Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2016

Testing Non-termination in Multi-threaded programs

Priyanka Thyagarajan
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Engineering Commons, Computer Sciences Commons, and the Literature in English, North America Commons

Recommended Citation
Thyagarajan, Priyanka, "Testing Non-termination in Multi-threaded programs" (2016). *Graduate Theses and Dissertations*. 15824.
https://lib.dr.iastate.edu/etd/15824

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات
www.manaraa.com

Testing non-termination in multi-threaded programs

by

Priyanka Thyagarajan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Samik Basu, Co-Major Professor

Gianfranco Ciardo, Co-Major Professor

Neil Zhenqiang Gong

Iowa State University

Ames, Iowa

2016

Copyright © Priyanka Thyagarajan, 2016. All rights reserved.

المنارة للاستشارات

www.manaraa.com

DEDICATION

I dedicate this thesis to my grandmother Mrs.Gnanasoundhari Sivaraj, who raised me to be the person I am today.

”You raise me up, so I can stand on mountains;

You raise me up to walk on stormy seas;

I am strong when I am on your shoulders;

You raise me up to more than I can be.”

المنارة للاستشارات

www.manaraa.com

TABLE OF CONTENTS

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

LIST OF TABLES

المنارة للاستشارات

www.manaraa.com

LIST OF FIGURES

المنارة للاستشارات

www.manaraa.com

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr.Samik Basu for his guidance, patience and support throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr.Gianfranco Ciardo and Dr.Neil Zhenqiang Gong. I would additionally like to thank Dr.Gianfranco Ciardo for his guidance during the initial stages of my graduate and research career. I extend my heartfelt gratitude to my family and friends for believing in me and being a constant source of motivation for all my endeavours. This research work was funded by grants CCF 0954132, CCF 1155780.

المنارة للاستشارات

www.manaraa.com

ABSTRACT

We study the problem of detecting non - termination in multi - threaded programs due to unwanted race conditions. We claim that the cause of non-termination can be attributed to the presence of at least two loops in two different threads, where the valuations of the loop controlling parameters are inter-dependent, i.e., value of one parameter in one thread depends on the execution sequence in the other thread and vice versa. In this thesis, we propose a testing based technique to analyze finite execution sequences and infer the likelihood of non-termination scenarios. Our technique is a light weight, flexible testing based approach that can be paired with any testing technique. We claim that testing based methods are likely to be scalable to large programs as opposed to static analysis methods. We present an outline of our implementation and prove the feasibility of our approach by presenting case studies on tailored sample programs. We conclude by discussing the limitations of our approach and future avenues of research along this line of work.

المنارة للاستشارات
www.manaraa.com

CHAPTER 1. INTRODUCTION

Concurrent programming is one of the cornerstones of computing in almost all real-world applications. However, it is challenging and in some cases, impossible to effectively analyze concurrent programs and prove that it satisfies the desired correctness requirements. The primary reason is, that execution paths in concurrent programs not only depend on the user inputs, but also on interleavings between threads, which in turn, is guided by the scheduler of the underlying computing environment. The complexity of the executions, therefore, makes automatic formal verification ineffective for even medium sized programs if not handled with caution. Further, this makes analysis of larger multi-threaded programs infeasible due to state-space explosion. The effectiveness of testing based techniques also remain less than desired because of issues of incompleteness. This is because, enumerating or analyzing all possible program execution paths involves exhaustive exploration of both the input space and the thread context switches space. Hence, many valid and buggy execution paths may remain unexplored.

Termination of a program is one of the primary concerns of a programmer. A program with a non-terminating execution sequence could cause violation of a system's safety and correctness requirements, thereby affecting its stability. Depending on the context of non-termination, it could also lead to resource unavailability or starvation (Cook et al. (2007), Rodrigues (2013)). From a security standpoint, an attacker could exploit weakly designed loops by strategically devising inputs that would cause the program to be trapped in a non-terminating execution sequence (Rodrigues (2013)). In concurrent event-driven programming, consider device drivers that provide event-handling services of independent threads, while communicating through shared memory. These device drivers are permitted by operating system to temporarily take over the execution of the threads in which the event occurred. A scenario could occur, where a loop in the code executed by the device driver could diverge when a relevant shared variable is

www.manaraa.com

modified by other threads in the same driver. Such scenarios could potentially cause denial of service, rendering the entire system unavailable. Hence, we see that non-terminating execution scenarios can greatly compromise the underlying computing environment's reliability.

The termination problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever. It is one of the most studied problems in computer science and in general, is undecidable even for sequential programs. In concurrent programs, the non-deterministic behavior introduced by interleavings between threads, memory model employed and compiler optimizations, increase the complexity of this problem further. A programmmer has little control over these concerns since they are typically handled by the compiling environment.

We focus on the following question, given a concurrent program, is it possible to determine if there is a likelihood of a non-terminating execution sequence? Specifically, this contribution primarily focuses on determining possible unbounded execution paths arising due to unexpected or poor context switches (thread schedules).

Techniques for program verification can be broadly classified into: Formal verification based techniques like model-checking/theorem proving and testing. Formal verification/theorem proving based solutions prove to be unsuitable approaches for our requirement, due to scalability issues and need for possible human intervention. When we proceed to examine testing, we observe: Testing based methods are adept at verifying properties that can be expressed as assertions. Testing is a technique that requires a program to terminate. Traditionally, non-termination is a property that cannot be expressed as assertions.

We now ask the question, is it possible to use a testing based approach to determine if a concurrent program terminates? Our primary contribution addresses this question, and presents a method using testing, that can infer a possible unbounded execution path (non-termination) in a concurrent program, by testing several carefully chosen finite-path assertional properties. We have developed a generic, programming language agnositc methodology to detect the likelihood of non-termination in multi-threaded programs. We are also interested in constructing a modular framework, that leverages the performance of our methodology by allowing it to be paired with any testing technique.

المنارة للاستشارات

www.manaraa.com

We employ testing as opposed to static analysis for three main reasons. Testing based techniques are typically more scalable than static analysis methods; bugs obtained via testing can be reproduced using the test cases; and testing does not require any special adjustments for memory model as long as the test criteria is well-defined.

1.1 Overview of existing approaches for determining non-termination

Most research efforts have been directed in developing semi-automatic or sound, but incomplete methods, and heuristics to verify, and check for termination. In sequential programs, one approach has been to ensure that any execution sequence involving cycles, (loops or recursion) moves the program execution towards a base case for the cycle. In this respect, Cook et al. (2005), Podelski and Rybalchenko (2004a) prove termination by using principles of ranking functions and well-foundedness of the program's transition relation. Another approach has been to prove non-termination in sequential programs by ensuring the presence of set a of program states with specific properties that imply non-termination-closed recurrence sets. Along these lines, while Gupta et al. (2008), utilizes concolic testing to illustrate non-terminating cases, Chen et al. (2014) utilize safety proving paired with counter-example guided abstraction refinement to detect violation of termination.

The problem of detecting non-termination becomes more challenging when concurrency is involved. This is because of the presence of interleavings and interferences between threads, that can impact the entities in the program, and therefore, its temination property. In Cook et al. (2007), the authors apply static analysis to develop sound and incomplete techniques proving that each thread in a program will eventually terminate. Similarly, in Popeea and Rybalchenko (2012), the authors apply rely-guarantee compositional techniques to determine termination of each thread based on the properties of its environment (other threads). The authors of Morse et al. (2011) and Musuvathi and Qadeer (2008) apply model checking techniques to verify liveness properties by constraining the context switches or considering fair scheduling. Atig et al. (2012b) apply model-checking along with the principle of scope-bounding to verify liveness properties and liveness violation. Perhaps the closest to our approach is the technique presented in Atig et al. (2012a). This technique involves converting a multi-threaded program

www.manaraa.com

```
while $x \leq 5$ do
    Do-something
    $x++$
    $y--$
end while
```

(a) Thread 1

```
while $y \leq 5$ do
    Do-something
    $y++$
    $z--$
end while
```

(b) Thread 2

```
while $z \leq 5$ do
    Do-something
    $z++$
    $x--$
end while
```

(c) Thread 3

Figure 1.1 Example illustrating inter-thread loop dependency

into its sequential counterpart and inserting carefully selected assertions, violations of which proves the presence of non-termination under fair scheduling.

1.2 Proposed Solution

We focus on non-terminations, that are caused due to interferences between threads. That is, we assume the threads do terminate if allowed to execute in isolation. Therefore, the non-termination is manifested in the execution of certain specific program fragments in an unbounded fashion, i.e., some loop gets executed unboundedly. Furthermore, as the cause of non-termination is due to interference, there must be at least two threads, whose execution remains trapped in loops, and the execution of one thread continues to interfere with the execution of the other. To detect the possibility of such scenarios leading to the non-termination, we propose a technique, that analyzes the result of testing certain carefully selected assertions.

Consider the multi-threaded program shown in Figure.1.1. The program consists of three threads, where each thread has a loop conditional over a shared variable, which is modified in a different thread. For example, Thread 1 has a loop conditional over the shared variable x and is modified by Thread 3. It therefore becomes apparent, that there is a possible unbounded execution. This execution sequence can occur if for each thread, before the loop condition is checked, the other thread modifies the loop controlling parameter.

We start by determining the number of times each loop in each thread is executed, we call this value *base count*. This can be done by instrumenting each loop in each thread, with a count variable (say cnt_i in thread i, for all $i \in [1..3]$), such that, it is incremented each time the loop unfolds and executing each thread individually. In our specific example, we note that

www.manaraa.com

the base count will be 5 for each loop if x, y, z are initialized to 0. Our next step involves placing assertions of the form assert($cnt_i \leq$ base count) in each of these threads. We use testing to check for violations of these assertions. In case of an assertion violation, we examine the program trace leading to the assertion violation, for count variable(s) pertaining to different thread(s).

In our example, after determining the base count for each loop in each thread, we test the program and find an assertion violation on cnt_1 in Thread 1. We analyze the program trace and find the appearance of cnt_3 corresponding to the Thread 3. Similarly, testing the example program with assertion on cnt_2 yields an assertion violation with cnt_1 in the program trace, testing the example program with assertion on cnt_3 yields an assertion violation with cnt_2 in the program trace. We make the following important observations:

- The number of times each loop in each thread unfolds depends on the the interferences.
- There exists a circular or mutual interdependency between each loop in each thread.

We can infer that these threads can continue to interleave in this fashion leading to an unbounded execution of the program. Hence, we use testing based techniques to infer the likelihood of an infinite execution sequence by verifying assertional properties on finite execution paths.

1.3 Contributions

The following are the contributions of this thesis:

- ***Methodology***: We propose a testing based method, to determine whether a multi-threaded program terminates or not. We are specifically interested in identifying non-terminating execution sequences due to unexpected or bad thread schedules. Our method is based on inferring the existence of unbounded execution path(s), by violations of assertions on finite execution paths. We examine the program trace to establish dependencies between threads. We deduce the presence of non-terminating execution path(s) by checking for circular or transitive dependencies between loop conditionals on shared variables in different threads.

www.manaraa.com

- ***Dependency Graph***: We also propose the *dependency graph*, an effective data structure to represent the dependencies between threads. We present a reduction from detecting non-termination in multi-threaded programs to cycle detection in this *dependency graph*.
- ***Tool implementation***: We have implemented a framework for detecting non-terminating behavior in concurrent programs. To the best of our knowledge, this is the first testing based approach to detect the likelihood of non-termination. We verify the correctness of our approach by testing our tool on systematically enumerated sample programs. Preliminary results reveal the feasibility of our approach.
- ***Identification of countermeasures***: In the event of identification of non-terminating behavior, our *dependency graph* aids in discerning the specific threads and loops that caused the unbounded execution. Hence, our tool aids in devising possible remedies to correct the unsafe program behavior without compromising the advantages of concurrency.

1.4 Outline

The thesis is organized into six chapters. In Chapter 1, we provided an introduction to our work, our problem statement and a brief description of our methodology. In Chapter 2, we present a discussion of the existing work that attempt to solve problems similar to our line of research. We delineate our methodology and contributions. In Chapter 3, we present our algorithm for determining non-termination. Next, we describe our data structure for effectively storing the inter thread dependencies and present our algorithm for cycle detection. We proceed to outline the architecture of our tool and details of implementation. In Chapter 4, we present our case-studies on sample programs to illlustrate the feasibility of our approach. In Chapter 5, we summarize inferences from our experiments, limitations of our approach and discuss possible directions for extension of our work.

