarms. I have demonstrated bug detection effectiveness of CONBOL by detecting 24 new crash bugs in a four million lines long smartphone software (calling it 'project S') which had not been detected by manual testing nor static analysis tools such as Coverity [14].

3.1 CONBOL Framework

3.1.1 CONBOL Overview

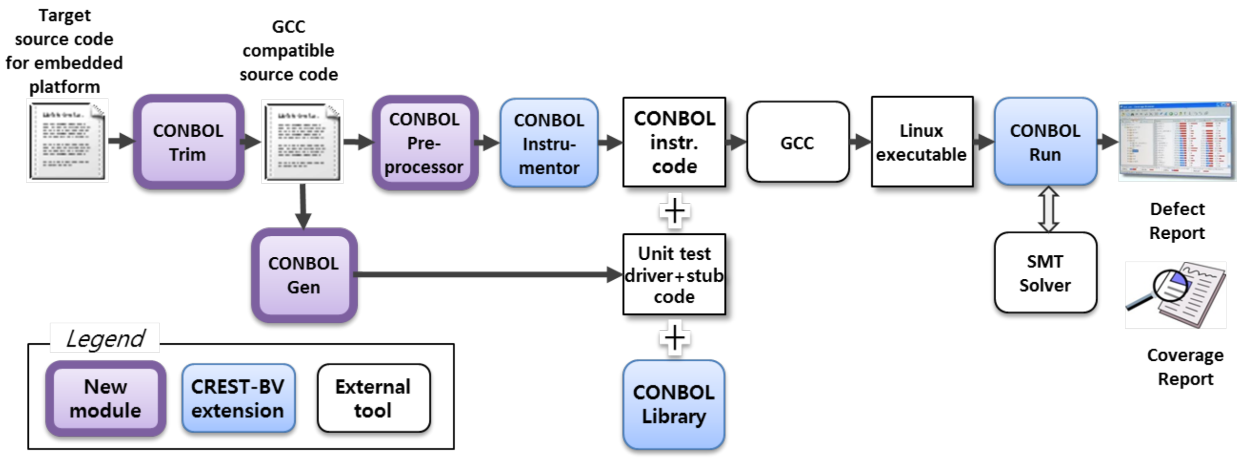


Figure 3.1: Overview of the CONBOL framework

Figure 3.1 shows the overall structure of the CONBOL framework. CONBOL is developed based on a concolic testing tool CREST-BV [35], 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 [19]. 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.

3.1.2 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.

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) [59]* 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.

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) [48] 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:

- *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.

3.1.3 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.

```
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); }
```

Figure 3.2: An example of an automatically generated unit test driver

Figure 3.2 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 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 [35]. 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.

3.2 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 3.2.1)*:
  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 3.2.2– 3.2.4)*:
  First, since the imprecise driver code that violates necessary precondition of a target function can cause false alarms [1], 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.

3.2.1 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 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.

[1]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:}
```

Figure 3.3: Test driver with preconditions

3.2.2 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 3.3 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. [2] CONBOL PP inserts such constraints of target functions automatically by using `CONBOL_assume(expr)`. [3] 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

[2]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

[3]`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.

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.

3.2.3 Scoring of Alarms

To reduce false alarms, CONBOL assigns a score to each violated assertion that CONBOL inserts (see Section 3.2.1) 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: [4]

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.

[4]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.

3.2.4 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.

3.3 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).

3.3.1 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: [5]

- 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.

3.3.2 Experimental Setup

Unit testing has several advantages to improve software quality such as early detection of bugs and corner case bug detection [58]. Unit testing has additional benefits for embedded software, since unit testing has less dependency on the target embedded platform by building testing driver/stubs [38]. Thus, we decided to apply CONBOL to the project S in unit level.

CONBOL uses reverse depth-first search strategy [6] to explore execution paths of the target unit and increase branch coverage fast. Unit testing of a target unit terminates when

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

- 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.

3.4 Case Study Results on the Project S

3.4.1 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 3.1.2). 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 [6], and constant parameters, respectively).

3.4.2 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 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. These 24 crash bugs were reported by testing with all three driver types.

[6] The average number of inserted precondition constraints for array indexes is small (i.e., 0.6) due to the three strict conditions (Section 3.2.2).

- *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 3.2.2). 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 3.1.3).

3.4.3 Coverage and Time Costs

Table 3.1 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.

Regarding the branch coverage result, we found the following reasons for the uncovered 40.4% (=100%-59.6%) of the branches. [7] First, a test execution terminated at an assertion violation and no further branches were covered in the execution (see Section 3.3.2), which makes 10.8% of the target

[7] 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 3.1 on the extended target program.

Table 3.1: 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 3.2: 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

branches uncovered. Second, functions that reach timeout were not covered completely, which makes 9.3% of the branches uncovered. [8] 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. [9]

3.4.4 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 3.2).

By inserting the precondition constraints (Section 3.2.2), 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 3.2.3), 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] In the exploratory experiments, increased timeouts did not improve the coverage much.

[9] 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 3.3.2). Thus, we think that more than 20.3 % of the target branches were uncovered due to the limitations of CONBOL.

Chapter 4. Realistic Unit Context Synthesis for Low False Alarms

A main drawback of the previous automated unit testing techniques is a large number of false alarms due to the inaccurate unit test drivers/stubs that do not represent real usage scenarios of units in a target program (see Section 1.2.2). For example, Csallner and Smaragdakis [16] and Ramos and Engler [23] report that 4.2 and 15.4 false alarms per one true alarm were generated in their automated unit testing experiments, respectively. CONBOL also raised 10.5 false alarms per one true alarm.

This false alarm issue is a serious obstacle to apply automated unit testing techniques in practice. Field engineers are sensitive to the false/true alarm ratio, because the field engineers have to analyze every alarm manually. From my experience of industrial collaboration [35, 33, 36, 51], I have observed that field engineers want to reduce false alarms even at the cost of large amount of computing resource/time.

To address the false alarm issue, I have developed an automated concolic unit testing technique CONCERT (CONColic unit tEsting using dynamic function coRrelation meTric). CONCERT automatically synthesizes unit test drivers and stubs which represent realistic contexts of a target function f (i.e., contexts which do not cause f to perform executions which are infeasible at system-level) by utilizing the source code of the other functions *closely relevant* to f in a target program. The relevance between functions is measured dynamically by observing system executions; if two functions frequently appear together in the same system executions, the relevance between these functions is considered as high (see Section 4.2.2).

4.1 A False Alarm Example Caused by Missing Unit Contexts

A target program in Figure 4.1 has a target function `f` under test (Lines 13–25). `f` takes two integer variables as inputs and calls two functions `g` (Line 20) and `h` (Line 24). Concolic unit testing techniques like CONBOL [36] analyze a target function `f`, insert the OOB assertions at Lines 18 and 22, and generate a unit test driver/stubs for `f` as shown in Figure 4.2.

The generated test driver for `f` declares two integer arguments of `f` as symbolic (Lines 4–5 in Figure 4.2). Then, the test driver invokes `f` with the two symbolic arguments (Line 6). Invocations of `g` and `h` in `f` are replaced with the invocations of the symbolic stubs `stub_g` and `stub_h` respectively. `stub_g` does nothing since `g` does not return a value (Line 8). `stub_h` returns a symbolic integer value (Lines 9–12). In addition, the assertions targeting OOB bugs (i.e., `assert(0<=x && x<5)` and `assert(0<=n && n<5)`) are inserted at Lines 18 and 22 of `f`, respectively.

Executions of the unit test driver/stubs raise alarms as follows. The assertion at Line 18 of `f` can be violated since the symbolic argument `arg1` to `f` at Line 6 of the unit test driver (Figure 4.2) can be larger than or equal to the size of `array` (i.e., five). Also, the assertion at Line 22 of can be violated because `n` can be one of 5, 7, and 9 at Line 19 as `stub_g` which is invoked instead of `g` at Line 20 does nothing.

However, all these alarms are false since `f` is called by only `b` and `b` always passes a value between zero and four as the first argument to `f` (see Lines 6–7 of Figure 4.1) and `n` at Line 19 is always less than or equal to four because of `g(&n)` invoked at Line 20. In other words, concolic unit testing techniques can raise false alarms because they generate unit test driver/stubs that are different from the real contexts

```
// x, y, and z are inputs of a target program
int main(int x, int y, int z){
    if (x > 0) return b(y,z);
    else return c(y,z);}

int b(int x, int y){
    if (0 <= x && x < 5)  return f(x,y);}

int c(int x, int y){
    return x-y;}

// Target function under test
int f(int x, int y){
    int array[5] = {1,3,5,7,9};
    int n;
    int result;

    // assert(0<=x && x<5); //To be inserted by the unit test tech.
    n = array[x];
    g(&n); // To be changed to stub_g by the unit test tech.
    if ((y % 2) != 0){
        // assert(0<=n && n<5); //To be inserted by the unit test tech.
        result = array[n];
    }  else result = h(n); //To be changed to stub_h by the unit test tech.
    return result;}

 void g(int *p){
     *p = *p / 2;}

 int h(int x){
     return x + 2;}
```

Figure 4.1: Target program with a target function `f`

of `f` which are constructed through `b` and `g`.

4.2 CONCERT Technique

This section describes CONColic unit tEsting using dynamic function coRrelation meTric (CONCERT) for effective unit testing.

4.2.1 Overview

CONCERT receives source code of a target program, a list of target functions to test, and system test cases of the target program as inputs. First, it calculates correlation between a target function

