Coursework 04
Detection of Software Vulnerabilities:
Static Analysis (Part II)

Introduction

This coursework introduces students to the basic concepts to verify rounding errors in single-threaded C programs and concurrency errors in multi-threaded C programs. In particular, this coursework provides three theoretical and practical exercises: *(i)* encode floating-point numbers using the theory of floating-point (QF_BVFP), *(ii)* apply the Lal / Reps sequentialization schema to multi-threaded C programs; and *(iii)* verify concurrency aspects of software embedded into industrial automation systems.

Learning Objectives

By the end of this lab you will be able to:

- Check rounding errors in single-threaded programs that contain floating-point numbers.
- Understand communication models and typical errors when writing concurrent programs.
- Implement sequentialization methods to convert concurrent programs into sequential ones.
- Implement the explicit schedule exploration of multi-threaded software.

1) **(Floating-point arithmetic)** Derive the equations C and P from the following program C. You must consider the floating-point theory implemented in the SMT solver (QF_BVFP) when writing the resulting formula C ∧ ¬ P [1].

```
int main() {
  double x = 0.1;
  double y = 0.2;
  double w = 0.3;
  double z = x + y;
  double a = x - y;
  double b = x * y;
  double c = x / y;
  assert(w == z);
  assert(a + b + c <= 1.0);
  return 0;
}
```

2) **(LR sequentialization)** Consider the following multi-threaded C program. Your task here is to write the sequentialized version of this program using the Lal / Reps sequentialization schema [2].

```
#include <pthread.h>
int g;
```

```
void *t1(void *arg) {
  int a1, *aptr1;
  aptr1=(int *)arg;
  a1=*aptr1;
  g=0;
  assert(a1 == 10 && g == 0);
  return 0;
}
void *t2(void *arg) {
  int a2, *aptr2;
  aptr2=(int *)arg;
  a2=*aptr2;
  g=0;
  assert(a2==20);
  return 0;
}
int main() {
  pthread_t id1, id2;
  int arg1=10, arg2=20;
  pthread_create(&id1, NULL, t1, &arg1);
  pthread_create(&id2, NULL, t2, &arg2);
  assert(g==0);
  pthread_join(id1, NULL);
  pthread_join(id2, NULL);
  return 0;
}
```

3) **(Lazy Exploration)** A given industrial automation system consists of monitoring and controlling an external environment using sensors, actuators, and other input/output interfaces. Usually, such a system can be implemented through a concurrent program, which consists of a collection of computational processes that run in parallel and that can interact with each other. For this particular case, consider that this automation system was implemented using three processes (i.e., P1, P2, and P3) and each process consists of two commands (e.g., a1 and b1 for process P1):

P_1:	**P_2:**	**P_3:**
a_1	a_2	a_3
b_1	b_2	b_3

Note that a concurrent program can produce different program executions (i.e., interleaving), depending on the scheduling algorithm employed by the underlying operating system. The number of interleavings is exponential in the number of processes and commands. Examples of possible interleavings include:
a1, b1, a2, b2, a3, b3;
a2, b2, a1, b1, a3, b3;
a1, a2, a3, b1, b2, b4.

Due to the sequential consistency criterion, you will not find an interleaving, where, for example, b1 runs before a1, and the writing occurs before reading. With this information, you must answer the following questions:

a) What are all possible interleavings of this system?
b) Implement a C / C ++ program based on the lazy exploration algorithm to explore all possible interleavings [3]. What is the complexity of your solution?
c) Consider that the actions of all processes consist of incrementing a global variable "x", which is initialized to zero before executing all processes. In this case, you would generate redundant interleaving (i.e., interleaving that always produces the same result). Implement a C / C ++ program to eliminate these redundant interleavings. What is the complexity of your solution?

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Question 1) Has the student written the SMT formulae to encode floating-point numbers using the theory of floating-point (QF_BVFP) via the C/C++ API of the SMT solver Z3?	
The student has written the SMT formulae to encode floating-pointer numbers using the C/C++ API of the SMT solver Z3.	(2)
The student demonstrates a limited capability to encode the SMT formulae to check for properties related to rounding errors. He/she does not correctly encode those properties using the C/C++ API of the SMT solver Z3.	(1)
No attempt has been made.	(0)

Question 2) Has the student sequentialized the multi-threaded C program using the Lal / Reps sequentialization schema?	
The student has sequentialized the multi-threaded C program using the Lal / Reps sequentialization schema.	(3)
The student demonstrates a limited capability to sequentialize the multi-threaded C program using the Lal / Reps sequentialization schema. He/she does not correctly encode the round-robin schedules, global memory copy, or checks for pruning the inconsistent simulations.	(2)
No attempt has been made.	(0)

Question 3) Has the student provided consistent answers about the lazy exploration of the multi-threaded program?	
The student has answered all three questions concerning lazy exploration. He/she describe all possible interleaving, the lazy exploration, and partial-order reduction algorithms.	(5)
The student has answered two questions concerning lazy exploration satisfactorily.	(3)
The student has answered one question concerning lazy exploration satisfactorily.	(1)
No attempt has been made.	(0)

References:

[1] SMT-LIB, http://smtlib.cs.uiowa.edu/index.shtml.

[2] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, Gennaro Parlato: Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. ASE 2015: 807-812.

[3] "Cordeiro, L. and Fischer, B. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In Intl. Conf. on Software Engineering (ICSE), pp. 331-340, IEEE/ACM, 2011."