المنارة للاستشارات
www.manaraa.com

CHAPTER 2. REVIEW OF LITERATURE

Discovering programming errors and bugs in code, is a field of research, that has been well pursued. Techniques for analyzing a program for discovering bugs, fall into either one of the following categories: static analysis or dynamic analysis based techniques. Formal methods based techniques like model-checking, theorem - proving, satisfiability modulo theories fall into the former category. While, dynamic analysis involves evaluation and testing of a program by executing it in real-time.

Program verification in the concurrent domain comes with its own complexities. Sequential programs are verified or tested by examining their program behaviors over the input space. However, a typical execution sequence in a concurrent program involves multiple threads that share memory address space and interleave with each other. These interleavings could cause unwanted data race conditions in the program. In addition to the inputs, each thread schedule also causes the program to take a distinct execution path. Further, the order of these interleavings is decided by external factors like processor utilization and I/O activity.

One of the challenging aspects of verifying the correctness requirements of a concurrent program, involves extremely exhaustive exploration of all possible distinct program behaviors, that are characterized by both inputs and thread schedules. Typical methods to treat this state-space explosion include, sequentialization of the concurrent program (Bouajjani et al. (2011), Inverso et al. (2014)), fair scheduling (Emmi et al. (2011)) and context-bounding (Qadeer and Rehof (2005)). In this work, we focus on the liveness violation or live-lock property: Does a multi-threaded program terminate? Most of the research in this area, involves verifying or proving or disproving this property in sequential programs. However, it is a relatively less trodden path in the domain of concurrent programs.

www.manaraa.com

2.1 Non-termination in sequential programs

Safety properties can be verified by the violation of assertions and require a finite path counter-example. However, non-termination modeled as a liveness violation/ livelock requires the existence of an unbounded execution path. In general, deciding if a program terminates is an undecidable problem. Therefore, the approaches for verifying if a program terminates have been directed towards, studying the structure of the program and how its entities are manipulated and develop sound/semi-automatic, but incomplete methods and heuristics.
This line of research involves constructing non-termination proofs by invariant generation or looking for the occurrence of recurring program states. Velroyen and Rümmer (2008) prove non-termination by showing that there exists a set of input values such that program does not terminate. Termination proofs are generated by invariant generation followed by theorem proving. The theorem prover attempts to construct a proof for non-termination from the invariants generated by the invariant generator. If unsuccessful, the incomplete proofs are used to refine the invariants. Several heuristics are used for invariant scoring, invariant filtering that help in weeding out irrelevant invariants and prioritizing the more useful ones. Larraz et al. (2014) uses MAX-SMT based invariant generation to prove non-termination. The authors analyze the program's control flow graph and for each strongly connected sub-graph, they use MAX-SMT techniques to discover a formula with very specific properties at each node. This formula should satisfy two properties. The first property being, *quasi-invariance* meaning, if the formula holds for one execution sequence it should continue to hold thereafter. Secondly, the formula should be *edge closing* meaning, it forbids the execution of any outgoing transition that would leave the strongly connected sub-graph.
Gupta et al. (2008) (TNT) utilizes concolic testing (Sen and Agha (2006a)) to generate candidate *lassos*. Here, a *lasso* represents a finite program path called *stem* followed by another finite program path called *loop*. The *loop* is a syntactic cycle in the control flow graph. The control flow graph of the *lassos* are then analyzed for finding *recurrent sets* to prove non-termination, through template based constraint solving. The authors define a recurrent set to be a set G of program states, such that, the following conditions are met:

www.manaraa.com

- G is non empty, atleast one state s in G is reachable.
- Every state $s \in G$ has a transition.
- Every transition in G remains within G.

Concolic testing is a systematic testing method that involves the iterative, simultaneous symbolic and concrete execution of the program. The constraints generated from the symbolic execution are used to generate concrete inputs for the next iteration of concrete execution. This way, the program now explores a previously unexplored path. The generation of lassos continues until all lassos are extracted or the algorithm concluded with non-termination.

```
assume(true);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(a)

```
assume(k ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(b)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(c)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}

assert(false);

i := 2;
```
(d)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

assert(false);

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}
```
(e)

```
assume(k ≥ 0 ∧ i ≥ 0);
assume(k ≥ 0);

skip;
while (i ≥ 0) {
    i := nondet();
    assume(i ≥ 0);
}
```
(f)

Figure 2.1 Sample program and successive approximations for proving non-termination through closed recurrence sets

Chen et al. (2014) encodes non-terminaton as a safety property by using *closed reccurence sets*(a stronger form of recurrence sets) proposed by Gupta et al. (2008). The authors use a safety prover to verify this property over an under-approximation of original program's control flow graph. If this property holds, the counter-example serves to be a witness to non-

المنارة للاستشارات
www.manaraa.com

termination. In the event of violation of the safety property, the counter example is used to refine the under-approximation, so that, the path leading to the violation of the property is weeded out. In this manner, the authors first determine if a loop can have an unbounded execution path, followed by which, they use a realizability checker to determine if the loop is reachable (i.e. finding the lasso followed by stem).

Consider, the example in Figure.2.1(a) a program with a possible non-terminating execution sequence for appropriate input values and non-deterministic choices. The algorithm introduce under-approximation by inserting $assume(true)$ at the beginning of the program and after every non-deterministic input assignment. $assert(false)$ is inserted at the exit of the loop, to encode *never terminates* condition. The safety prover provides a counter-example of the form $k < 0 \wedge i < 0$, this is used to refine the under approximation, the initial $assume(true)$ is modified to $assume(k \geq 0)$ as shown in Figure.2.1(b). This ensures the counter-example that lead to violation of *never terminate* is removed. For the next iteration, the safety prover provides the counter-example, $k \geq 0 \wedge i < 0$. Hence $assert(false)$ was reachable. The under-approximation is now refined by including $i \geq 0$ in the assume statement as shown in Figure.2.1(c). In the next iteration of safey proving, the counter-example presented is $k \geq 0 \wedge i \geq 0$ followed by $i < 0$ during the non-deterministic choice. To handle this path, the condition $assume(true)$ after the non-deterministc choice is changed to be $assume(i \geq 0)$ as shown in Figure.2.1(d). Now, there exists no execution path that could lead to $assert(false)$. The algorithm has successfully found an underapproximation of the program that never terminates.

The next step is to prove the existence of a *stem*, in other words, that the loop is reachable form the initial state. This is done by placing an $assert(false)$, before the loop as shown in Figure.2.1(e) and running the safety prover. A counter-example provided by the safety prover proves that the loop is reachable. The authors, then proceed to check the satisfiability of the generated underapproximation constraints in Figure.2.1(f) to verify the soundness of the under-approximation. Hence, this approach explores the possibility of proving or analyzing non-termination by safety checking.

Cook et al. (2014) propose utilizing a combination of over-approximation and under-approximation combined with constraint solving for generating closed recurrence sets. The advantage of this

www.manaraa.com

approach is the ability to efficiently handle non-linear, non-deterministic and heap based statements by abstractions or over-approximations.

Computing ranking functions/proving the well-foundedness of the transition relation of a program, are classical techniques for constructing termination arguments. These notions are based on proving that for every transition, the program execution converges towards the exit condition of a loop or recursive procedure. Consider a program trace $\pi = s_0 \rightarrow s_1 \rightarrow s_2....$, where s_i represent individual program states and $\rightarrow$ represents a single step transition. To prove termination naively, we need to prove that there exists no such infinite sequence. The $ranking function$ method, involves finding a function ρ that maps the program states to a well-founded, ordered set W, such that $\rho(s') < \rho(s)$ for all $s \rightarrow s'$. Cook et al. (2006) discuss a method involving generation of ranking functions aided by counter example guided abstraction refinement based on Podelski and Rybalchenko (2004b). More specifically, this involves showing for the transitive closure over the transition relation restricted to reachable states($R_I{}^+$), there exists a decreasing ranking function. More formally, this is proved by showing that there exists a disjunction of well-founded relations T such that $R_I{}^+ \subseteq T$. Given a program P, it is transformed into P', such that an error condition is not reachable only if $R_I{}^+ \subseteq T$ holds. Hence, a safety checker can be used to verify this. A counter-example will be converted into an input for a constraint solver based rank function systhesizer. The rank function generator outputs a well-founded rank relation W, that is used to refine the termination argument T (i.e. $T = T \cup W$). The notion of finding ranking function to construct termination argument for programs has been further explored by Cook et al. (2005), Podelski and Rybalchenko (2004a), Cook et al. (2010), Cook et al. (2013).

2.2 Non-termination in concurrent programs

We begin, by presenting a discussion on model-checking based approaches for detecting liveness violation, since non-termination itself can be viewed as liveness violation or live-lock. Morse et al. (2011) explores a context bounded model checking approach for checking LTL-liveness properties in multi-threaded programs. State hashing is used to prune the redundant interleavings. A B$\ddot{u}$chi automata of the negated LTL property is transformed into a monitor thread.

www.manaraa.com

```
$lock(lck)$
**while** $(x > 0)$ **do**
    atomic$(x--)$
**end while**
$unlock(lck)$
```

(a) Thread 1

```
**while** (nondet) **do**
    atomic$(x--)$
**end while**
```

(b) Thread 2

```
**while** (nondet) **do**
    $y = y + 1$
    $lock(lck)$
    atomic$(x--)$
    $unlock(lck)$
**end while**
```

(c) Thread 3

Figure 2.2 An example program for proving thread termination

The program to be tested is instrumented with this monitor thread, the instrumented program is checked using a SMT based model-checker. Along these lines, Musuvathi and Qadeer (2008) developed another context bounded model checker (CHESS) for verifying liveness properties, by employing an explicit fair-scheduler, that considers only a subset of the interleaving space. *fair schedules* may be considered as, schedules in which, if a thread is scheduled infinitely often, it is also enabled for execution infinitely often. Thus, there is no thread starvaton. Atig et al. (2012b) describes the notion of scope-bounding as opposed to context-bounding for verification of liveness properties in multi-threaded programs. scope - bounding states that, between a call and return of any procedure in any thread, there should be a finite number of context switches or interleavings. In other words, if a thread executes a procedure, it should empty the call stack after atmost k context-switches. This does not place an overall limit on the number of context-switches for the program, hence a thread could execute forever. Thus, this work could be potentially used for verifying termination of a program.