```
int test_driver_f(){
  int arg1;
  int arg2;
  SYM_int(arg1);
  SYM_int(arg2);
  f(arg1, arg2);}

void stub_g(int *p) { }
int stub_h(int x){
  int ret;
  SYM_int(ret);
  return ret;}
```

Figure 4.2: Unit test driver and stubs for `f` generated by concolic unit testing techniques

`f` and other functions in a target program based on the occurrences of the two functions in each of the system test executions. Then, based on the correlation information, CONCERT selects a subset of the closely related functions of `f` and their code is included in unit test drivers/stubs. Note that these functions closely related to `f` can be used to filter out infeasible input values to `f` and consequently reduce false alarms caused by such infeasible inputs to `f` (for example, see Figure 4.6 which utilizes correlated functions of `f` (i.e., `b` and `g`) to remove false alarms raised in Figure 4.1). Finally, CONCERT applies concolic testing to the generated test drivers/stubs to enforce various contexts to a target function so to exercise diverse and realistic target unit test executions.

Note that it is important for unit test drivers/stubs to contain only the functions that are most closely related to a target function. This is because adding more function code of a target program to unit test drivers/stubs can significantly increase the symbolic search space, which can degrade both unit testing efficiency and effectiveness. At one extreme end, a concolic unit testing technique does not use related function code at all and generates purely symbolic unit test drivers/stubs as described in Section 3.1.3, which will suffer many false alarms (see Section 4.1). At the other extreme end, a technique can use all functions of a target program in unit test drivers/stubs, which will result in system testing and lose the advantage of unit testing. Thus, it is important to include only *most closely related function code to a target function* in unit test drivers/stubs for effective unit testing. [1]

4.2.2 Overall Process of CONCERT

Figure 4.3 illustrates the four steps of CONCERT which are described in Sections 4.2.2–4.2.2.

Obtaining Function Coverage Profile from System Test Executions

This step executes a target program with given system test cases and obtains a function coverage profile with respect to the given system test cases. The function coverage profile shows which functions are executed by each of the system test cases. For example, suppose that CONCERT executes the target program in Figure 4.1 with the system test cases (0,0,0), (1,1,1), and (1,2,4). With these three test

[1] There can be multiple different ways to choose functions whose source code is included in unit testing such as using dependence of functions, using characteristics of functions, and/or using distances of functions in a call graph.

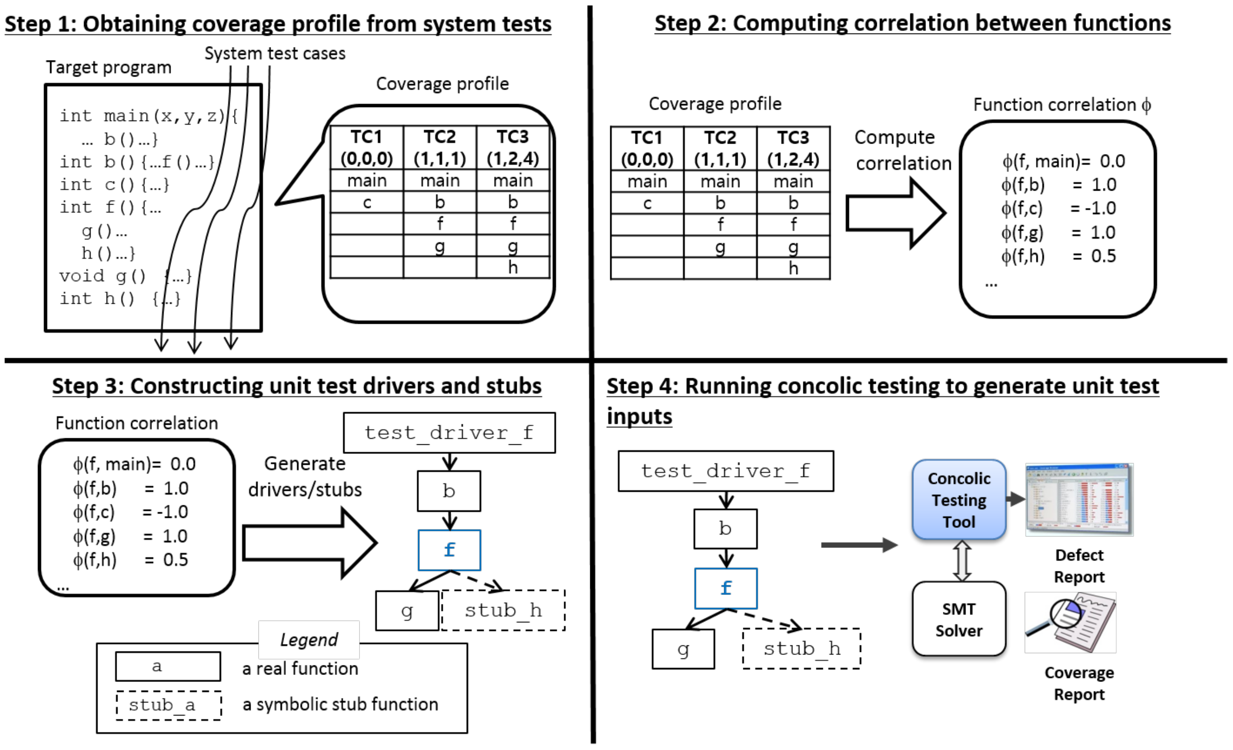


Figure 4.3: Overall process of CONCERT

Table 4.1: Function coverage profile of Figure 4.1 with the given three system test cases

System TC:(x, y, z)	TC1:(0,0,0)	TC2:(1,1,1)	TC3:(1,2,4)
Executed functions	`main`, `c`	`main`, `b`, `f`, `g`	`main`, `b`, `f`, `g`, `h`

cases, the target program starting from `main` will cover {`main`, `c`}, {`main`, `b`, `f`, `g`}, and {`main`, `b`, `f`, `g`, `h`} respectively as shown in Table 4.1.

Computing Correlation between Functions

This step computes correlation between functions in a target program. Suppose that a target program has $n(= \alpha + \beta + \gamma + \delta)$ system test executions on two functions `f` and `g` where α, β, γ, and δ are the numbers of system test executions that execute both `f` and `g`, only `f`, only `g`, and neither `f` nor `g`, respectively. Based on the observations of the two functions in the system executions, we compute ϕ *correlation coefficient* [15] using the following formula:

$$\phi(\mathtt{f}, \mathtt{g}) = \frac{\alpha\delta - \beta\gamma}{\sqrt{(\alpha+\beta)(\gamma+\delta)(\alpha+\gamma)(\beta+\delta)}} \tag{4.1}$$

This ϕ coefficient indicates the degree of correlation between `f` and `g`; higher ϕ coefficient means `f` and `g` are more closely related. [2] The values of $\phi(\mathtt{f}, \mathtt{g})$ range from -1 (all system tests execute exclusively either `f` or `g`) to +1 (all system tests execute either both `f` and `g` or none of `f` and `g`).

For example, Table 4.2 shows the correlation between `f` and other functions based on Table 4.1. First column shows the function name whose correlation with `f` is computed. Second to fifth columns

[2] ϕ coefficient is a special form of the Pearson correlation coefficient [55] for the two binary variables. ϕ coefficient cannot be computed for a function which is always executed (e.g., `main`) or never executed with given system test cases. We treat such functions have no positive or negative correlation with other functions and assign 0 to the coefficient value.

Table 4.2: Correlation between `f` and other functions based on Table 4.1

Function	α	β	γ	δ	ϕ coefficient
`main`	2	0	1	0	0.0
`b`	2	0	0	1	1.0
`c`	0	2	1	0	-1.0
`g`	2	0	0	1	1.0
`h`	1	1	0	1	0.5

show the number of the observations α to δ. The last column shows the correlation coefficient value of each function with `f`. For example, we compute the correlation between `f` and `h` as 0.5 as follows. The test case (0,0,0) executes none of `f` and `h` (i.e., δ is one in the fifth column of the last row in Table 4.2), (1,1,1) executes only `f` (β is one in the third column of the last row), and (1,2,4) executes both `f` and `h` (α is one in the second column of the last row). Using the values of α to δ, we can compute the correlation between `f` and `h` as 0.5 ($\phi(\texttt{f},\texttt{h}) = \frac{1-0}{\sqrt{2\times1\times1\times2}} = 0.5$).

Constructing Unit Test Drivers and Stubs

CONCERT generates unit test drivers/stubs for each target function `f` by including code of the closely related functions of `f`; all other functions reachable from r_i are replaced with symbolic stubs in the test driver. It selects the closely related functions of `f` as follows:

1. *Static call-graph construction:*

   CONCERT constructs a static call graph of a target program. (e.g., Figure 4.5 for the program in Figure 4.1). In a static call graph, we call node n_1 as a predecessor of another node n_2 if there exists a path from n_1 to n_2 (we call n_2 as a successor of n_1).

2. *Identifying the roots of a target function f*

   In the static call graph, CONCERT identifies the predecessor functions of f (calling them *roots* of f) such that all nodes in the path from the root r_i to f have *high correlations* with f and the all immediate predecessors of the root r_i have low correlation with f (see Algorithm 1 for the detail). This r_i serves as a root function to provide realistic contexts to f by calling other relevant functions to f. For example, $n2$ and $n6$ are identified as roots of f in Figure 4.4 where each node represents a function and the label of node shows its function name and its correlation with f.

3. *Selecting the successor functions of r_i which are closely related with f*

   For each root r_i, CONCERT selects the successor functions s_{ij} of r_i such that all nodes in the path from r_i to s_{ij} have high correlation with f (see Algorithm 2 for the detail). Intuitively speaking, this step selects closely related functions of f that can be directly or transitively invoked from r_i in the unit test driver. For example, CONCERT selects $\{n2, n4, n5, f, n8, n9, n12\}$ for a root $n2$ of f in Figure 4.4(a).

   Our conjecture is that s_{ij} can contribute to provide realistic contexts to f (e.g., through global data structure updates) since s_{ij} is closely related with f as observed in the given system test cases. For example in Figure 4.4, $n2$ may calculate and pass an argument x to $n5$ and $n5$ may update y using x and then pass y as an argument to a target function f under test (i.e., $n2$ and

$n5$ make a context to f through parameters). Or $n4$ and $n8$ may affect f by updating a global variable z which f uses. Or $n9$ may affect f if f passes a pointer *pt* to $n9$ and $n9$ updates the variable pointed by *pt*.

Running Concolic Testing to Generate Test Inputs

This step applies concolic testing to the generated test drivers and stubs to generate test inputs. CONCERT utilizes a target-oriented search strategy to explore various behaviors of a target function `f` first, and then its driver/stub functions (e.g., `b` and `g` in Figure 4.1) (see Section 4.2.4).

4.2.3 Algorithms to Generate Unit Test Drivers and Stubs

A static function call graph $G(V, E)$ is a directed graph where V is a set of nodes representing functions in a program and E is a relation $V \times V$. Each edge $(f, g) \in E$ indicates that f directly calls g. f is a predecessor of g if there exists a path from the node representing f to the node representing g. Similarly, f is a successor of g if there exists a path from g to f.

Algorithm 1 takes a call graph G, a target function f, correlation ϕ between f and the other functions, and a correlation threshold τ. It selects the root functions of f and the closely related successor functions of each root function r_i, and then returns *Roots* which contains the root functions of f and *Related* which maps r_i to a set of the functions in every path from r_i to the successor functions s_{ij} of r_i such that all nodes in the path from r_i to s_{ij} have high correlation with f for each r_i.

At the beginning of Algorithm 1, f is inserted into the initialized job queue *Queue* (Line 6) and the initialized set of visited nodes $Visited$ (Line 7). Then the algorithm iterates until *Queue* becomes empty (Line 8) to collect the root functions of f. In the **while** loop (Lines 8–21), the algorithm pops a function node v (Line 9) from *Queue*. At the first iteration, v is f. Then, for each predecessor node u of v (Lines 11–17), u is added to *Queue* (Line 13) and $Visited$ (Line 14) for further graph traversal if $\phi(f, u) \geq \tau$ and u is not yet visited (Line 12). In such case, the flag *isRoot* is set to false because v cannot be a root of f (Line 15). If *isRoot* is true after visiting all immediate predecessors of v in the **foreach** loop (Lines 11–17), v is designated as a root function of f (Line 19). Once *Roots* is obtained completely (i.e., the **while** loop terminates), the algorithm collects the successor functions of each root $r_i \in$ *Roots* which are closely related to f by calling $GetSuccessors$ (Lines 22–24). Finally, the algorithm returns *Roots* and *Related* such that *Related*$[r_i]$ is a set of the functions which have high correlation with f and can be directly or transitively invoked from r_i in the unit test driver.

Algorithm 2 ($GetSuccessors(G, f, r_i, \phi, \tau)$) selects successor functions s_{ij} of a root function r_i such that every node in the path from r_i to s_{ij} have high correlation with `f`. Then, it returns *Successors* which contains all selected successor functions.

For example, Figure 4.4 shows two groups of functions in a call graph whose source code is used in two unit test drivers/stubs for a target function f. Each node represents a function and the label of a node consists of the function name n and the correlation between the function and f. Suppose that the correlation threshold τ is 0.7. Then, $n2$ and $n6$ are selected as the roots of f and CONCERT generates two unit test drivers/stubs each of which invokes $n2$ and $n6$ respectively. For example, $n2$ is a root of f because

- The correlation between $n2$ and f=0.8 $\geq \tau$, and
- The only middle node in the path from $n2$ to f is $n5$ which has high correlation with f (i.e., 0.9), and