Cook et al. (2007) extends ranking function based termination proofs for sequential programs to multi-threaded programs, by generating environmental abstractions to approximate the behavior of surrounding threads. Here, as opposed to proving the overall termination of the program, the authors try to prove individual thread termination. The method returns a set of conditions that a thread requires of its environment to terminate. This set of conditions is called A or the agreement, it is in the conjunctive normal form and it is iteratively constructed. Consider the example in Figure.2.2. To prove that Thread 1 terminates, the algorithm begins by constructing a naive agreement $A_1 = true$, meaning no restrictions are placed on the way shared variables are modified Thread 2, Thread 3. The algorithm now attempts to come up

المنارة للاستشارات
www.manaraa.com

with termination proofs for the non-deterministic alternate execution of the Thread 1 and restrictions imposed by A_1. In this case, it is trurly non-deterministic. The A_1 is not strong enough to prove termination, the termination check comes up with the counter example corresponding to $x = x - 1$, $x = \mathit{nondet}$, $\mathit{assume}(x > 0)$. Based on the counter example generated, the A_1 is refined to $A_2 = \mathit{true} \wedge (x' \leq x)$(primed variables correspond to variable valuation following a transition). For the Thread 1 , A_2, the termination checker fails to provide a counter example, a successful approximation has been derived.
Next step is to analyze if threads, Thread 2, Thread 3 modify shared variable x as mandated by A_2. For Thread 2, this is true, since an atomic($x--$) can only decrement the value of x. For Thread 3, this is false, since there exists a non-deterministic update to x. Hence a new agreement A_3 is generated. $A_3 = \mathit{true} \wedge (x' \leq x) \vee (\mathit{lck} \neq 1)$. This states that the Thread 3 can make non-deterministic updates to x on when lck is not acquired by Thread 1. The newly generated agreement A_3 and Thread 1 are subjected to termination checking. No counter example is generated. Hence the termination proof of Thread 1 is given by $\mathit{true} \wedge (x' \leq x) \vee (\mathit{lck} \neq 1)$. In this way, Cook et al. (2007) prove *thread modular termination.*

Another approach along this line is Popeea and Rybalchenko (2012)'s method which involves constructing compositional termination arguments for multi-threaded programs using rely-guarantee reasoning. Rely guarantee reasoning first developed in Jones (1981), allows thread-modular reasoning by placing certain assertions or restrictions on the behavior of the other threads. More specifically, these assertions restrict the way environmental threads modify global variables. This method relies on the concepts of proving that the transitive closure of a thread's transition relation is well-founded for thread-modular termination. This is done by candidate ranking function synthesis as described in Podelski and Rybalchenko (2004a). Environment transitions keep track of the effect of the other threads on the thread of interest. In Cook et al. (2007), only a restricted class of environment transitions were taken into account, since only global variables were considered. Popeea and Rybalchenko (2012) define the environment transitions to keep track of both global variables and local variables pertaining to the other threads. This proof rule is automatically constructed by a transition predicate abstraction and refinement procedure, that involves solving horn clauses.

www.manaraa.com

Atig et al. (2012a) reduces the problem of detecting non-termination in multi-threaded programs to a reachability problem in sequential programs. This approach (MUTANT) detects non-termination in multi-threaded programs, by violations of carefully selected assertions in their sequential counterparts under fair scheduling. Based on the context bound, an instrumented sequential program is generated, this annotated sequential program is fed to an Satisfiability Modulo Theory based bounded model-checker. The authors characterize any non-terminating execution sequence in a program to consist of a stem and repeated executions of a lasso. Stem represents a finite set of executions, lasso represents the repeated execution of the same set of actions over and over again. The authors place a context bound $K \in \mathbb{N}$, where $K = k_1 + k_2$, k_1, k_2 are context bounds of stem and lasso respectively. The general idea is to show that any non-termination in a multi-threaded program can be decomposed into a stem and lasso. For each period of lasso, each thread encounters the same sequence of global state evaluations at the context-switch points(when the execution of a thread is interleaved by another thread). During each lasso period, each thread also re-encounters the same sequence of topmost stack-frame evaluation, since the stack should be non-decreasing in the lasso period for a non-terminating execution sequence to occur. The authors infer the existence of a non-terminating execution sequence by analyzing the global state evaluations, topmost stack frame evaluations during a single lasso period, for each thread $t \in Tid$. The authors handle state space explosion by considering fair schedules, in which every thread that is scheduled infinitely often, is also enabled infinitely often. Hence, the method reports only fair non-terminations in concurrent programs. This method is perhaps, the closest to our line of research.

2.3 Testing multi-threaded programs

In this thesis, we present a novel, light-weight, testing based methodology for detecting non-termination in multi-threaded programs. While we pair random testing with our algorithm to validate the efficiency of our approach, our modular framework can be used in conjunction with any testing technique. In concurrent programs, it is imperative to note, that in addition to user input, the thread schedule or order of context switching also causes a distinct program behavior. Recall, that concurrent programs have inherent non-deterministic behavior. Specifi-

www.manaraa.com

```
dosomething (int $x$, int $y$)
$z = 2y$
**if** $(z = x)$ **then**
    **if** $(x < y + 10)$ **then**
        error
    **end if**
**end if**
```

Figure 2.3   A program for illustrating concolic testing

cally, the non-determinism in the schedule order relies on external factors like memory models, complier optimizations and so forth. We now present a discussion on certain systematic testing techniques that exhaustively explore both the input and interleaving space.

Farzan et al. (2013) and Sen and Agha (2006b) describe adaptations of concolic testing technique to multi-threaded programs. concolic testing (CUTE) originally developed by Sen and Agha (2006a) uses concrete execution combined with symbolic execution to systematically explore all possible program execution paths. The authors begin, by executing the program concretely with inputs generated by a random input generator. The symbolic execution follows concrete execution, at the end of the execution, symbolic constraints that represent the current path of execution are extracted. The symbolic path constraint is of the form of a conjunction of linear inequalities or predicates, that represent individual branch constraints. The predicate corresponding to the program's last branch executed, is negated and sent to a constraint solver. The modified symbolic path constraint corresponds to a set of conditions, that should be satisfied for the program to take a previously unexplored path. The constraint solver returns a set of inputs that satisfy the modified symbolic path constraint. The generated test inputs are then passed for the next round of concrete execution. This is done in a loop till there exists no more paths to be processed. This way the input search space is systematically explored in a manner that could possibly prevent the testing tool from re-exploring previously explored program path.

Consider, the example in Figure.2.3, the function *dosomething* has two input arguments x and y. For the error condition to be reached, the program should take an execution path, that enters both the conditional statements. Concolic testing begins, by performing concrete

المنارة للاستشارات
www.manaraa.com

execution for some arbitrary input values ($x = 20, y = 7$), following the corresponding symbolic execution ($x = x_0$, $y = y_0$, $z = 2y_0$), the symbolic path constraints ($2y_0 \neq x_0$) are extracted. The path constraint contains a single branch constraint negating the constraint ($2y_0 \neq x_0$), solving for y_0, x_0, we get $x_0 = 2, y_0 = 1$. The next round of concrete execution is performed with the newly generated inputs. The program now follows a new execution path, $z = 2$, $z = x$, hence execution enters the first conditional statement, however, the second conditional statement fails. The symbolic path constraint extracted is $(z_0 = x_0) \wedge (x_0 \geq y + 10)$. Negating the last branching constraint, $(z_0 = x_0) \wedge (x_0 < y + 10)$, applying constraint solving techniques, we get $x_0 = 8$, $y_0 = 4$. Applying these newly generated inputs, the program follows $z = 8$, $z = x$, $x < y + 10$ $(8 < 14)$. Hence, the error state is reached. In this fashion, concolic testing, systematically generates test inputs to discover bugs.

Sen and Agha (2006b) present an extension of concolic testing (JCUTE) to multi-threaded programs to methodically explore the input and interleaving space. The authors propose a method, where they begin testing with concrete execution, symbolic execution extracts symbolic path constraints. Unlike symbolic path constraints in sequential programs, these constraints represent the path currently explored as a function of both the inputs and the interleaving. In the event, that the modified symbolic path constraint is not solvable, a race flipping technique is employed. The race flipping technique involves the following, when two threads are involved in a race condition, a new schedule is generated such that, one of the threads involved in the race condition is delayed as much as possible. These steps are performed in a loop till all possible program execution paths are processed. It is important to note, that this technique explores the interleaving space only when a previously unexplored program path could not be explored by input exploration.

Farzan et al. (2013) present a more complete testing technique (CONCREST) that performs the exhaustive exploration of all possible program execution paths bounded by k interferences. Consider a program with two threads, a shared variable x, if a read operation on x in Thread 1, is preceded by a write operation on x in Thread 2, we say, Thread 2 interferes with Thread 1. Here, shared read and shared write refer to read and write operation performed by a thread on a shared global variable. The authors propose a method which uses concolic testing com-

المنارة للاستشارات
www.manaraa.com

```
1: $y = 0$
2: **if** $(y > 0)$ **then**
3:     error
4: **end if**
```

(a) Thread 1

```
1: $x = input()$
```

(b) Thread 2

```
1: $x = 0$
2: **if** $(x = 2)$ **then**
3:     $y = 1$
4: **end if**
```

(c) Thread 3

Figure 2.4 A program for illustrating con2colic testing

bined with an *interference scenario* generator. An interference scenario corresponds to a new path of execution obtained by negating branch conditions in the program trace obtained from concolic execution. Each interference scenario is added to a central forest structure called the *interference forest*. For each interference scenario generated, a *realizability checker* examines, if there exist a set of inputs and a thread schedule that would lead to the scenario. If such a set of inputs and a thread schedule exists, the scenario is said to be *realizable*. These set of inputs, thread schedule are used for the next iteration of concolic execution. Every unrealizable scenario is pushed into a list to be processed in the forthcoming iterations. The algorithm begins by executing each thread individually, then proceeds to do the following in a loop till all interference scenarios are explored or the context bound is reached: concolic execution, interference scenario generation and realizability checking. For each iteration in the loop, the algorithm tries to associate a previously unrealisable scenario with appropriate shared writes from the *interference forest*, that might make it realizable.

Consider the example in Figure.2.4 consisting of three threads. x and y are global variables. The goal here is to reach the error condition in the Thread 1. Con2colic testing detects the bug in the following manner:

- Concolic testing on Thread 1, gives a program trace showing, $write(y, 0)$, $read(y)$, $(y \leq 0)$. The interference scenario generator now produces the scenario $write(y, 0)$, $read(y)$, $y > 0$. Now, this path cannot be reached unless, before the $read(y)$ is performed, y is modified to a value greater than 0 by a different thread. In other words, there exist no thread-local way of reaching this specific program location. Hence, the realizability checker returns false.

www.manaraa.com

- Concolic testing on Thread 2, returns a program trace $write(x, userinput)$. There exists no more unexplored paths in Thread 2. The unrealizable scenario in Thread 1 has no matching shared write in the central forest structure. Hence, no action is taken.

- Concolic testing on Thread 3, returns a program trace $write(x, 0)$, $read(x)$, $x \neq 2$. The interference scenario generator returns a scenario : $write(x, 0)$, $read(x)$, $x = 2$. The realizability checker renders this to be not thread-locally realizable. However, there exists a shared write to x in Thread 2 in the central forest data structure. Hence, an interference is introduced such that after, $write(x, 0)$, an interference from Thread 2 shared write on x occurs, before the $read(x)$. The inputs and thread schedule corresponding to this is generated for the next iteration of concolic testing. The unrealizable scenario in Thread 1 has no matching writes in the central forest structure. Hence, no action is taken.

- Concolic testing executes the newly insert inserted scenario, the program trace returned is $write(x, 0)$, $write(x, 2)$ - interference from Thread 2, $read(x)$, $x = 2$, $write(y = 1)$. There are no paths to be explored in Thread 2. The previously unrealizable scenario in Thread 1 has a matching shared write in Thread 3. Hence an interference is introduced before the shared read on y corresponding to the if($y > 0$). Inputs and thread schedule satisfying this newly generated scenario are generated.

- During the next iteration of the algorithm, the program is executed, with the newly generated inputs and thread schedule. The program takes the execution path given by, $write(y, 0)$, $write(y, 1)$ - interference from Thread 3, $y > 0$, $error$. Hence the error condition is reached.

In this way, con2colic performs a systematic search over the input and interleaving space to explore all possible program execution paths bounded by k interferences.

2.4 Summary

Detecting non-termination is an undecidable problem in general. Unlike verification of safety properties, that require a finite path witness, non-termination which can be viewed as

المنارة للاستشارات
www.manaraa.com

liveness violation requires the existence of an unbounded execution path. Most methods for detecting non-termination in sequential programs involve analyzing the structure of the program and how the data is modified. Detecting or proving non-termination in sequential programs has been vastly explored. Classical approaches include proving non-termination or proving termination. Techniques like invariant generation Velroyen and Rümmer (2008), Larraz et al. (2014), generation of recurrence sets Gupta et al. (2008), Chen et al. (2014), Cook et al. (2014) are used for constructing non-termination proofs. Ranking functions, well-foundedness of transition relation (Podelski and Rybalchenko (2004b), Cook et al. (2005), Cook et al. (2006), Podelski and Rybalchenko (2004a), Cook et al. (2010), Cook et al. (2013)) are techniques used for proving termination of sequential programs. In general, these techniques are sound or semi-automatic and incomplete.

In the concurrent domain, Morse et al. (2011), Musuvathi and Qadeer (2008), Atig et al. (2012a) represent three different approaches towards bounded model checking for verifying liveness properties. They handle state-space explosion by context-bounding, fair scheduling and scope-bounding respectively. Popeea and Rybalchenko (2012), Cook et al. (2007) are static analysis based methods that consider thread modular termination, while placing assertions on the other threads. Our approach is the closest to Atig et al. (2012a). This approach involves converting a concurrent program to an instrumented sequential program, violations of assertions in the sequential program are used to detect non-termination under fair scheduling. In general, all existing approaches are based on static analysis. To the best of our knowledge, our approach is the first testing based method for detecting the likelihood of non-termination in multi-threaded programs. Adopting a testing based methodology to detect non-termination, gives us certain significant advantages as compared to static analysis techniques. Firstly, testing based techniques are more scalable than static analysis based techniques. Secondly, false positives can be easily verified, since bugs reported by testing can be recreated. Lastly, testing based approaches do not require explicit adjustments to handle non-determinism introduced by the type of memory model or compiler optimizations.

In this thesis, we have focused on developing a generic testing based methodology to detect non-termination in multi-threaded programs by carefully inserting assertions. The modularity

المنارة للاستشارات
www.manaraa.com

of our implementation, allows our technique to be used in combination with any testing technique for multi-threaded programs. In this chapter, we presented a short discussion on some of the more complete testing techniques for testing muli-threaded programs. con2colic Farzan et al. (2013), an adaptation of concolic testing for multi-threaded programs, proves to be a promising technique in exhaustive test case generation by the systematic exploration of both the input and thread context space. In this thesis, we present a validation of our method's feasiblity by using random testing.

المنارة للاستشارات

www.manaraa.com

CHAPTER 3. TESTING FOR NON-TERMINATION

In this chapter, we describe our methodology to detect non-terminating execution sequences in multi-threaded programs. We focus on developing a testing based solution for detecting the likelihood of non-termination. Testing is only capable of verifying properties of finite program traces. Hence, our method involves identifying violations of carefully inserted assertions to infer possible inter-thread interferences, that can cause non-termination. We utilize a graph data structure *dependency graph* to represent and keep track of such dependencies. We reduce the problem of determining the potential of a concurrent program to have a non-terminating execution sequence, to that of detecting cycles in our *dependency graph*. Our method has the following features:

- ***Implicit handling of non-determinism caused by memory models***:
  Non-determinism is inherent in multi-threaded programs, one of the ways they are imparted, are through compiler optimizations and memory models. These optimizations can influence the order of reads and writes to potentially shared variables. Such changes in orderings might lead to unwanted data race conditions. Unlike static analysis based approaches, our approach being testing based, does not require explicit handling or adjustments to consider such non-determinism.

- ***Language agnostic***:
  The principal goal of our work, is to pave the way for utilizing testing based techniques in determining when a multi-threaded program may not terminate. We propose a methodology that can be easily adapted to testing programs written in a wide range of programming languages rather than developing an implementation that is specific to a programming language.

www.manaraa.com

- ***Modularity***:

  Our modular implementation gives the flexibility of pairing our technique with any of the popular testing methods like concolic by Sen et al. (2005), con2colic testing by Farzan et al. (2013), directed automatic random testing by Godefroid et al. (2005) and symbolic testing Cadar and Sen (2013). Our approach being testing based, the completeness and efficiency of our method is directly attributed by that of the underlying testing technique deployed. In this thesis, we have integrated our method with random testing to validate the feasibility of our approach.

3.1 Preliminaries

In order to explain our solution, we introduce the terms *inter-thread loop dependency*, *base count* and *count violation.* Consider a multi-threaded software involving n threads. Each thread's execution involves at most k loops.

Definition 3.1.1. An *inter-thread loop dependency* occurs when the execution of j-th loop in the i-th thread depends on the execution of j'-th loop in the i'-th thread. That is, the loop controlling parameter in j-th loop of i-th thread is modified by some j'-th loop in the i'-th thread.

We denote this by, $\langle i', j' \rangle \mapsto \langle i, j \rangle$.

Definition 3.1.2. The *base count* is the number of times a loop in a thread unfolds on it own, in the absence of interferences from other threads. Each loop in each thread is associated with a specific *base count*, that can be computed by executing the software in a controlled environment without any inter-thread interferences.

Definition 3.1.3. Given the execution of a concurrent software, a *count violation* occurs when there exists a loop in some thread such that the number of times the loop unfolds exceeds the *base count* due to *inter-thread loop dependencies.*

For a non-terminating execution sequence to occur due to interferences, we require more than one loop in different threads, whose execution continues to be trapped in loops, while the

www.manaraa.com

```
1: while $x \leq 5$ do
2:     Do-something
3:     $x++$
4:     $y--$
5: end while
```

(a) Thread 1

```
1: while $y \leq 5$ do
2:     Do-something
3:     $y++$
4:     $z--$
5: end while
```

(b) Thread 2

```
1: while $z \leq 5$ do
2:     Do-something
3:     $z++$
4:     $x--$
5: end while
```

(c) Thread 3

Figure 3.1 Example illustrating inter-thread loop dependency

execution of one of the threads affects the other. Hence, non-termination in multi-threaded programs is characterized by such mutual/circular *inter-thread loop dependencies* between more than one threads combined with *count violations*.

Formally, we denote this by saying there exists a likelihood of non-termination in multi-threaded programs, if,

$$\exists \langle i, j\rangle, \langle i', j'\rangle s.t. \langle i, j\rangle \mapsto^* \langle i', j'\rangle, \langle i', j'\rangle \mapsto^* \langle i, j\rangle \tag{3.1}$$

where, $i \in [1, n], n$ is the number of threads

where, $j \in [1, k], k$ is the maximum of number of loops in any thread.

where, $\mapsto^*$ is the transitive closure of $\mapsto$ over $T := \{1, ...n\}$

Consider the example in Figure.3.1 with three threads. We observe, that the loop controlling parameter in Thread 1 is modified by Thread 3, the loop controlling parameter in Thread 2 is modified by Thread 1 and loop controlling parameter in Thread 3 is modified by Thread 2. We note the following:

- We observe, transitive/circular *inter-thread loop dependencies* of the form $\langle 1, 1\rangle \mapsto \langle 2, 1\rangle$, $\langle 2, 1\rangle \mapsto \langle 3, 1\rangle$, $\langle 3, 1\rangle \mapsto \langle 1, 1\rangle$. Hence, there exists transitive closure of the form $\langle i, j\rangle \mapsto^* \langle i', j'\rangle$ and $\langle i', j'\rangle \mapsto^* \langle i, j\rangle$.
- For each of these threads, these *inter-thread loop dependencies* can cause a *count violation*.
- If these dependencies continue to interfere before each time the loop conditional is checked, these threads can be trapped in loops forever.

المنارة للاستشارات

www.manaraa.com

Using testing based techniques paired with appropriately selected assertions, to identify such dependencies is the central to our method. Our method, involves the following:

- Identify *count violations* due to *inter-thread loop dependencies*.
- Identify transitive closures of *inter-thread loop dependencies* over the set $T := \{1, ..., n\}$, as defined in Equation.3.1.

3.2 Dependency Graph

As discussed in the previous section, our method to identify the likelihood of non-termination relies on identifying *count violations* and *inter-thread loop dependencies*. These *count violations* and *inter-thread loop dependencies* are obtained by violation of carefully inserted assertional properties. In this section, we present the *dependency graph* data structure for encoding inter-thread loop dependencies and analyzing them to detect presence of non-terminating execution sequences.

Definition 3.2.1. Consider a program with n threads, with k to be the maximum number of loops in any thread. A *dependency graph* $DG = (V, E)$, where V is the set of nodes and E is the set of directed edges($E \subseteq V \times V$). A node $v \in V$ is labeled with a set of at most t $(1 \leq t \leq n \times k)$ tuples of the form $\langle i, j \rangle$, where, $i \in [1, n]$ and $j \in [1, k]$. Here, the i's are pair-wise disjoint. Any edge $e \in E$ has a source and destination node such that, the destination node is labeled with exactly one tuple.

Definition 3.2.2. The semantics of the edge is as follows: An edge from node v to v' indicates the inter-thread loop dependency as follows:

(a) the destination node v' of any edge is labeled by one tuple $\langle i', j' \rangle$.

(b) If the source node v of the edge is labeled by $\{\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle ... \langle i_l, j_l \rangle\}$, then the *counting violation* of j'-th loop in the i'-th thread involves atleast one j_m-th loop in the i_m-th thread, $\forall m \in [1, l]$.

Figure.3.2 shows a sample dependency graph. Each node represents a thread-loop pair in the program and the edges represent *inter-thread loop dependencies* between thread-loop pairs.

www.manaraa.com

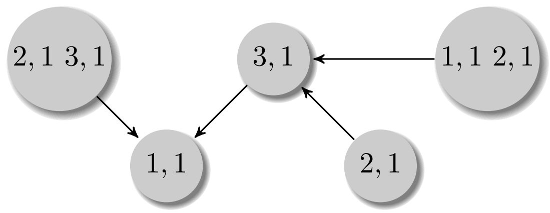


Figure 3.2 An example of a *dependency graph*

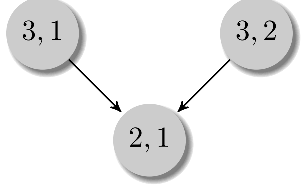


Figure 3.3 An example of a *dependency graph*

For example, the edge from node with label $\langle 2, 1\rangle$ to node with label $\langle 3, 1\rangle$, indicates that there exists a dependency from thread-loop pair $\langle 2, 1\rangle$ to thread-loop pair $\langle 3, 1\rangle$, which causes a *count violation* in the latter. Similarly, the edge from node withl label $\langle\langle 2, 1, \rangle\langle 3, 1\rangle\rangle$ to a node with label $\langle 1, 1\rangle$ indicates that both the thread-loop pairs $\langle 2, 1\rangle, \langle 3, 1\rangle$ interfere with the thread-loop pair $\langle 1, 1\rangle$, causing a count violation in the latter.

3.2.1 Constructing *dependency graph*

As discussed in the previous sections, when represent the *inter-thread loop dependencies* causing *count violations* in the *dependency graph.* Let us consider, an *inter-thread loop dependencies* from a set of thread-loop pairs AV to a thread-loop pair v. Every sub-group of thread-loop pairs in AV, such that they have the same $threadidentifier$, is treated as a disjunctive dependency. That is, there exists an *inter-thread loop dependency* from either one of these thread-loop pairs, that causes a *count violation* in the destination node v. Every thread-loop pair with distinct $threadidentifier$ in AV, is treated as a conjunctive dependency. In this case, for aN *inter-thread loop dependency* to cause a *count violation* in destination node v, the dependency must involve all the thread-loop pairs with distinct $threadidentifiers$.

We construct the *dependency graph* by segregating these dependencies into a disjunction of conjunctions. For each conjunctive dependency, we represent the thread-loop pairs involved in

www.manaraa.com

Algorithm 1 Algorithm for constructing *dependency graph*

```
procedure ConstructGraph(v, AV, G)
    $addnode(v)$
    $M = \emptyset$                                   ▷ 2-D list, each sub list is a list of tuples
    while $AV \neq \emptyset$ do
        $templist = \emptyset$                         ▷ temporary list of tuples
        $thread \leftarrow$ thread id of $AV[0]$ (first element of E)
        $templist \leftarrow AV[0]$
        remove $AV[0]$
        for $(i = 0; i <| AV |; i++)$ do
            if thread id of $AV[i] = thread$ then
                append $AV[i]$ to $templist$
            else
                break
            end if
        end for
        if $M \neq \emptyset$ then
            $sindex \leftarrow 0$
            $eindex \leftarrow| M |$
            for $(j = 0; j <| templist |; j++)$ do
                for $(k = sindex; k < eindex; k++)$ do
                    if $j <| AV |$ then
                        append $M[k]$ to $M$
                    end if
                    append $templist[k]$ to $M[k]$
                end for
                $sindex \leftarrow eindex$
            end for
        else
            $index = 0$
            for $(l = 0; l <| templist |; l++)$ do
                $M[index] \leftarrow templist[l]$
            end for
        end if
    end while
    for (each sub list n in M) do
        $addnode(n)$
    end for
end procedure
```

المنارة للاستشارات
www.manaraa.com

the conjunction in a single node and insert an edge from this node to the node v. Consider $AV = \langle\langle 1,1\rangle,\langle 2,1\rangle\rangle$ and $v = \langle 3,1\rangle$, there exists a conjunctive dependency where, thread-loop pair $\langle 3,1\rangle$ is dependent on both the thread loop pairs in AV. This is indicated by creating a node with label $\langle\langle 1,1\rangle,\langle 2,1\rangle\rangle$ and a node with label $\langle 3,1\rangle$ and creating an edge from the former to the latter as indicated in Figure.3.2. Consider $AV = \langle\langle 3,1\rangle,\langle 3,2\rangle\rangle$ and $v = \langle 2,1\rangle$, there exists a disjunctive dependency where, thread-loop pair $\langle 2,1\rangle$ depends on either one of the thread-loop pairs in AV. Such an *inter-thread loop dependency* is updated in the following manner: a node with label is added $\langle 3,1\rangle$, a node with label $\langle 3,2\rangle$ is added, edges from each of these nodes are added to a node with label $\langle 2,1\rangle$. This is indicated in the *dependency graph* given by Figure.3.3. This kind of dependency occurs in nested loops. When a thread-loop pair is dependent on one of the loops in a set of nested loops, it is not feasible to isolate the exact source of *inter-thread loop dependency.* In this case, we perform a safe-approximation by storing this as a disjunctive dependency. We insert a node for each loop in the set of nested loops and an edge from each of these nodes to the node representing the dependent thread-loop pair.