```
Input: a call graph $G(V, E)$, a target function $f \in V$, a mapping of function correlation
       $\phi : V \times V \to [-1, 1]$, and a correlation threshold $\tau$
Output: a set of root functions $Roots$ and a mapping $Related : Roots \to \mathcal{P}(V)$
1  $GetFunctionsToUse(G, f, \phi, \tau)\{$
2  $Roots \leftarrow empty$
3  $Related \leftarrow empty$
4  $Queue \leftarrow empty$
5  $Visited \leftarrow empty$
6  $Queue.push(f)$;
7  $Visited.add(f)$;
8  while $Queue.isNotEmpty$ do
9      $v \leftarrow Queue.pop()$;
10     $isRoot \leftarrow true$;
11     foreach $u \in V$ such that $(u, v) \in E$ do
12         if $\phi(f, u) \geq \tau$ and $u \notin Visited$ then
13             $Queue.push(u)$;
14             $Visited.add(u)$;
15             $isRoot \leftarrow false$;
16         end
17     end
18     if $isRoot = true$ then
19         $Roots.add(v)$;
20     end
21 end
22 foreach $r_i \in Roots$ do
23     $Related[r_i] \leftarrow GetSuccessors(G, f, r_i, \phi, \tau)$;
24 end
25 return $Roots$ and $Related$;}
```

Algorithm 1: Algorithm to obtain the roots of f and the successors of each root that have high correlation with f

- $n2$ has only one immediate predecessor $n1$ which has low correlation with f (i.e., 0.5).

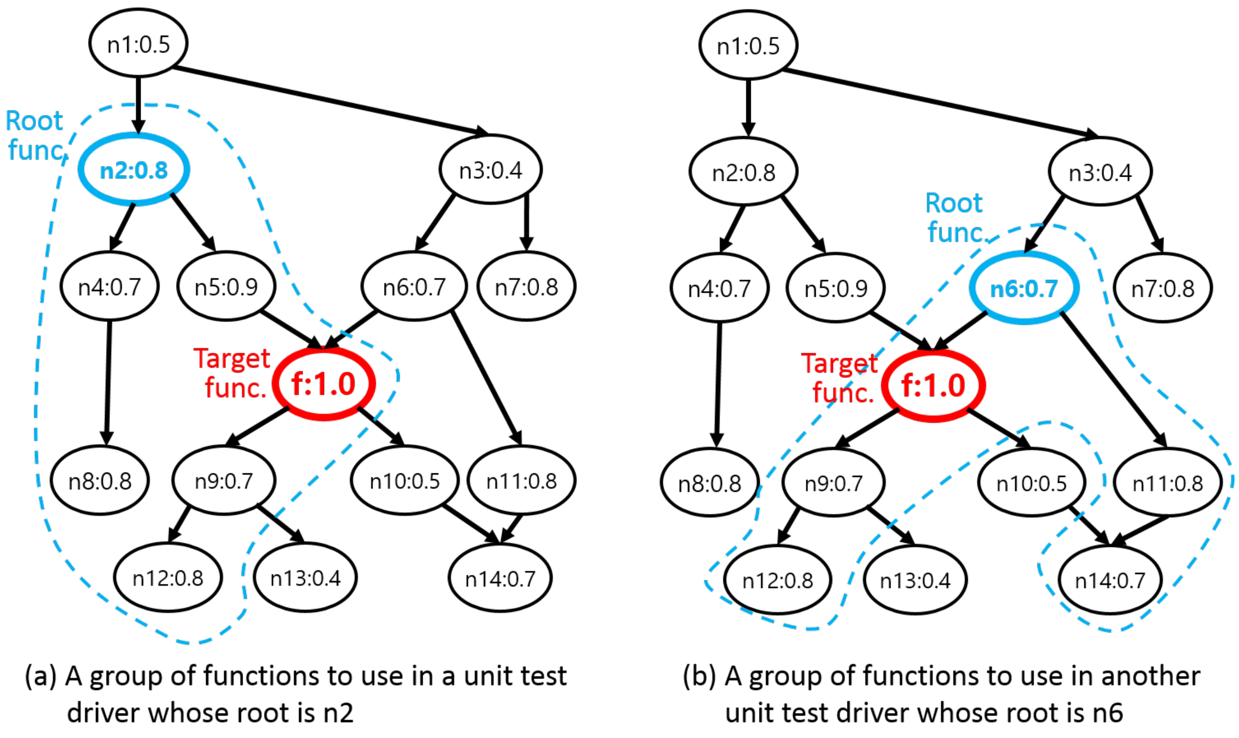
Finally, $GetSuccessors(G, f, n2, \phi, \tau)$ (Algorithm 2) returns $Successors = \{n2, n4, n5, f, n8, n9, n12\}$. Thus, the source code of the functions enclosed in the dotted line in Figure 4.4(a) are used in the unit test driver which invokes the root $n2$. Similarly, $n6$ is another root function of f and the functions enclosed in the dotted line in Figure 4.4(b) are used in another unit test driver which invokes $n6$.

An Example of Test Drivers/Stubs Generated by CONCERT

Figure 4.1 shows a target program which has `main`, `b`, `c`, a target function `f` (Lines 13–25), `g`, and `h`. Figure 4.5 shows a function call graph of the target program with corresponding correlations with `f` based on the three test cases in Table 4.1. Suppose that the correlation threshold is 0.7. Algorithm 1 returns $Roots = \{$`b`$\}$ and $Related[$`b`$] = \{$`b`, `f`, `g`$\}$ for Figure 4.1. In the target program in Figure 4.1, `b` is the only root of `f` whose correlation with `f` is higher than the correlation threshold 0.7 and `f` and `g`

```
Input: a call graph $G(V,E)$, a target function $f \in V$, a root function $r_i \in V$, a mapping of
       function correlation $\phi : V \times V \to [-1,1]$, and a correlation threshold $\tau$
Output: a set of successor functions $Successors$
GetSuccessors($G, f, r_i, \phi, \tau$){
Successors $\leftarrow r_i$;
Queue $\leftarrow$ empty;
Queue.push($r_i$);
while Queue.isNotEmpty do
    $u$ = Queue.pop();
    foreach $v \in V$ such that $(u,v) \in E$ do
        if $\phi(f,v) \geq \tau$ and $v \notin Successors$ then
            Successors.add($v$);
            Queue.push($v$);
        end
    end
end
return Successors;}
```

Algorithm 2: Algorithm to obtain successors of r_i that have high correlation with a target function f



n1:0.5
Root func.
n2:0.8
n3:0.4
n4:0.7
n5:0.9
n6:0.7
n7:0.8
Target func.
f:1.0
n8:0.8
n9:0.7
n10:0.5
n11:0.8
n12:0.8
n13:0.4
n14:0.7
(a) A group of functions to use in a unit test driver whose root is n2
(b) A group of functions to use in another unit test driver whose root is n6

Figure 4.4: Two groups of functions whose code is used in two different unit test drivers/stubs with a correlation threshold $\tau = 0.7$

are the only successor functions of the root `b` whose correlations with `f` are higher than 0.7. Thus, the generated unit test driver/stubs contain the source code of `b`, `f`, `g`, and `stub_h` (instead of `h`). Finally, the test driver invokes the root function `b` with symbolic inputs. Figure 4.6 shows the unit test driver

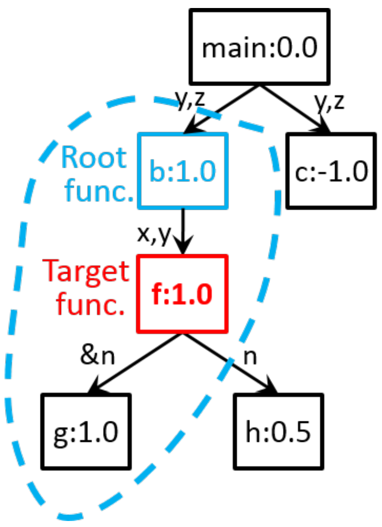


Figure 4.5: Static call graph (with correlation with `f`) of the target program in Figure 4.1

```
int test_driver_f{
  int param1;
  int param2;
  SYM_int(arg1);
  SYM_int(arg2);
  b(arg1, arg2);}

int b(int x, int y){...}

int f(int x, int y){...}

void g(int *p){...}

int stub_h(int x){
  int ret;
  SYM_int(ret);
  return ret;}
```

Figure 4.6: Unit test driver for `f` in Figure 4.1 generated by CONCERT

for `f` generated by CONCERT. The unit test driver calls `b` with symbolic arguments and then `b` calls `f` with realistic arguments constructed by `b`. Also, `f` calls real `g` (not `stub_g`) at Line 20 in Figure 4.1 and uses a realistic `n` value obtained from `g` at Lines 23 and 24.

Note that, unlike the previous concolic unit testing technique which suffers from the false alarms , CONCERT does not raise false alarms in this example. This is because CONCERT generates a unit test driver/stubs which invokes `b` to provide realistic arguments to `f` and uses `g` instead of the dummy symbolic stub `stub_g`.

```
01:void f(int x, int y){//x, y are symbolic variables
02:  if (x == 0){
03:    g(y);
04:  }else{ ... }}
05:
06:void g(int y){
07:  for (int i=0; i<y; i++){
08:    ...
09:  }}
```

Figure 4.7: An example that DFS fails to increase the branch coverage of a target function `f`

4.2.4 Target-oriented Search Strategy for Target Units

The DFS search strategy has a limitation for CONCERT because it does not distinguish the branches of a target function `f` from the branches of `f`'s driver/stub functions and may result in low branch coverage of `f`. Figure 4.7 shows an example that DFS does not cover the `else` branch of a target function `f` (Line 4) because it explores the search space of the closely related `g` first. Suppose that the initial test case for `f` is (0,0). The first symbolic path formula is $x = 0 \wedge 0 \not< y$. The DFS strategy negates the last condition of the previously generated symbolic path formula and solves the formula $x = 0 \wedge 0 < y$ to generate the test case (0,1). With (0,1), the secondly generated symbolic path formula is $x = 0 \wedge 0 < y \wedge 1 \not< y$. Negating the last condition of the secondly generated symbolic path formula will generate the test case (0,2) and this step repeats indefinitely without covering the `else` branch at Line 4.

To overcome this limitation, we have developed a target-oriented search strategy to give high priority to cover the branches of a target function. While the DFS strategy uses one stack to store the branches to negate in the next iteration, target-oriented search uses two stacks to keep the branches to negate. One stack (calling it a priority stack) stores the branch conditions to negate in the target function `f` and the other stack (calling it a normal stack) stores the branch conditions to negate in the other functions. target-oriented search always negates the branch conditions in the priority stack first and negates the branch conditions in the normal stack only when the priority stack is empty. For example, for the first symbolic path formula of Figure 4.7, target-oriented search strategy stores the branch condition $x = 0$ of `f` to the priority stack and the branch condition $0 \not< y$ of `g` in the normal stack. Thus, since $x = 0$ is negated before $0 \not< y$, target-oriented search covers the `else` branch in `f`.

4.3 False Alarm Reduction and Alarm Prioritization Heuristics

This section explains the two static false alarm reduction heuristics (Sections 4.3.1 and 4.3.2), one dynamic false alarm reduction heuristic (i.e., the common likely-invariant heuristic in Section 4.3.3), and two alarm prioritization heuristics (Section 4.3.4). The aforementioned two static false alarm reduction heuristics and the two alarm prioritization heuristics are general ones (i.e., applicable to not only CONCERT but also other automated unit testing techniques). However, the common likely-invariant heuristic is closely related to CONCERT. This is because the heuristic utilizes multiple drivers/stubs of a target function `f` to focus on likely-feasible test executions of `f`.

4.3.1 False Alarm Reduction by Keeping Consistency Between Allocated Memory and Its Size Variable

CONCERT identifies and utilizes implicit consistency between the size of allocated memory and a variable representing the size of the memory to generate constraints to reduce false alarms caused by the inconsistency.

For example, suppose that a root function `r` receives an input pointer `ptr` to allocated memory and `ptr_size` which represents the size of the memory as parameters (e.g., `void r(int *ptr, int ptr_size)`). A unit test driver will dynamically allocates n bytes for `ptr` (Section 3.1.3) for automated unit testing. If $n \neq$ `ptr_size`, the corresponding unit test execution may be infeasible due to the inconsistency which does not occur at real system executions. Thus, a unit test driver should recognize the relation between `ptr` and `ptr_size` and dynamically allocate memory whose size is `ptr_size`.

For that purpose, based on the names of variables, CONCERT identifies integer input variables of a root function r which may represent size of allocated memory among function parameters, global variables, and `struct` members as follows. If an integer variable name contains "`len`", or "`size`" (e.g., `ptr_size`), CONCERT treats the variable as a size variable. Then, CONCERT finds a corresponding pointer input variable of `r` (i.e., parameters, global variables, and `struct` members) whose name has the longest common substring of the size input variable name (e.g., CONCERT recognizes the relation between `ptr` and `ptr_size` because their variable names are similar). [3]

4.3.2 False Alarm Reduction by Value Range Analysis

To avoid false alarms caused by infeasible integer inputs, CONCERT uses a static value range analyzer [57]. The static value analyzer analyzes not only functions included in a unit test, but also a whole program to obtain possible value ranges of input variables to a target function (i.e., global variables and function parameters) in a sound manner. When CONCERT generates test drivers, CONCERT adds `SYM_assume(expr)` [4] at the beginning of a target function `f` where `expr` represents possible value ranges of input variables of `f`. If an input value to `f` is not in the possible range, a current execution terminates immediately without generating infeasible test input and concolic testing tries the next test execution.

4.3.3 False Alarm Reduction by Common Likely-Invariants of Unit Test Drivers

This heuristic utilizes dynamic invariant generation (e.g., Daikon [20]) to infer a *common likely-invariant* of various unit test drivers which corresponds to realistic context of a target function f. Suppose that f has multiple test drivers/stubs drv_i^fs. During the test generation, Daikon infers a likely-invariant ϕ_i^f of drv_i^f at the entry of f. Since each of the unit test drivers drv_i^f serves only partial and approximate context of f, ϕ_i^fs can be different each other.

A main idea is that the execution σ of f that satisfies a common likely-invariant of the unit test drivers $\Phi_f (= \bigwedge_i \phi_i^f)$ is likely feasible since σ satisfies all likely-invariants of various test drivers/contexts of f. Thus, CONCERT reports only alarms raised from the executions that satisfy Φ_f at the entry of f.

However, a common likely-invariant Φ_f can be unsatisfiable because each of test drivers/stubs drv_i^fs has only partial context of `f`. Figure 4.8 shows an example of an unsatisfiable Φ_f. Unit test drivers `drv1`

[3]Currently, CONCERT utilizes only a subset of popular C coding conventions. We plan to extend CONCERT to conveniently adopt more coding conventions.

[4]`SYM_assume(expr)` is a macro of `if(!expr) exit(0);`.