We describe our algorithm for constructing and later updating the dependency graph in Algorithm.1. The procedure takes the parameters v, AV, and G as inputs, we begin with an empty graph. The following are the parameters used by this procedure:

- v- a tuple of form ⟨thread identifier, loop identifier⟩ representing the destination node.
- AV- a sorted list of tuples $\langle threadidentifier, loopidentifier\rangle$ representing the *inter-thread loop dependencies* that could be causing possible *count violations* in the execution of the loop in the thread represented by v. The list is sorted by $threadidentifier$. We get AV from testing.
- G- *dependency graph.*
- M- a list of list of tuples t. t are of the form $\langle threadidentifer, loopidentifier\rangle$. At the end of the computation, each sub list of tuples will be stored as a node.
- *templist*- a temporary list of tuples of form t for computation purposes.

www.manaraa.com

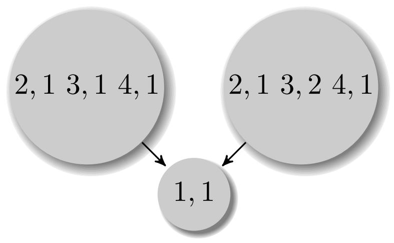


Figure 3.4 Example for illustrating the construction of a *dependency graph*

Our algorithm to update the *inter-thread loop dependencies* involves the following steps:

- The node v is added to G.
- The following actions are performed in a loop until all the tuples in AV are processed.
  - Let $thread = threadidentifier$ of first thread-loop pair in AV. The first element in AV is removed and added to temporary list *templist*.
  - Every thread-loop pair in AV, that has the same $threadidentifier$ as $thread$ is added to *templist*.
  - if M is non-empty: if the length of *templist* is *len*, $len - 1$ copies of M are created and appended to M, while a thread-loop pair of *templist* is added to each newly created copy of M.
  - if M is empty: for each thread-loop pair in the *templist*, a new sub list is created in M, and the thread-loop pair in *templist* is added to the newly created sub list.
- For each sublist of thread-loop pairs in M, a node is added and an edge is added from the newly added node to the node v in the graph G.

Consider, the destination node v to be $\langle 1, 1\rangle$. Let $AV = \langle\langle 2, 2\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle\rangle$. The construction or updation of *dependency graph* involves the following steps: Our procedure adds the destination node v to the *dependency graph*.

iteration 1: *templist* is cleared, $thread = 2$ ($threadidentifier$ of first element in AV: $\langle 2, 2\rangle$). $\langle 2, 2\rangle$ is removed from AV and pushed into *templist*. No further updation to *templist* is done, since no other thread loop pair in AV has the same $threadidentifier$. M is empty, therefore, a new sub list in M is created, $\langle 2, 2\rangle$ is pushed into the newly created sub list.

www.manaraa.com

iteration 2: *templist* is cleared, *thread* $= 3$, corresponding to the first element in $AV : \langle 3, 1\rangle$, $\langle 3, 1\rangle$ is removed from AV and pushed into *templist*. There exists a thread-loop pair $\langle 3, 2\rangle$, with the same *threadidentifier* as *thread*. Hence, $\langle 3, 2\rangle$ is pushed into *templist*. Now, M is non-empty, already containing a sub-list with the thread-loop pair $2, 1$. There exist 2 elements in the *templist*, corresponding to this, a copy of M of created and it is appended to M. M now has, two sub lists, each with the element $\langle 2, 1\rangle$. Each thread-loop pair in *templist* is appended to each copy of the original version of M. $M = \langle\langle\langle 2, 1\rangle, \langle 3, 1\rangle\rangle, \langle\langle 2, 1\rangle, \langle 3, 2\rangle\rangle\rangle$.

iteration 3: *templist* is cleared, *thread* $= 4$, corresponding to the first thread-loop pair in $AV : \langle 4, 1\rangle$. $\langle 4, 1\rangle$ is removed from AV and added to *templist*. No further updation is done to *templist* since, there exists no more thread-loop pairs with the same *threadidentifer*. $M = \langle\langle\langle 2, 1\rangle, \langle 3, 1\rangle\rangle, \langle\langle 2, 1\rangle, \langle 3, 2\rangle\rangle\rangle$. *templist* simply contains one thread-loop pair, this thread-loop pair is appended to each sub list in M. M is now $\langle\langle\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 4, 1\rangle\rangle, \langle\langle 2, 1\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle\rangle\rangle$.

There are no more thread-loop pairs in AV to be processed. A node is added for every sub list in M. Edges are added from the newly added nodes to the destination node $\langle 1, 1\rangle$. This results in a *dependency graph* as shown in Figure.3.4.

3.2.2 Detecting non-termination

The *dependency graph* representation described in the previous sections, allows us to detect non-termination in a multi-threaded program through cycle detection in the program's *dependency graph*. From Equation.3.1, we know that there exists a non-terminating execution sequence in a concurrent program, if there exist two thread-loop pairs, such that there exist symmetry in the transitive closures over their *inter-thread loop dependency* relations. That is, there exist thread-loop pairs $\langle i, j\rangle$, $\langle i', j'\rangle$, such that the $\langle i, j\rangle \mapsto^* \langle i', j'\rangle$ and vice versa. Detecting such symmetry in transitive closure directly translates to cycle detection in *dependency graph*, since an edge e from node(thread-loop pair(s)) v to v' in the graph represents an *inter thread loop dependency* from v to v' causing a *count violation* in v'. We now present our definition of cycle, which is slightly different from looking for traditional cycle in graphs.

www.manaraa.com

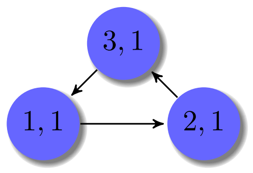


Figure 3.5 Dependency graph with a simple cycle

Definition 3.2.3. A path in a graph G is a finite or infinite sequence of edges that connects a sequence of nodes in G. We say a graph G, has a cycle when one of the following conditions are met:

- Simple cycles: There exists a path from a node labelled $\langle i, j\rangle$, such that the node labelled $\langle i, j\rangle$ is revisited. In this path, there can exist no intermediate node with a label having multiple thread-loop pairs.

- Complex cycles: There exists a path starting from a node labelled with multiple thread-loop pairs, $\langle\langle i_1, j_1\rangle, \langle i_2, j_2\rangle ... \langle i_l, j_l\rangle\rangle$, such that, a node labelled i_m, j_m is visited $\forall m \in [1, l]$.

Figure.3.5 shows a *dependency graph* with a simple cycle given by: $\langle 1, 1\rangle, \langle 2, 1\rangle, \langle 3, 1\rangle$. Figure.3.6 shows a *dependency graph* with a complex cycle given by: $\langle\langle 1, 1\rangle, \langle 3, 1\rangle\rangle$, $\langle 2, 1\rangle$, $\langle 3, 1\rangle$, $\langle 1, 1\rangle$. This is because, there exists a path from $\langle\langle 1, 1\rangle, \langle 3, 1\rangle\rangle$ in which each of the thread-loop pairs $\langle 1, 1\rangle$ and $\langle 3, 1\rangle$ pairs are individually visited. A complex cycle should be viewed as a logical cycle in the *inter-thread loop dependencies* between thread loop pairs as opposed to a physical cycle in the *dependency graph*. In Figure.3.6, we find the following cyclic *inter thread loop dependency*, thread-loop pair $\langle 2, 1\rangle$ is dependent on thead-loop pairs $\langle\langle 1, 1\rangle, \langle 3, 1\rangle\rangle$, while thread-loop pair $\langle 3, 1\rangle$ is dependent on thread-loop pair $\langle 2, 1\rangle$ and in turn, thread-loop pair $\langle 1, 1\rangle$ is dependent on thread-loop pair $\langle 3, 1\rangle$. Hence, we say that there exists a non-terminating execution sequence given by this complex cycle.

Algorithm.2 describes our CYCLE(G) used to detect the occurrence of non-terminating execution sequences by cycle detection in *dependency graph*. The algorithm has two sections. The first sections detects simple cycles as described in Algorithm.3 , for every node v in the graph G, it executes a call to SIMPLECYCLE(v, *recstack*). Here, *recstack* represents the recursion stack. An outline of SIMPLECYCLE procedure is available in Algorithm.3. This is a

المنارة للاستشارات
www.manaraa.com

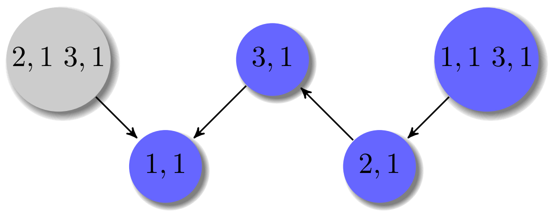


Figure 3.6 Dependency graph with a complex cycle

Algorithm 2 Outline of the Algorithm for detecting non-termination

```
1:  procedure CYCLE($G$)
2:      for (every node $v$ in $G$) do
3:          if SimpleCycle($v, recstack$) then
4:              return true
5:          end if
6:      end for
7:      for (every node $v$ with multiple tuples in label) do
8:          mark all nodes as not visited
9:          DFS(v)
10:         if (Each tuple in label of $v$ occurs in $list$) then
11:             return true
12:         end if
13:     end for
14: end procedure
```

Algorithm 3 Outline of the Algorithm for detecting simple cycles

```
1:  procedure SIMPLECYCLE($v, recstack$)
2:      if node $v$ is not $visited$ then
3:          mark $v$ as $visited$
4:          push $v$ onto $recstack$
5:      end if
6:      for (every adj node $v'$ of $v$) do
7:          if ($v'$ is not $visited$ & SimpleCycle($v', recstack$)) then
8:              return true
9:          else if ($v'$ is in $recstack$) then
10:             return true
11:         end if
12:     end for
13:     pop $recstack$
14:     return false
15: end procedure
```

المنارة للاستشارات
www.manaraa.com

Algorithm 4 Outline of the DFS Algorithm for detecting complex cycles

```
1: procedure DFS($v$)
2:     mark $v$ as visited
3:     for (every tuple $i$ in the label of $v$) do
4:         $list \leftarrow i$
5:     end for
6:     for (every adjacent node $v'$ of $v$) do
7:         if ($v'$ is not visited) then
8:             DFS($v'$)
9:         end if
10:    end for
11: end procedure
```

regular DFS based cycle detection. DFS traversal is performed over the graph G to create the equivalent DFS tree or forest, a cycle is detected by detecting a back edge. For each node v in a graph, the node is marked visited and pushed onto the recursion stack. For each adjacent node v' of v, SIMPLECYCLE is recursively called. When an node v' that is present in the recursion stack is revisited, a cycle or back edge is detected.

The second section detects complex cycles as described in Algorithm.4, for every node v with multiple tuples in the label, the procedure DFS(v) is called. For every node v with multiple labels, a depth first traversal is done. Each node visited during this traversal is added to a global data structure *list*. CYCLE(G) iterates through this list to check if all the tuples representing the node v are revisited individually. If a cycle is found, the procedure returns true and false otherwise.