```
01: void drv1(){
02:   int x;
03:   SYM_int(x);
04:   even(x);}
05:
06: void drv2(){
07:   int x;
08:   SYM_int(x);
09:   odd(x);}
00:
11: void even(int n){
12:   if (n%2==0){
13:     f(n);}}
14:
15: void odd(int n){
16:   if (n%2==1){
17:     f(n);}}
18:
19: void f(int n){
20:   ...
21: }
```

Figure 4.8: An example of an unsatisfiable common likely-invariant Φ_f

(lines 1–4) and `drv2` (lines 6–9) call `even` (line 4) and `odd` (line 9), respectively. `even` (lines 11–13) calls a target function `f` (line 13) if its argument is even. `odd` (lines 15–17) calls `f` (line 17) if its argument is odd. Suppose that we have likely-invariants of `drv1` and `drv2` at the entry of f as $n\%2 = 0$ and $n\%2 = 1$, respectively. Then, Φ_f (i.e., $n\%2 = 0 \wedge n\%2 = 1$) is unsatisfiable.

If Φ_f is unsatisfiable, a minimal unsatisfiable core ψ_f of Φ_f is calculated by using Z3 [47]. [5] Then, CONCERT finds a clause c in ψ_f such that the number of ϕ_i^fs that contain c is the smallest and removes ϕ_i^fs that contain c from Φ_f (calling it Φ'_f). In this way, CONCERT can get a satisfiable common likely-invariant Φ'_f which is satisfiable in the largest number of unit driver contexts. Finally, CONCERT applies Φ'_f instead of Φ_f to filter alarms.

Figure 4.9 shows a diagram of input domain of a target function f and the sets of input values which satisfy likely-invariants of four test drivers drv_1^f to drv_4^f at the entry of f (i.e., ϕ_1^f to ϕ_4^f); each ellipse represents a set of input values of f that satisfy a corresponding likely-invariant ϕ_i^f. Note that a common likely-invariant Φ_f is unsatisfiable because the ellipses satisfying ϕ_1^f, ϕ_2^f, and ϕ_3^f have common intersection (represented as dark area in Figure 4.9) but the ellipse satisfying ϕ_4^f does not intersect with those of ϕ_1^f and ϕ_2^f. Thus, to obtain a satisfiable common likely-invariant, CONCERT removes ϕ_4^f from Φ_f and reports only alarms satisfying $\Phi'_f = \phi_1^f \wedge \phi_2^f \wedge \phi_3^f$.

[5] For a given unsatisfiable formula α in CNF, an unsatisfiable core ψ is a subset of clauses of α which is still unsatisfiable. An unsatisfiable core ψ is minimal if all proper subsets of ψ is satisfiable.

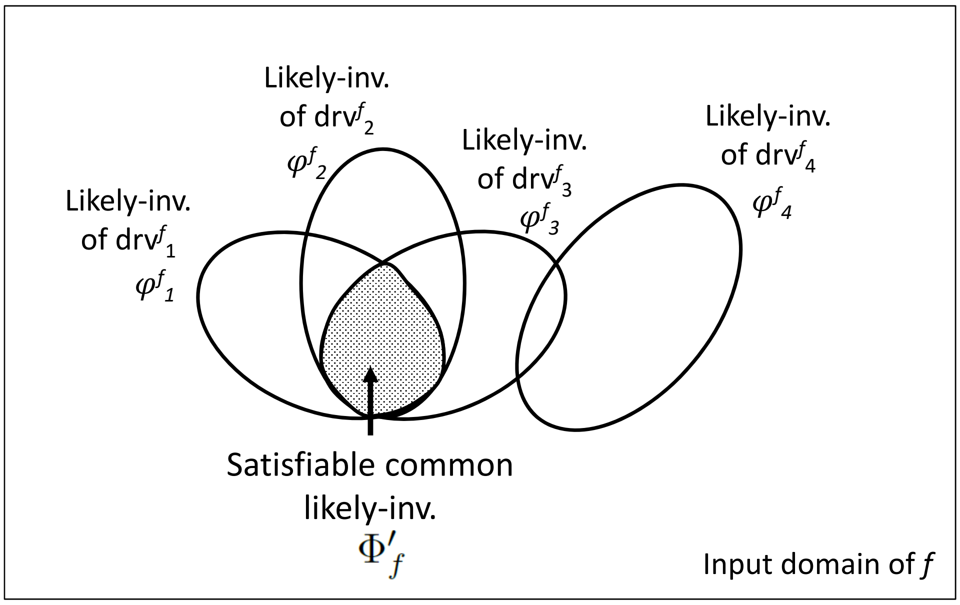

Likely-inv. of drv^f_2 φ^f_2
Likely-inv. of drv^f_3 φ^f_3
Likely-inv. of drv^f_4 φ^f_4
Likely-inv. of drv^f_1 φ^f_1
Satisfiable common likely-inv. Φ'_f
Input domain of f

Figure 4.9: A diagram of input domain of f which shows likely-invariants of four test drivers ϕ^f_1 to ϕ^f_4 and their satisfiable common likely-invariant Φ'_f

4.3.4 Alarm Prioritization using Code Complexity and External Input

CONCERT gives high priority to the alarms raised from complex target functions whose cyclomatic complexity are high because complex functions are difficult to write correctly and tend to have bugs. [6] In addition, CONCERT assigns high priority to the alarms raised after external input read functions were invoked (e.g., file system read APIs such as `fread`, `fscanf`, and `fgetc`). This is because the external input can be an arbitrary value and missing sanity check of external inputs is known as a main cause of crash bugs.

[6]This alarm prioritization heuristic might not work for domains where code complexity is tightly controlled like safety critical systems (e.g., avionics, nuclear power plants, etc). In such domains, the difference of code complexity between functions might not be meaningful and the code complexity might not be correlated with the probability of bugs.

Chapter 5. Empirical Evaluation of Automated Unit Test Generation with Realistic Unit Context

This chapter presents an empirical evaluation on CONCERT. To evaluate the bug detection ability and the false/true alarm ratio of CONCERT, I have applied CONCERT together with the other concolic unit testing techniques to the 15 real-world open source C programs (i.e., the SIR [17] and SPEC2006 [65] benchmarks). Through the experiments targeting crash bugs of the target programs, CONCERT demonstrates both high bug detection ability (i.e., it detects 83.6% of all target bugs) and relatively low false/true alarm ratio (i.e., 2.4 false alarms per one true alarm). Compared to the previous concolic unit testing techniques, CONCERT reduces the number of false alarms significantly while maintaining the comparable bug detection ability.

5.1 Experiment Setup

To evaluate the effectiveness of CONCERT as an automated unit testing technique, we have designed the three research questions (Section 5.1.1) and compared CONCERT with other concolic unit testing techniques (Section 5.1.3) on the 15 target programs (Section 5.1.2). Section 5.1.4 describes what we measured to answer the research questions and Section 5.1.5 describes a setup of our testbed. Section 5.1.6 discusses threats to validity of the experiment.

5.1.1 Research Questions

RQ1. Bug Detection Ability: *How many bugs does CONCERT detect compared to the other concolic unit testing techniques?*
RQ1 evaluates the bug detection ability of the concolic unit testing techniques to show the effectiveness as testing techniques.

RQ2. Bug Detection Accuracy: *How much false/true alarm ratio does CONCERT decrease compared to the other techniques?*
RQ2 evaluates the accuracy of the concolic unit testing techniques to detect target bugs in terms of false/true alarm ratio (i.e., $\frac{\text{a number of false alarms}}{\text{a number of true alarms}}$).

RQ3. Effects of the False Alarm Reduction and Alarm Prioritization Heuristics: *How much false/true alarm ratio do the false alarm reduction and alarm prioritization heuristics decrease? Also, how much do the false alarm reduction and alarm prioritization heuristics affect the bug detection ability?*
RQ3 evaluates the impact of the false alarm reduction and alarm prioritization heuristics (Section 4.3) to the bug detection accuracy and the bug detection ability of CONCERT.

5.1.2 Target Bugs and Programs

We target the crash bugs because crash bugs are very serious problems in software systems, and we can detect a crash automatically without user-given test oracles which are rarely available in the target programs. Also, we target the crash bugs that were confirmed by the original developers through the bug-fix commits (which have been reported since the benchmark program version was released until

Table 5.1: Target bugs and programs

Target programs and version	Lines	# of functions	# of sys. test cases	Branch coverage (%)	# of target bugs	Description
Bash-2.0	32714	1214	11	46.2%	6	Shell interpreter
Flex-2.4.3	7471	147	567	45.7%	2	Lexical analyzer generator
Grep-2.0	5956	132	809	50.3%	5	Pattern matcher
Gzip-1.0.7	3054	82	214	55.8%	2	Compression utility
Make-3.75	28715	555	1043	64.5%	3	Build script interpreter
Sed-1.17	4085	73	360	47.3%	2	Stream editor
Vim-5.0	66209	1749	975	35.8%	6	Text editor
Perl-5.8.7	79873	2240	1201	52.3%	6	Perl interpreter
Bzip2-1.0.3	4737	114	6	67.4%	2	Compression utility
Gcc-3.2	342561	5553	9	43.7%	15	C compiler
Gobmk-3.3.14	154583	2682	1354	65.2%	5	Go game program
Hmmer-2.0.42	35992	539	4	75.6%	3	Gene sequence analyzer
Sjeng-11.2	10146	144	3	77.9%	2	Chess game program
Libquantum-0.2.4	2255	101	3	68.5%	3	Quantum computing simulator
H264ref-9.3	51578	590	6	63.6%	5	H264 encoder/decoder
Average	55328.6	1061.0	437.7	57.3%	4.5	

December 2015 (i.e., the time of writing this paper)) and can be detected by unit testing (i.e., both the buggy statements and the violated assertions are located in the same function `f`). Note that CONCERT can detect not only the crash bugs, but the violation of functional specification if assert statements to check the functional correctness are given.

The experiment targets all programs in the SIR [17] and SPEC2006 [65] benchmarks. Table 5.1 describes the 15 open source target programs, including their sizes in code lines, the number of the functions to test, the number of the system test cases used, branch coverage achieved by executing the system test cases, the numbers of the target crash bugs, and the brief description of the target programs.

`Bash` to `Vim` are taken from the SIR benchmark. `Perl-5.8.7` to `H264ref-9.3` are taken from the SPEC2006 benchmark. The experiment does not use `mcf-1.2` in the SPEC2006 benchmark because it has only one system test case. CONCERT does not generate realistic unit drivers/stubs of a target function for such case (i.e., all executed functions are considered closely relevant each other if there is only one system test case). For all target programs, we used all system test cases provided in the benchmarks. The target programs have two to fifteen target crash bugs (4.5 on average) (Section 5.1.4 explains how we select target crash bugs).

5.1.3 Concolic Unit Testing Techniques

We have compared the following five concolic unit testing techniques.

- *Symbolic unit testing (SUT)*: It generates a symbolic unit testing driver with symbolic arguments and symbolic global variables with no constraints on the symbolic values. Also, SUT uses symbolic stubs to replace all functions called by a target function `f` (see Section 3.1.3).

- *Symbolic unit testing with random alarm selection (SUT_R)*: This technique is same to SUT in terms of test generation, but it only reports randomly selected alarms; it selects the same number of alarms reported by CONCERT per target program. For example, since CONCERT reports 540 alarms from `bash-2.0` (=192 true alarms+348 false alarms as shown in the last column of Table 5.8), SUT_R randomly selects 540 alarms out of all alarms reported by SUT.

  We compare CONCERT with SUT_R to check if CONCERT is more effective as a unit testing technique than the random alarm selection (i.e., the number of detected bugs of CONCERT is larger than SUT_R and the false/true alarm ratio of CONCERT is lower than SUT_R). We repeat the testing runs 30 times to minimize the random effect.

- *CONBOL:* It is an improved version of SUT with false alarm reduction heuristics and alarm scoring rule [36]. We have developed a CONBOL prototype on top of SUT with the heuristics and the alarm scoring rule described in [36].

- *$CONCERT_0$*: It is similar to CONCERT except that CONCERT_0 does not use the false alarm reduction and alarm prioritization heuristics described in Section 4.3.