For the *dependency graph* in Figure.3.5, our CYCLE(G) detects the simple cycle through the DFS based cycle check discussed in SIMPLECYCLE. Let us consider, the *dependency graph* in Figure.3.6. There exists a complex cycle given by $\langle\langle 1,1\rangle,\langle 3,1\rangle\rangle,\langle 2,1\rangle,\langle 3,1\rangle,\langle 1,1\rangle$. There exists a path from $\langle\langle 1,1\rangle,\langle 3,1\rangle\rangle$ in which each of the thread-loop pairs $\langle 1,1\rangle$ and $\langle 3,1\rangle$ pairs are individually visited. DFS performs a depth first traversal staring from node $\langle\langle 1,1\rangle,\langle 3,1\rangle\rangle$ and pushes the label of each node visited onto a list. This list is later explored to identify the complex cycle and CYCLE(G) returns true.

المنارة للاستشارات
www.manaraa.com

Algorithm 5 Outline of the testing algorithm for detecting non-termination

```
procedure EXPLORE($J$, $B$, $k$)
    if k $\geq$ n then
        return false
    else
        if $J = \emptyset$ then
            $J \leftarrow B$
            $k \leftarrow k + 1$
        end if
        $i \leftarrow$ *threadid* of $j_0$ | $j_0$ is first element of J
        $j \leftarrow$ *loopid* of $j_0$
        $m \leftarrow$ *basecount* of $j_0$
        remove $j_0$
        $C \leftarrow$ all $k - 1$ combinations of $x \in \{y \mid y \in [1, n] \wedge y \neq i\}$ appended by $i$
        insert assertion in $j$-th loop of $i$-th thread: $assert(cnt_{ij} \leq m)$
        for (each $c_i \in C$) do
            $AV \leftarrow$ testing($c_i$)
            if ($AV \neq \emptyset$) then
                ConstructGraph ($\langle i, j\rangle, G, AV$)
                if Cycle($G$) then
                    return true
                end if
            end if
        end for
        Explore ($J$, $B$, $k$)
    end if
end procedure
```

www.manaraa.com

3.3 Non-termination by testing

Our method for determining non-termination in multi-threaded programs, involves determining the *base count*, followed by calling the EXPLORE procedure that performs testing over increasing combinations of threads recursively. The first step is to pre-instrument the program of interest. For each loop in each thread, count variables (cnt_{ij} for the j-th loop in the i-th thread) are inserted in such a way, that they are incremented each time a loop in a thread unfolds. Recall, that each loop in each thread is associated with a *base count* that indicates the number of times, the loop executes on its own, without any interferences. The first step of our approach involves determining the *base count*.

3.3.1 Determining *base count*

We execute each thread without any interferences to determine *base count*, since we assume that each thread terminates, when executed on its own. After each thread is executed, their thread identifiers, loop identifiers and their *base count* are updated in the list B, which we call the *base list*. Consider, a thread having thread identifier 4 with 3 loops is executed sequentially. Let, m_{41}, m_{42}, m_{43} be the *base count* that we determined. The tuples $\langle 4, 1, m_{41}\rangle, \langle 4, 2, m_{42}\rangle, \langle 4, 3, m_{43}\rangle$ are appended to B.

3.3.2 Testing for non-termination by EXPLORE

The EXPLORE procedure tests different combinations of threads recursively. It commences testing by considering all combinations of 2 threads to discover *inter-thread loop dependencies* that require a single thread-loop pair to cause a *count violation* in another thread-loop pair and for each round of recursion, it increases the number of threads tested by 1, to incrementally discover dependencies. Let us consider, our input program to have n threads, with l being the maximum number of loops in any thread. Once the *base count* is determined, the EXPLORE procedure is called. Algorithm.5 shows an outline of our EXPLORE procedure, that performs testing for detecting non-termination. The procedure takes B, J, and k as inputs.

المنارة للاستشارات

www.manaraa.com

- B- base list contains a list of all loops in the program, along with their thread identifier and *base count*. It is a list of tuples of the form $\langle threadidentifier, loopidentifier, basecount\rangle$. It represents a master list of all the jobs (thread-loop pairs) to be explored.

- J- job list contains current working list of pending jobs or thread-loop pairs. It is also a list of tuples of the form $\langle threadidentifier, loopidentifier, basecount\rangle$ The procedure begins with $J = \emptyset$ and populates it from the base list B.

- k- the number of threads selected for current iteration of testing.

We start by loading the list of jobs into J from B and by setting $k = 2$. For each entry in the J, we perform two actions. First action is to instrument the program with appropriate assertions. Consider, the current job in J to be $\langle i, j, m_{ij}\rangle$. After the last statement of the j-th loop in the i-th thread, we insert an assertion of the form $assert(cnt_{ij} < m_{ij})$, where the cnt_{ij} represents the count variable instrumented and m_{ij} represents the *base count* value. The second action is to create a combination list C. The C contains a list of list of threads to be selected for the current testing iteration. This list is populated by doing the following:

- We compute, all possible $k-1$ combinations of $x \in T$, where $T := \{x \mid x \in [1, n] \wedge x \neq i\}$.

- Each $k - 1$ combination of threads is appended by i.

- Each k combination of threads is now pushed into C.

Once the C is populated, for each entry in the list, those specific threads are selected for testing. Testing returns AV. If an assertion violation on the cnt_{ij} is observed, AV contains a list of count variables that appeared in the execution sequence leading to the violation. AV is a list of $\langle threadidentifier, loopidentifier, basecount\rangle$, where, $threadidentifier \neq i$. If no assertion violation was encountered, the AV contains $NULL$. If an assertion violation is encountered, the *dependency graph* G is updated, and the graph is checked for cycles by CYCLE(G). If a cycle is found, the procedure terminates, announcing the likelihood of non-termination, the *dependency graph* is displayed.

In the event of not finding a cycle, the method proceeds with testing until all the combinations in C are exhausted. If we have performed testing for all jobs in J, we repeat instrumenting

www.manaraa.com

assertions, computing C, testing and cycle checks until we find a non-termination or exhaust all jobs in J. Once we have exhausted all the pending jobs for the current combination, but we have still not detected non-termination, we increment k and retry our testing procedure. This helps us to explore more interference scenarios in execution sequences introduced by considering more number of threads at the same time.

Let us consider the example in Figure.3.1. We now illustrate how our method can be used to discover the non-termination sequence that can occur, when in each thread, before the loop conditional is checked, if an interference modifies the loop controlling parameter.

- We begin by determining the *base count*, by executing each thread in a completely sequential fashion. If the values of x, y and z are initialized to 0, we find B $= \langle\langle 1, 1, 5\rangle, \langle 2, 1, 5\rangle, \langle 3, 1, 5\rangle\rangle$.
- We call the procedure EXPLORE to begin testing. J is initialized to B, k is initialized to 2. The first element in J, j_0 is $\langle 1, 1, 5\rangle$.
- After the last statement of the first loop in the first thread, an assertion of the form $assert(cnt_{11} \leq 5)$ is inserted. Combinations of size k are computed. $C = \langle\langle 1, 2\rangle, \langle 1, 3\rangle\rangle$. j_0 is removed from the J.
- For the first element in C, $\langle 1, 2\rangle$, testing is done after Thread 1 and Thread 2 are selected. No assertion violation is found.
- For the next element in C, $\langle 1, 3\rangle$, testing after appropriate thread selection reveals assertion violation on cnt_{11}. In the execution sequence leading to the assertion violation, we find cnt_{31} corresponding to the first loop in Thread 3.
- We update graph G with a node $\langle 1, 1\rangle$, a node $\langle 3, 1\rangle$ and finally an edge from $\langle 3, 1\rangle$ to $\langle 1, 1\rangle$. On checking for cycles, the CYCLE(G) returns false.
- We have exhausted all the thread combinations in C. For the first element in J, $\langle 2, 1, 5\rangle$, we perform instrumentation in Loop 1 of Thread 2. We populate the C with $\langle\langle 2, 1\rangle, \langle 2, 3\rangle\rangle$.
- For the combination $\langle 2, 1\rangle$, testing yields an assertion violation on cnt_{21} with cnt_{11} in the program trace.

المنارة للاستشارات
www.manaraa.com

- Corresponding to the assertion violation, we update graph G with a node $\langle 2, 1\rangle$ and an edge from $\langle 1, 1\rangle$ to $\langle 2, 1\rangle$($\langle 1, 1\rangle$ is duplicate and hence not inserted). Our non-termination check returns false.

- For the combination $\langle 2, 3\rangle$, testing does not yield an assertion violation. We have now exhausted all combinations in C.

- For the next element in J, $\langle 3, 1, 5\rangle$, we insert an appropriate assertion after the last statement of Loop 1 in Thread 3. We compute thread combinations, $C = \langle\langle 3, 1\rangle, \langle 3, 2\rangle\rangle$.

- Corresponding to $\langle 3, 1\rangle$, appropriate threads are selected, testing yields no assertion violation.

- Corresponding to $\langle 3, 2\rangle$, appropriate threads are selected, testing gives an assertion violation on cnt_{31} with cnt_{21} in the program trace. We update G, with an edge from $\langle 2, 1\rangle$ to $\langle 3, 1\rangle$. The termination check CYCLE(G) returns true upon find the simple cycle between $\langle 1, 1\rangle$, $\langle 2, 1\rangle$ and $\langle 3, 1\rangle$.

- Our program terminates indicating the presence of a possible non-terminating execution sequence and displays the *dependency graph* as shown in Figure.3.5.

Our novel incremental testing approach gives us several significant advantages. If, we begin testing by considering all n threads together, isolating the specific inter-thread loop dependency that caused the *count violation* will be a laborious task. This would also complicate the cycle detection in our *dependency graph*. Further, if there exists an non-terminating scenario due to *inter-thread loop dependencies* between just two threads, we would be exploring bulkier scenarios involving more number of threads which is of little relevance. In this case, our technique with incremental testing would also lead to faster discovery of non-termination as compared to selecting all threads for testing.

Hence, we presented a testing based technique to detect the presence of unbounded execution paths due to unwanted race conditions. Our technique leverages the ability of testing based approaches to quickly verify assertional properties. We have developed an efficient method that expresses non-termination in multithreaded programs in assertional properties. Finally, we infer

www.manaraa.com

the likelihood of non-terminating program paths from the violation of these carefully selected assertions over bounded execution paths.

3.4 Tool Description

We present an overview of our language independent, generic, testing based framework for detecting non-termination. Our tool is implemented using a combination of C++ and Python. Our tool involves recursive guided testing, where we systematically determine *inter-thread loop dependencies* by considering all possible combinations of $k = 2$ threads to begin with, and for each new iteration, we consider $k + 1$ combinations of threads for testing. This is done either, till all possible combinations are tested for all $k \leq n$, with n being the number of threads in the program, or non-termination is observed.

3.4.1 Tool overview

Figure.3.7 illustrates a simple block diagram of our tool. The inputs to our tool are the pre-instrumented multi-threaded program to be tested and the number of threads in the test program. The pre-instrumentation involves the following: For each loop j in each thread i, a cnt_{ij} variable is inserted. In the j-th loop of the i-th thread, a statement $cnt_{ij}{+}{+}$ is included as the first statement, such that, the count variable is incremented every time the loop is executed. Additional pre-instrumentation is done, such that, every time a cnt_{ij} variable is incremented, it is written to a temporary file. This temporary file is used for computation purposes by our tool. A code-snippet from one of the pre-instrumented test programs is shown in Figure.3.8, where, the lines of code 10, 12, 18 - 21 were inserted as a part of pre-instrumentation.