- *CONCERT*: It applies all false alarm reduction heuristics and alarm prioritization heuristics. The alarm prioritization heuristics report only alarms raised in the functions whose cyclomatic complexity are top 20% high in the target program or alarms raised after external input handling functions are invoked (see Section 4.3.4).

We implemented CONCERT and the other concolic unit testing techniques in 5300 lines of C++ code based on CREST-BV [35]. To generate test drivers and stubs, CONCERT uses `Clang/LLVM-3.4` [41]. CONCERT uses DAIKON [20] and Z3 [47] to infer likely-invariants of unit drivers and the LLVM-based static variable range analysis tool [57] to compute the possible ranges of variables.

5.1.4 Measurement

Regarding RQ1 and RQ3, we measure the number of target bugs detected by each of the concolic unit testing techniques.

We consider that a target bug is detected if a technique generates a unit test execution that covers one of the buggy statements in a target unit under test and violates the inserted assertions in the target unit. [1] To identify the buggy statements, we manually analyzed all crash bug-fix commits of the all subsequently releases of the target programs included in the SIR and SPEC2006 benchmark programs. [2] We consider that a statement s of a target program is a buggy statement if s corresponds to the changed/fixed statements in a crash bug-fix commit. For example, we have reviewed total 37 crash bug-fix commits included in the change-logs for `bash`. Among the 37 reported crash bugs, only six crash bugs can be detected by unit testing (i.e., the buggy statement and the assertion violation site exist in a same target function) and we target these six crash bugs for `bash-2.0`.

Regarding RQ2 and RQ3, as a number of false alarms, we count the numbers of test executions that violate the assertions without covering the buggy statements. For true alarms, we count the number of test executions that cover the buggy statements and violate the assertions.

[1]Since array out-of-bound bugs do not necessarily crash a target program, we count the number of the assert violations instead of crashes.

[2]For `hmmer`, `sjeng`, and `libquantum`, we could not find details of crash bug-fix commits publicly available. Thus, we manually analyzed the difference of code between the released versions and identified buggy statements based on the changelogs.

5.1.5 Testbed Setting

For the CONCERT and CONCERT_0, we set the timeout of concolic testing (i.e., test generation) as 180 seconds per unit test driver. [3] Target functions can have different timeouts because the number of unit test drivers of each function can be different. For SUT and CONBOL, we set timeout of test generation of a target function as the same amount of time to the timeout of the function by CONCERT. For example, if `f` has four unit test drivers, the timeout of `f` for SUT and CONBOL are set as 720 seconds ($=4 \times 180$). We set the function correlation threshold τ as 0.7 since two random variables which have 0.7 or greater correlation coefficient are interpreted as highly correlated [21].

Since the experiment scale is large (i.e., targeting 15915 functions in the 15 target programs), the experiments were performed on 100 machines each of which is equipped with Intel quad-core i5 4670K (4.0GHz) and 8GB RAM, running Ubuntu 14.04.2 64bit version. We run four concolic unit tests on a machine in parallel to maximize utilization of CPU cores.

5.1.6 Threats to Validity

A threat to external validity is the representativeness of our target programs. We expect that this threat is limited since the target programs are widely used real-world ones and tested/analyzed by many other researchers. Another threat to external validity is the possible bias of the system tests we used to obtain correlation between functions. We tried to minimize this threat by utilizing all available system test cases in the benchmarks.

The primary threat to internal validity is possible faults in the implementation of the concolic unit testing techniques we studied. To address this threat, we extensively tested our implementation. A threat to construct validity is the use of the crash bugs that were already reported in the bug-fix commits. We target crash bugs that were confirmed by the developers because it would require too much effort to manually validate the alarms without confirmed reports in this large scale experiment (15915 target functions with tens of thousands of alarms). However, this threat is limited because all target programs are well-maintained real-world programs so that these programs may not have many unknown bugs.

5.2 Experiment Result

This section reports and analyzes the experiment results. [4] Section 5.2.1 describes the data obtained from the experiment which can help readers understand the answers to the research questions in Sections 5.2.2–5.2.4. For all comparison, we applied Wilcoxn test with a significance level 0.05 to show the statistical significance. All comparison results mentioned in this section are statistically significant unless mentioned otherwise.

5.2.1 Experiment Data

Statistics on Generated Unit Test Drivers/Stubs

Table 5.2 shows that CONCERT generated 5.2 unit test drivers for each target function on average over the 15 target programs (see the second column of the last row). Each unit test driver contains the

[3] We choose timeout as 180 seconds because exploratory studies with timeout as 60, 300, and 600 seconds suggested that increases beyond 180 seconds had negligible effects on the overall experiment results.

[4] The experiment setup and data are available at http://swtv.kaist.ac.kr/data/CONCERT_data.zip

Table 5.2: Average number of root functions per target function, the numbers of functions in a unit test driver, and a length of a call depth from a root function to a target function

Targets	Avg. # of root func. per target func.	Avg # of functions in a unit test driver	Avg. call depth from root to target
Bash-2.0	6.1	8.6	2.4
Flex-2.4.3	4.4	8.7	2.3
Grep-2.0	5.4	10.4	2.7
Gzip-1.0.7	5.1	9.4	3.2
Make-3.75	3.9	10.5	1.6
Sed-1.17	4.7	9.0	2.5
Vim-5.0	6.5	8.8	4.3
Perl-5.8.7	6.3	5.5	3.0
Bzip2-1.0.3	6.2	7.4	2.8
Gcc-3.2	3.8	7.0	2.4
Gobmk-3.3.14	5.3	8.4	3.5
Hmmer-2.0.42	5.7	8.4	3.8
Sjeng-11.2	4.2	7.7	2.1
Libquantum-0.2.4	5.7	9.7	2.5
H264ref-9.3	4.0	9.7	2.5
Average	5.2	8.6	2.8

source code of the 8.6 functions relevant with a target function on average (see the third column). Also, a target function is located at 2.8 call depth from the root function in a unit test driver on average (see the fourth column).

Time Cost, Generated Test Inputs, and Branch Coverage

Table 5.3 shows the testing time of the concolic unit testing techniques on 100 machines. SUT, CONBOL, CONCERT$_0$, and CONCERT spend 29.5, 29.2, 31.4 and 38.4 minutes per target program on average. CONCERT spend more time than the other techniques because the common likely-invariants heuristic takes additional time to monitor and analyze the test executions to infer the invariants.

Table 5.4 shows the number of test inputs generated by the concolic unit testing techniques. For example, CONCERT generates 360323.3 test inputs per target program (or 339.6 per target function) and SUT generates 1.7 million test inputs per target program or 1585.3 test inputs per target function on average. CONCERT generates much less number of test inputs than SUT because the size of the unit test driver of CONCERT is much larger than SUT (the unit test driver of CONCERT has 8.6 functions on average (see Table 5.2)) which makes each test execution slower than SUT on average. The number of the test inputs generated by CONBOL is similar to SUT and the number of CONCERT$_0$ is similar to CONCERT.

Table 5.5 shows the branch coverage of the concolic unit testing techniques. SUT, CONBOL, CONCERT$_0$, and CONCERT cover 58.0, 57.9, 60.9, and 58.9% of the branches of the target functions, respectively. [5] The reported branch coverages do not include the branches of the inserted assertions.

[5] CREST transforms a target program to an equivalent extended version whose branches contain only one atomic condition per branch. The branch coverage data in this paper is based on the extended target program.

Table 5.3: Time (in minutes) taken on the 100 machines to generate test inputs

Target programs	SUT	CONBOL	CONCERT$_0$	CONCERT
Bash-2.0	39.7	38.6	41.7	47.7
Flex-2.4.3	3.7	3.7	3.6	4.6
Grep-2.0	3.8	3.8	4.2	5.3
Gzip-1.0.7	2.0	1.9	2.7	3.3
Make-3.75	12.5	12.1	13.1	16.1
Sed-1.17	1.7	1.7	2.1	2.6
Vim-5.0	65.5	63.8	61.1	73.3
Perl-5.8.7	82.2	82.7	72.6	79.8
Bzip2-1.0.3	3.4	3.4	3.9	5.0
Gcc-3.2	120.3	118.5	138.0	179.4
Gobmk-3.3.14	75.4	74.2	87.0	104.4
Hmmer-2.0.42	15.8	16.4	18.9	25.3
Sjeng-11.2	3.5	3.4	3.8	5.0
Libquantum-0.2.4	2.7	2.7	3.2	3.9
H264ref-9.3	11.0	11.3	15.1	20.4
Average	29.5	29.2	31.4	38.4

Table 5.4: The number of test inputs generated by the concolic unit testing techniques

Target programs	SUT	CONBOL	CONCERT$_0$	CONCERT
Bash-2.0	2263040	2353562	405317	385557
Flex-2.4.3	470504	456389	126341	131444
Grep-2.0	281645	284462	82322	102903
Gzip-1.0.7	91481	90567	55216	64940
Make-3.75	1126144	1171190	383084	305102
Sed-1.17	99290	94326	60453	65710
Vim-5.0	4768589	4816275	1150637	927652
Perl-5.8.7	3097566	3159518	723619	755920
Bzip2-1.0.3	175496	184271	86971	77307
Gcc-3.2	8409823	8830315	1392020	1471351
Gobmk-3.3.14	2924200	3070410	730033	615971
Hmmer-2.0.42	418711	401963	217745	211870
Sjeng-11.2	185493	194768	68103	65360
Libquantum-0.2.4	117410	122107	69783	67647
H264ref-9.3	800384	768369	157636	156116
Average	1681985.1	1733232.8	380618.7	360323.3

Table 5.5: Branch coverage achieved(%)

Target programs	SUT	CONBOL	CONCERT$_0$	CONCERT
Bash-2.0	51.0	50.9	53.6	52.7
Flex-2.4.3	67.7	67.5	65.7	66.2
Grep-2.0	69.5	69.5	71.6	71.2
Gzip-1.0.7	58.5	58.6	64.1	60.7
Make-3.75	57.3	57.0	63.2	62.6
Sed-1.17	54.6	54.4	62.5	58.5
Vim-5.0	44.0	44.1	46.8	46.7
Perl-5.8.7	52.1	52.1	54.7	53.1
Bzip2-1.0.3	57.6	57.8	62.3	58.4
Gcc-3.2	56.1	56.0	62.2	59.0
Gobmk-3.3.14	57.9	57.6	59.3	59.2
Hmmer-2.0.42	62.9	63.0	60.8	60.3
Sjeng-11.2	59.5	59.3	60.8	56.3
Libquantum-0.2.4	66.5	66.3	67.3	64.4
H264ref-9.3	54.7	54.7	58.6	53.8