1. The base count evaluator determines the *base count* for each thread-loop pair in the test program.
2. For each thread-loop pair, i, j, the instrumentation tool inserts an appropriate assertion of the form $assert(cnt_{ij} \leq basecount)$.
3. Corresponding to the thread identifier i of the current thread-loop pair being tested, a combination generator generates all possible k combinations of i with the other threads

www.manaraa.com

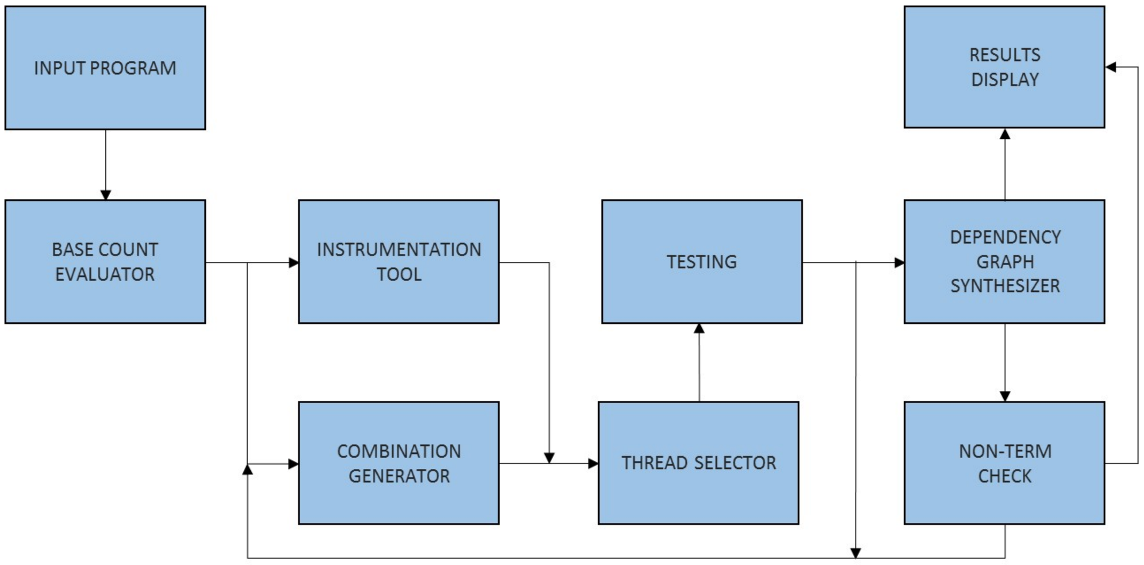


Figure 3.7 Architecture diagram of our tool

in the program. This combination of k threads, corresponds to the set of threads that need to be selected for the current phase of testing.

4. The instrumentation tool and the combination generator pass the current thread-loop pair, test program carefully instrumented with assertions and the current combination to the thread selector. The thread selector selects the threads indicated in the current combination for testing.

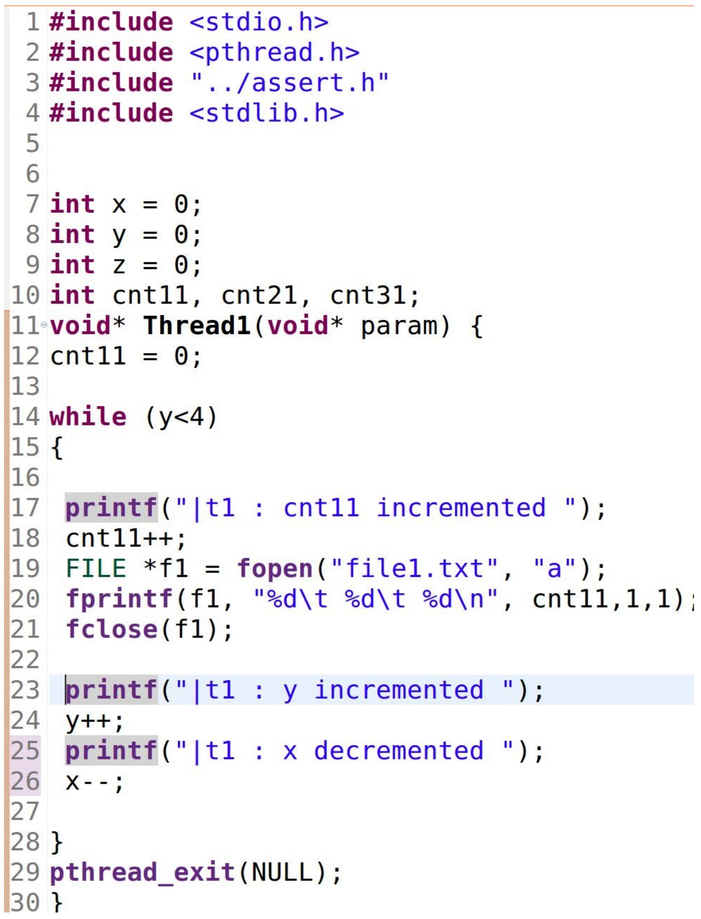
5. Following the instrumentation and thread selection, the test program is tested. Our algorithm uses simple random testing for verifying the inserted assertions.

6. If an assertion violation is encountered, the *dependency graph* synthesizer records this *inter-thread loop dependency.* Also, a non-termination check is performed, which checks for cycles in the *dependency graph.*

7. If the non-term check returns true, our tool terminates, presenting the scenario leading to non-termination.

8. This procedure is repeated till all k combinations of all thread-loop pairs are tested for all values of $k <= n$ (n- number of threads in test program) or non-termination is detected.

www.manaraa.com



```
#include <stdio.h>
#include <pthread.h>
#include "../assert.h"
#include <stdlib.h>


int x = 0;
int y = 0;
int z = 0;
int cnt11, cnt21, cnt31;
void* Thread1(void* param) {
cnt11 = 0;

while (y<4)
{

 printf("|t1 : cnt11 incremented ");
 cnt11++;
 FILE *f1 = fopen("file1.txt", "a");
 fprintf(f1, "%d\t %d\t %d\n", cnt11,1,1);
 fclose(f1);

 printf("|t1 : y incremented ");
 y++;
 printf("|t1 : x decremented ");
 x--;

}
pthread_exit(NULL);
}
```

Figure 3.8 A snippet of pre-instrumented code

المنارة للاستشارات

www.manaraa.com

3.4.2 Tool components

- **Base count evaluator**:

  The base count evaluator, takes the pre-instrumented multi-threaded program as input. For each thread-loop pair in the test program, it determines the *base count*. This is done by executing each thread on its own, without interferences from other threads. As described in the previous sections, the base count evaluation returns a base list B with thread-loop pairs and their respective *base count* values.

- **Instrumentation tool**:

  Given a thread-loop pair i, j, the instrumentation tool parses the input program. Following the last statement of the j-th loop in the i-th thread, it inserts an assertion of type $assert(cnt_{ij} \leq basecount)$.

- **Combination generator**:

  Given a thread-loop pair i, j, the combination generator generates a combination list C. This list is populated by synthesizing k- sized combinations of i with $j \in T \wedge j \neq i$ (T is the set of all threads in the test program). This combination corresponds to the k threads that will considered for testing.

- **Thread selector**:

  Given a thread combination of size k, the thread selector parses through the input code. A new program is created from the instrumented program outputted by the instrumentation tool. The thread selector ensures, that the new program created, only consists of the threads selected for the current phase of testing. This instrumented, thread selected program is now ready for testing.

- **Testing** :

  Our tool uses random testing for detecting *inter-thread loop dependencies* modeled as assertions. To explore different possible program behavior due to different input valuations, we use a random input generator while determining the *base count*. Once *base count* is determined, we reuse the input valuations for recursive testing. To explore different

www.manaraa.com

possible program behavior due to different possible schedules, we repeatedly execute the program till an assertion violation is encountered or an upper bound for the number of execution is reached. If the upper bound is reached, we conclude that no *inter-thread loop dependency* was found and proceed with the next combination for testing. We have conducted our experiments with the upper bound on the number of executions to be 200. It is important to note, that an upper bound for number of executions of 200, does not imply that 200 distinct schedules are explored, since we rely entirely on the underlying compiler to generate schedules. Hence, this does not guarantee, that 200 distinct program behaviors were explored. If our method concludes, that no non-termination could be found, we recommend retrying testing by increasing this value or by increasing the rangle of the random input generator. This way, more interference scenarios or program behaviors can be explored as compared to the previous case. Previously, we attempted to use CONCREST Farzan et al. (2013), which is a concolic testing technique for multi-threaded programs. In general, we observed that CONCREST had scalability issues. We provide a complete discussion on our experience with concolic testing in Chapter 4.

- **Dependency graph synthesizer**:
  This is an implementation of the ConstructGraph procedure, which is central to our tool. As discussed earlier, the *dependency graph* is the data structure, that is used to encode the observed *inter-thread loop dependencies* causing *count violations*. When an assertion violation is discovered from testing, a *dependency graph* is constructed by adding nodes and edges corresponding to the *inter-thread loop dependency involved*. Before the updation of the dependency, a duplication check is run to ensure that only distinct *inter-thread loop dependencies* are recorded. This facilitates an efficient non-termination check.

- **Non-term check**:
  This is singularly, the most important component of our tool. Each time the *dependency graph* is updated, the Non-term check is run. This component is an implementation of Cycle and uses SimpleCycle to check for simple cycles, and DFS to detect complex cycles as defined in Definition.3.2.3. These procedures are modified implementations of

المنارة للاستشارات

www.manaraa.com

a DFS based cycle detection. If either one of the cycles are detected, the non-term check returns true indicating the likelihood of a non-terminating execution sequence in the program tested.

- **Results display**:

  Our tool terminates giving one of the following outputs:

  - Reports likelihood of non-termination, displays the adjacency list of the *dependency graph* and the scenario leading to non-termination. The scenario consists of a list of thread-loop pairs, which could lead to possible non-termination. In other words, this is a list of thread-loop pairs which form a cycle in our *dependency graph*.
  - Reports that no non-termination was found and displays the adjacency list of the *dependency graph*.
  - Time-out (7200s / 2 hrs): Reports that no non-termination was found and displays the adjacency list of the *dependency graph*.

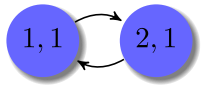
www.manaraa.com

CHAPTER 4. RESULTS

We now present an experimental evaluation of our approach to detect non-termination in multi-threaded programs. For the performance evaluation and scalability study of our technique, there are no existing benchmarks available. Hence, we use a systematic enumeration of different possible scenarios or execution patterns that could lead to non-termination in multi-threaded programs as benchmarks for the validation of our approach's correctness and feasibility. We present the results of this evaluation in Table.4.1. In Sections 4.1 to 4.3, we describe the enumeration of non-terminating patterns in multi-threaded programs, we present a discussion on representative example patterns from Table.4.1.

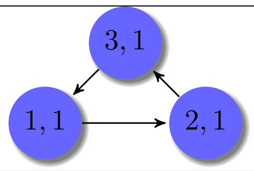
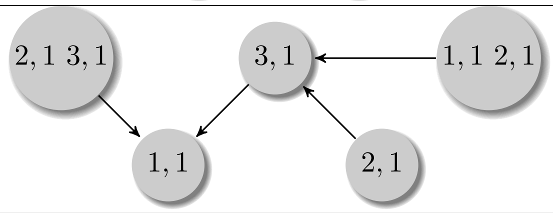
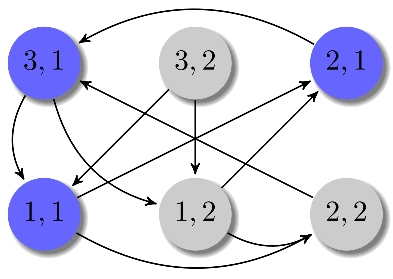
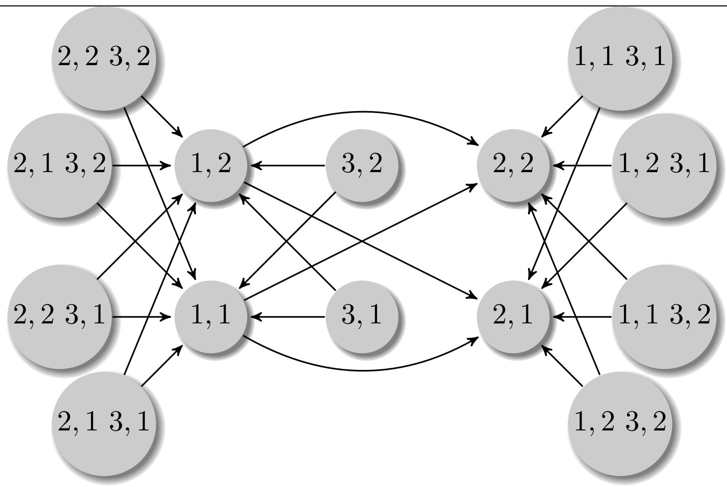
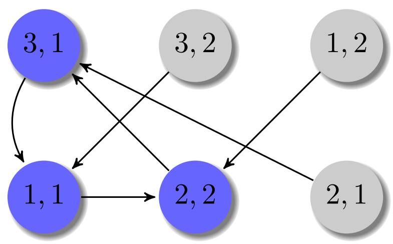
Pattern refers to execution patterns, that could lead to non-termination. Each pattern consists of thread-loop pairs as its elements. The j-th loop in the i-th thread is represented by $\langle i, j\rangle$. Similar to the *dependency graph*, an edge from an element $\langle i, j\rangle$ to $\langle i', j'\rangle$ represents, that there exists an *inter-thread loop dependency* from $\langle i, j\rangle$ to $\langle i', j'\rangle$, causing a *count violation* in $\langle i', j'\rangle$. Now, if the edge has a label $\langle i'', j''\rangle$, there exists an *inter-thread loop dependency* from $\langle i, j\rangle$, $\langle i'', j''\rangle$, to $\langle i, j\rangle$ causing a *count violation* in $\langle i, j\rangle$. **n** refers to the number of threads in the test code, **Dependency Graph** shows the resulting dependency graph. **t(s)** gives the tool's overall execution time, **R** gives the result outputted by our tool ('T' for non-termination and 'F' otherwise).