Average	58.0	57.9	60.9	58.9

5.2.2 RQ1: Bug Detection Ability

Table 5.6 shows the numbers of the target bugs and the detected bugs by the concolic unit testing techniques. The first column shows the target programs, the second column shows the number of the target bugs in the target programs, and the third to the last columns show the numbers of the detected bugs by the concolic unit testing techniques. For example, CONCERT detected 13 bugs among the 15 target bugs (bug detection ratio 86.7% (=13/15)) in `gcc-3.2` (see the last column of the 11th row of Table 5.6). For all 15 target programs, CONCERT detected 56 bugs among the 67 target bugs (bug detection ratio 83.6% (=56/67)).

CONCERT$_0$ shows the highest bug detection ability (i.e., 92.5% (=62/67) on the 15 target programs). The bug detection ability of CONCERT is less than CONCERT$_0$ because the false alarm reduction and alarm prioritization heuristics miss six (=62-56) target bugs. SUT also shows high bug detection ability (i.e., 91.0% (=61/67)) but at the cost of a huge number of false alarms (see Section 5.2.3). SUT$_R$ shows the worst bug detection ability (i.e., 45.1%(=30.2/67)). Finally, CONBOL detects only 48 bugs.

5.2.3 RQ2: Bug Detection Accuracy

Table 5.7 shows the number of false alarms and the false/true alarm ratios of the techniques. For example, CONCERT raises 348 false alarms and its false/true alarm ratio is 1.8 on `bash-2.0` (= 348/192 where a number of true alarms is 192) (see the last two columns of the third row).

Note that, for every target program, CONCERT raises the smallest number of false alarms and its false/true alarm ratio is the lowest too (CONCERT raises 432.1 false alarms and its false/true alarm ratio is 2.4 per program on average (see the last row of Table 5.7)). Thus, we can conclude

Table 5.6: The numbers of the detected target bugs by the concolic unit testing techniques

Target program	No. of the target bugs	SUT	SUT_R	CONBOL	CONCERT_0	CONCERT
Bash-2.0	6	5	1.7	4	5	4
Flex-2.4.3	2	2	1.6	1	1	1
Grep-2.0	5	3	1.8	4	4	3
Gzip-1.0.7	2	2	0.6	2	2	2
Make-3.75	3	3	1.9	2	3	3
Sed-1.17	2	2	0.7	2	2	2
Vim-5.0	6	5	3.4	3	5	4
Perl-5.8.7	6	6	2.3	4	6	6
Bzip2-1.0.3	2	2	1.7	2	2	2
Gcc-3.2	15	14	5.6	11	14	13
Gobmk-3.3.14	5	4	1.8	3	5	5
Hmmer-2.0.42	3	3	0.9	2	3	3
Sjeng-11.2	2	2	1.1	2	2	2
Libquantum-0.2.4	3	3	2.5	2	3	2
H264ref-9.3	5	5	2.6	4	5	4
Total	67	61	30.2	48	62	56

Table 5.7: The numbers of false alarms and false/true alarm ratios of the concolic unit testing techniques

Target programs	SUT		SUT_R		CONBOL		CONCERT_0		CONCERT	
	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio
Bash-2.0	58933	96.9	533.8	85.7	34771	87.8	2545	12.7	348	1.8
Flex-2.4.3	13520	53.2	173.7	52.7	8788	46.7	769	9.6	135	3.2
Grep-2.0	8093	32.4	208.0	29.6	4775	31.2	572	5.0	126	1.4
Gzip-1.0.7	2459	9.1	179.4	7.6	1328	6.9	313	2.8	126	1.6
Make-3.75	36094	82.0	408.0	82.0	19130	54.3	2677	18.1	303	2.8
Sed-1.17	3182	10.9	168.5	9.1	1655	8.7	354	4.2	130	2.3
Vim-5.0	116877	251.3	634.2	230.5	75971	204.2	7244	38.3	488	3.3
Perl-5.8.7	92189	135.0	1283.7	113.3	52548	108.3	5103	18.2	1011	3.6
Bzip2-1.0.3	4570	16.7	128.9	15.8	2971	14.1	576	6.1	90	1.9
Gcc-3.2	250292	145.3	3076.1	128.4	125146	121.0	8909	13.3	2506	4.2
Gobmk-3.3.14	97473	172.5	765.7	143.6	61408	175.0	4284	17.7	546	2.4
Hmmer-2.0.42	7321	20.3	155.0	19.3	4100	18.6	366	4.9	99	1.5
Sjeng-11.2	2021	10.2	130.8	9.2	1173	9.5	223	3.5	99	2.2
Libquantum-0.2.4	3156	15.4	122.7	14.8	2178	17.3	196	3.3	87	2.0
H264ref-9.3	20843	43.2	537.9	41.1	12715	39.4	1109	7.2	387	2.4
Average	47801.5	73.0	567.1	65.5	27243.8	62.9	2349.3	11.0	432.1	2.4

Table 5.8: Effects of the false alarm reduction and alarm prioritization heuristics

	$CONCERT_0$		$CONCERT_0$ +static time reduction		$CONCERT_0$ +static time reduction +common likely-invariants		CONCERT	
Target programs	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio
Bash-2.0	2545	12.7	2240	10.1	651	3.2	348	1.8
Flex-2.4.3	769	9.6	531	10.6	191	4.2	135	3.2
Grep-2.0	572	5.0	435	4.1	213	2.2	126	1.4
Gzip-1.0.7	313	2.8	210	2.3	147	1.8	126	1.6
Make-3.75	2677	18.1	2222	17.8	687	6.0	303	2.8
Sed-1.17	354	4.2	262	4.1	157	2.6	130	2.3
Vim-5.0	7244	38.3	4999	28.4	1210	7.5	488	3.3
Perl-5.8.7	5103	18.2	3368	10.8	1719	5.8	1011	3.6
Bzip2-1.0.3	576	6.1	386	7.1	162	3.2	90	1.9
Gcc-3.2	8909	13.3	5791	8.9	2901	4.7	2506	4.2
Gobmk-3.3.14	4284	17.7	3385	13.1	1058	4.4	546	2.4
Hmmer-2.0.42	366	4.9	323	4.7	155	2.3	99	1.5
Sjeng-11.2	223	3.5	179	3.4	111	2.3	99	2.2
Libquantum-0.2.4	196	3.3	151	3.1	107	2.3	87	2.0
H264ref-9.3	1109	7.2	932	5.4	464	2.8	387	2.4
Average	2349.3	11.0	1694.3 (-27.9%)	8.9 (-18.7%)	662.2 (-71.8%)	3.7 (-66.4%)	432.1 (-81.6%)	2.4 (-77.8%)

that CONCERT is far more accurate to detect the target bugs than the other concolic unit techniques. For example, CONCERT reduces the number of false alarm 99.1% (=(47801.5-432.1)/47801.5), 23.8% (=(567.1-432.1)/567.1), 98.4% (=(27243.8-432.1)/27243.8), and 31.2% (=(2349.3-432.1)/2349.3) compared to SUT, SUT_R, CONBOL, and $CONCERT_0$ respectively. In terms of the false/true alarm ratio, CONCERT decreases the ratio 96.7% (=(73.0-2.4)/73.0), 96.3% (=(65.5-2.4)/65.5), 96.1% (=(62.9-2.4)/62.9), and 78.2% (=(11.0-2.4)/11.0) compared to SUT, SUT_R, CONBOL, and $CONCERT_0$ respectively. Thus, we can conclude that CONCERT is more accurate to detect the target bugs than the other concolic unit testing techniques.

$CONCERT_0$ raises larger number of false alarms (i.e., 2349.3) and its false/true alarm ratio is higher than CONCERT since CONCERT applies false alarm reduction and alarm prioritization heuristics (see Section 4.3 and Section 5.2.4). SUT suffers a huge number of false alarms (i.e., 47801.5) and the highest false/true alarm ratio (i.e. 73.0). SUT_R raises a less number of alarms (i.e., 567.1) than SUT but its false/true alarm ratio is not much different from SUT. CONBOL also suffers a large number of false alarms (i.e., 27243.8) and high false/true alarm ratio (i.e., 62.9). Note that CONBOL's false alarm filtering and alarm scoring heuristics were optimized for the project S [36] of Samsung Electronics and the experiment result shows that those heuristics of CONBOL are not generally effective.

5.2.4 RQ3. Effects of the False Alarm Reduction and Alarm Prioritization Heuristics

Table 5.8 shows the effects of the false alarm reduction and alarm prioritization heuristics. The table shows that the false/true alarm ratio decreases as more heuristics are applied. For example, the static

Table 5.9: The effect of each false alarm reduction and alarm prioritization heuristic

Target	CONCERT w/o keeping consistency		CONCERT w/o value range analysis		CONCERT w/o common likely invariants		CONCERT w/o alarm prioritization		CONCERT	
Target programs	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio	# of false alarms	false/true alarm ratio
Bash-2.0	362	2.3	369	1.9	615	2.9	651	3.2	348	1.8
Flex-2.4.3	133	3.4	144	3.6	366	8.1	191	4.2	135	3.2
Grep-2.0	141	1.7	138	1.6	449	4.6	213	2.2	126	1.4
Gzip-1.0.7	134	1.9	135	1.8	264	3.3	147	1.8	126	1.6
Make-3.75	293	2.8	325	3.0	428	3.6	687	6.0	303	2.8
Sed-1.17	133	2.6	137	2.3	260	4.1	157	2.6	130	2.3
Vim-5.0	507	3.8	528	3.6	1186	7.4	1210	7.5	488	3.3
Perl-5.8.7	1202	4.8	1102	4.0	2763	9.1	1719	5.8	1011	3.6
Bzip2-1.0.3	91	2.2	99	2.1	171	3.3	162	3.2	90	1.9
Gcc-3.2	2580	4.7	2632	4.5	3238	5.1	2901	4.7	2506	4.2
Gobmk-3.3.14	563	2.7	574	2.6	887	3.7	1058	4.4	546	2.4
Hmmer-2.0.42	104	1.6	108	1.6	206	2.9	155	2.3	99	1.5
Sjeng-11.2	112	2.7	109	2.4	189	3.7	111	2.3	99	2.2
Libquantum-0.2.4	94	2.5	94	2.1	179	3.7	107	2.3	87	2.0
H264ref-9.3	422	2.8	426	2.5	720	4.0	464	2.8	387	2.4
Average	458.1	2.8	461.3	2.7	794.7	4.6	662.2	3.7	432.1	2.4

time reduction heuristics decreases the false/true alarm ratio of CONCERT$_0$ 18.7% (=(11.0-8.9)/11.0) on average (see the seventh column of the last row). When the static time reduction heuristics and the common likely-invariants heuristic are applied together to CONCERT$_0$, the false/true alarm ratio decreases 66.4% on average. Finally, when the alarm prioritization heuristic is applied in addition to all false alarm reduction heuristics (i.e., CONCERT), the false/true alarm ratio decreases 77.8% on average.

Regarding the bug detection ability, the static time reduction heuristics do not decrease the bug detection ability (i.e., CONCERT$_0$ + the static time reduction heuristics detects 62 target bugs). But the common likely invariant heuristic decreases the number of detected bug by three (i.e., CONCERT$_0$ + the static time reduction heuristics + the common likely invariants detects 59 target bugs). Finally, the alarm prioritization heuristics (i.e., CONCERT) decrease the number by three (i.e., 56 target bugs detected).

Table 5.9 shows the effect of each false alarm reduction and alarm prioritization heuristic in terms of false/true alarm ratio. CONCERT without keeping consistency and CONCERT without value range analysis have relatively 16.6%(=(2.8-2.4)/2.4) and 8.8%(=(2.7-2.4)/2.4) higher false/true alarm ratio than CONCERT on average respectively. CONCERT without common likely invariants and CONCERT without alarm prioritization have relatively 90.4%(=(4.6-2.4)/2.4) and 51.4%(=(3.7-2.4)/2.4) higher false/true alarm ratio than CONCERT. As shown in the table, the common likely-invariant heuristic is the most effective to reduce false alarms among the false alarm reduction and alarm prioritization heuristics. Regarding the bug detection ability, CONCERT without keeping consistency and CONCERT without value range analysis do not change the bug detection ability (i.e., they detect all 56 target bugs detected by CONCERT). But CONCERT without the common likely invariant and CONCERT without alarm prioritization detect two and three more bugs than CONCERT, respectively (i.e., the common likely-invariant and the alarm prioritization heuristics decrease bug detection ability 3.6% (=2/56) and 5.4% (=3/56), respectively). Thus, as shown in the experiment, each of the false alarm reduction heuristics and alarm prioritization heuristics is valuable for automated unit testing since each of them decreases the false/true alarm ratio much at the modest cost of missing bugs.

Chapter 6. Conclusion and Future Work

6.1 Conclusion

In this dissertation, I have demonstrated that an automated unit testing technique that synthesizes realistic unit context can automatically detects bugs in complex real-world programs with low false alarms. For this purpose, I developed an automated unit test framework CONBOL for large embedded software and showed that CONBOL had high bug detection ability 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. However, CONBOL suffered high false/true alarm ratio (i.e., 10.5) which decreases practicality of the technique.

To resolve this false alarm problem, I extended CONBOL to develop an automated unit testing technique CONCERT which generates unit test drivers/stubs which closely mimic the target program contexts with small amount of code to detect many bugs with *much less false alarms*. The salient idea of CONCERT is utilizing a dynamic function correlation metric to select and use the code of functions of a target program that are closely related to a target function under test in unit test drivers/stubs. Finally, through the experiments, CONCERT demonstrates both high bug detection ability (i.e., it detects 83.6% of all target crash bugs) and relatively low false/true alarm ratio (i.e., 2.4 false alarms per one true alarm). To the best of my knowledge, CONCERT is the most accurate automated unit testing framework for complex C programs in the world.

6.2 Future Work

6.2.1 Improving Function Correlation Metrics

I will utilize static analysis technique to overcome the limitations of the dynamic correlation metric on functions. For example, the proposed dynamic correlation metric is not applicable to functions which are not executed at all during system executions since the dynamic correlation is measured based on the function execution profile. In addition, if a target program has only a few system test cases, the dynamic correlation may be inaccurately measured since the observed function execution profile provide little information.

Static analysis such as def-use analysis can supplement dynamic correlation metric for such functions and refine the correlation information through static relationship between a target function and other functions. For example, we can apply a high weight to correlation between a target function and another function if they have high static def-use dependency or short call-graph distance.

6.2.2 Utilizing Unit Test Cases to Generate Effective System Test Cases

I plan to develop a technique to generate a bug triggering system-level tests based on unit tests. Note that the unit tests contains rich information which can be utilized to generate system tests and a system test can be considered as composition of unit tests. Basic idea is as follows. First, CONCERT generates a large number of accurate test cases for each unit and suppose that t_f is a generated test case for a target function f which violates an assertion in f. Then, we guide/control concolic testing or search-based exploration of a target program to reach a state which enforces t_f to f. Note that this task is easier than directly generate a system test that violates given assertions since we utilize information of a unit failing test (i.e., t_f).

6.2.3 Utilizing Dynamic Function Correlation Information for Various Purposes

I will apply dynamic function correlation information for various light-weight impact analysis [1] tasks since information of correlation among functions can have interesting applications in software development. For example, suppose that we recognize that a function f and a function g are closely correlated in a target program P. Then, such information can guide developers of P to group f and g together whey they update P. More specifically, if f is modified, g has to be tested together with f although g is not modified explicitly. Also, we can utilize function group information to identify/define reusable components and/or we can suggest refactoring guideline based on the function group information to achieve high cohesion and low coupling.

Bibliography

[1] R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] L. Baresi, P. Lanzi, and M. Miraz. TestFul: an evolutionary test approach for Java. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, 2005. Freenix Track.

[4] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, 2009.

[5] M. Buenen and G. Muthukrishnan. World quality report. Technical report, Capgemini, 2016.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 443–446, 2008.

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

[8] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[9] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2006.

[10] R. Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[11] S. Chidamber and C. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1991.

[12] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[14] Coverity. https://www.synopsys.com/software-integrity/products/static-code-analysis.html.

[15] H. Cramer. *Mathematical Methods of Statistics*. Princeton University Press, 1946.

[16] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software-Practice and Experience*, 34(11):1025–1050, 2004.

[17] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[18] S. Elbaum, H. Chin, M. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009.

[19] B. Elkarablieh, R. Godefroid, and M. Levin. Precise pointer reasoning for dynamic test generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[20] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[21] J. Evans. *Straightforward Statistics for the Behavioral Sciences.* Brooks/Cole Publishing Company, 1996.

[22] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[23] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

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

[25] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

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

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

[28] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *Proceedings of the NASA Formal Methods Symposium(NFM)*, 2009.

[29] H. Jaygarl, S. Kim, T. Xie, and Carl. Chang. OCAT: Object capture based automated testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[30] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[31] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)*, 2009.

[32] 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(3):355–374, 2012.

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

[34] Y. Kim, M. Kim, and N. Dang. Scalable distributed concolic testing: a case study on a flash storage platform. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2010.

[35] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012. SEIP track.

[36] 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 *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2013.

[37] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[38] M. Kucharski. Making unit testing practical for embedded development. http://electronicdesign.com/article/embedded/Making-Unit-Testing-Practical-for-Embedded-Development, 2011. Online; accessed 31 October 2016.

[39] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2008.

[40] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Proceedings of the Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC-PART)*, 2009.

[41] 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 (CGO)*, 2004.

[42] Y. Lee, B. Liang, S. Wu, and F. Wang. Measuring coupling and cohesion of object-oriented programs based on information flow. In *Proceedings of the International Conference on Software Quality (ICSQ)*, 1995.

[43] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993.

[44] R. Majumdar and R. Xu. Directed test generation with symbolic grammars. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2007.

[45] M. Marri, T. Xie, N. Tillmann, J.de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, 2009.

[46] P. McMinn. Search-based software test data generation: A survey. *Journal of Software Testing, Verification, and Reliability*, 14(2):105–156, 2004.

[47] L. Moura and N. Bjorner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[48] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, 2002.

[49] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

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

[51] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015. SEIP track.

[52] C. Pasareanu, P. Mehlitz, D. 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 *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[53] C. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[54] B. Pasternak, S. Tyszberowicz, and A. Yehudai. GenUTest: A unit test and mock aspect generation tool. *Software Tools for Technology Transfer*, 11(4):273–290, 2009.

[55] K. Pearson. Notes on regression and inheritance in the case of two parents. In *Proceedings of the Royal Society of London*, 1895.

[56] D. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium (SEC)*, 2015.

[57] R. Rodrigues, V. Campos, and F. Pereira. A fast and low-overhead technique to secure programs againtst integer overflows. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2013.

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

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

[60] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[61] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2010.

[62] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2006.

[63] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2005.

[64] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security (ICISS)*, 2008.

[65] The SPEC CPU 2006 benchmark suite. `https://www.spec.org/cpu2006/`.

[66] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), 2002.

[67] N. Tillmann and J. Halleux. Pex - white box test generation for .net. In *Proceedings of the International CConference on Tests And Proofs (TAP)*, 2008.

[68] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2005.

[69] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2000.

[70] B. Vlasic. Toyota's slow awakening to a deadly problem. *The New York Times*, 2 2010. `http://www.nytimes.com/2010/02/01/business/01toyota.html`.

[71] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of the European Dependable Computing Conference (EDCC)*, 2005.