Table 4.1 Results of experimental evaluation

#	Pattern	n	Dependency graph	t(s)	R
1	$\begin{tikzcd} 1,1 \arrow[r] & 2,1 \arrow[l, bend left] \end{tikzcd}$	2		2.3	T

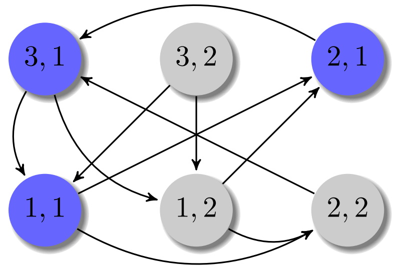
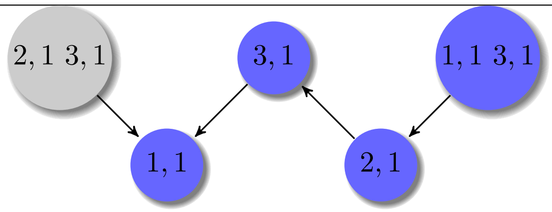
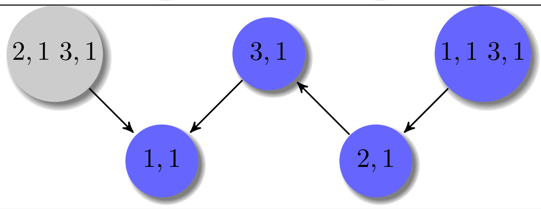
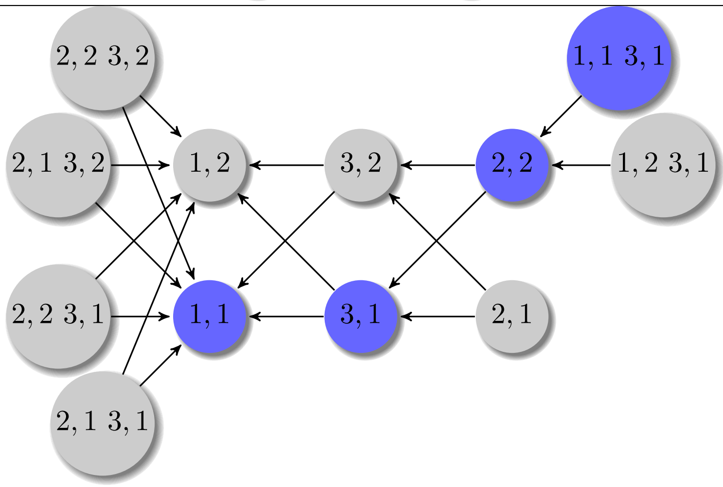
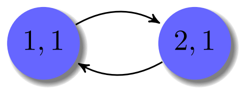
المنارة للاستشارات
www.manaraa.com

Table 4.1 (Continued)

#	Pattern	n	Dependency graph	t(s)	R
2	$\begin{tikzcd} 1,1 \arrow[r] & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$	3		100.6	T
3	$\begin{tikzcd} 1,1 & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$	3		150.3	F
4	$\begin{tikzcd} 1,1 \arrow[r] & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$ $\begin{tikzcd} 1,2 \arrow[r] & 2,2 \arrow[d] \\ & 3,2 \arrow[ul, bend left] \end{tikzcd}$	3		342.4	T
5	$\begin{tikzcd} 1,1 \arrow[r] & 2,1 \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$ $\begin{tikzcd} 1,2 \arrow[r] & 2,2 \\ & 3,2 \arrow[ul, bend left] \end{tikzcd}$	3	 2, 2 3, 2 1, 1 3, 1 2, 1 3, 2 1, 2 3, 2 2, 2 1, 2 3, 1 2, 2 3, 1 1, 1 3, 1 2, 1 1, 1 3, 2 2, 1 3, 1 1, 2 3, 2	526.9	F
6	$\begin{tikzcd} 1,1 & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$ $\begin{tikzcd} 1,2 \arrow[r] & 2,2 \arrow[d] \\ & 3,2 \end{tikzcd}$	3		387.1	T

www.manaraa.com

Table 4.1 (Continued)

#	Pattern	n	Dependency graph	t(s)	R
7	$\begin{tikzcd} 1,1 \arrow[dr] & 1,2 \arrow[dl] \\ 2,1 \arrow[dr] & 2,2 \arrow[dl] \\ 3,1 \arrow[uu, bend left] & 3,2 \arrow[uu, bend right] \end{tikzcd}$	3		866.5	T
8	$\begin{tikzcd} 1,1 \arrow[r, "{3,1}"] & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$	3		119.4	T
9	$\begin{tikzcd} 1,1 \arrow[r, "{2,1,3,1}"] & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \end{tikzcd}$	3		174.8	T
10	$\begin{tikzcd} 1,1 & 2,1 \arrow[d] \\ & 3,1 \arrow[ul, bend left] \\ 1,2 \arrow[r, "{3,2}"] & 2,2 \arrow[d] \\ & 3,2 \arrow[ul, bend left] \end{tikzcd}$	3		949.3	T
11	$\begin{tikzcd} 1,1 \arrow[r] & 2,1 \arrow[l, bend left] \end{tikzcd}$	2		3.9	T

www.manaraa.com

```
1: while $x \leq 5$ do
2:     Do-something
3:     $x++$
4:     $y--$
5: end while
```

(a) Thread 1

```
1: while $y \leq 5$ do
2:     Do-something
3:     $y++$
4:     $x--$
5: end while
```

(b) Thread 2

Figure 4.1 Case study-1: Loops with inter-thread loop dependencies

4.1 Loops with *inter-thread loop dependencies*

Non-termination occurring due to circular *inter-thread loop dependencies* between simple loops in threads, leading to *count violations* falls under this category. Entries 1, 2, 3, 11 in Table.4.1 are examples of this category.

4.1.1 Case study-1

Let us consider the pattern described in the 1-st entry in Table.4.1. This pattern describes the *inter-thread loop dependencies* in Figure.4.1. There are two threads, with *inter-thread loop dependency* from $\langle 1, 1\rangle$ to $\langle 2, 1\rangle$, causing a *count violation* in $\langle 2, 1\rangle$ and vice versa. Hence, there exists a non-terminating execution path.

Following the evaluation of *base count*, when Thread 1, Thread 2 are executed with an appropriate assertion in $\langle 1, 1\rangle$, an assertion violation occurs, with cnt_{21} in the program trace. The *dependency graph* is updated accordingly. When Thread 2, Thread 1 are executed with an appropriate assertion in $\langle 2, 1\rangle$, an assertion violation occurs, with cnt_{12} in the the program trace. Following the updation of *dependency graph*, the non-termination check reveals a simple cycle as indicated. Test program for entry 11 in Table.4.1 is similar to the program described by Figure.4.1, except that we consider more complex loop constraints. In this case, our tool performs an identical sequence of computations to detect the likelihood of non-termination in 2.3s.

المنارة للاستشارات
www.manaraa.com

```
1: while $x \leq 5$ do
2:     Do-something
3:     $x++$
4: end while
```

(a) Thread 1

```
1: while $y \leq 5$ do
2:     Do-something
3:     $y++$
4:     $z--$
5: end while
```

(b) Thread 2

```
1: while $z \leq 5$ do
2:     Do-something
3:     $z++$
4:     $x--$
5: end while
```

(c) Thread 3

Figure 4.2 Case study-2: Loops with inter-thread loop dependencies

4.1.2 Case study-2

Consider the example Figure.4.2. This corresponds to the 3-rd entry in the Table.4.1. The example has three threads, with an *inter-thread loop dependency* from $\langle 2, 1\rangle$ to $\langle 3, 1\rangle$ causing a count violation on $\langle 3, 1\rangle$ and an *inter-thread loop dependency* from $\langle 3, 1\rangle$ to $\langle 1, 1\rangle$ causing a count violation on $\langle 1, 1\rangle$. There is no sub-group of thread-loop pairs with a circular or mutual *inter-thread loop dependency*.

$k = 2$: Following *base count* evaluation, EXPLORE executes Thread 1, Thread 2 with an appropriate assertion on $\langle 1, 1\rangle$. No assertion violation is observed. Thread 1, Thread 3 are executed with an appropriate assertion on $\langle 1, 1\rangle$. An assertion violation occurs with cnt_{31} in the program trace. The graph is updated accordingly. EXPLORE executes Thread 2, Thread 1, with an assertion on $\langle 2, 1\rangle$. No assertion violation is found. The observation is the same when Thread 2, Thread 3 are executed with an assertion on $\langle 2, 1\rangle$. Thread 3, Thread 1 are executed with an appropriate assertion on $\langle 3, 1\rangle$. No assertion violation is observed. Thread 3, Thread 2 are executed with an appropriate assertion on $\langle 3, 1\rangle$. Assertion violation with cnt_{21} in the program trace was observed. Since no non-termination was found yet, the next iteration proceeds by considering all the threads for testing.

$k = 3$: Considering all three threads, with an assertion on $\langle 1, 1\rangle$, yields an assertion violation with both cnt_{21}, cnt_{31} in the program trace. This is because, all three threads are selected for testing. The *dependency graph* is appropriately updated. However, considering all three threads with an assertion on $\langle 2, 1\rangle$ yields no assertion violation.

Lastly, similar to the first case, executing all three threads with an assertion on $\langle 3, 1\rangle$, yields an assertion violation with both cnt_{11}, cnt_{21} in the program trace. The *dependency*

المنارة للاستشارات
www.manaraa.com

```
1: while $x \le 5$ do
2:     Do-something
3:     $x++$
4:     $y--$
5:     while $a \le 5$ do
6:         $a++$
7:         $b--$
8:     end while
9: end while
```

(a) Thread 1

```
1: while $y \le 5$ do
2:     Do-something
3:     $y++$
4:     $z--$
5:     while $b \le 5$ do
6:         $b++$
7:         $c--$
8:     end while
9: end while
```

(b) Thread 2

```
1: while $z \le 5$ do
2:     Do-something
3:     $z++$
4:     $x--$
5:     while $c \le 5$ do
6:         $c++$
7:         $a--$
8:     end while
9: end while
```

(c) Thread 3

Figure 4.3 Case study-3: Nested loops with inter-thread loop dependencies

graph is updated. A non-termination check does not reveal a simple or complex cycle. Hence, no non-termination is reported.

4.2 Nested loops with *inter-thread loop dependencies*

Non-termination is caused due to *inter-thread loop dependencies* in nested loops. The dependencies can be between inner loops or outer loops or an inner loop and an outer loop. Examples include 4, 5, 6, 7 in Table.4.1.

4.2.1 Case study-3

Consider, the example in Figure.4.3. This corresponds to 4-th entry in the Table.4.1. The example has three threads, with circular *inter-thread loop dependencies* between all the three outer loops. There also exists a circular dependency between all the three inner loops. Hence, there exist two separate scenarios that may lead to non-termination.

$k = 2$, Thread 1: An assertion is placed on $\langle 1, 1\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Testing Thread 1, Thread 3 yields an assertion violation with cnt_{31}, cnt_{32} (both count variables appear because of the inherent dependency between the outer loop and inner loop in nested loops). Updation of graph is done. An assertion is placed on $\langle 1, 2\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Testing Thread 1, Thread 3 gives an assertion violation with cnt_{31}, cnt_{32}. Updation of graph is done.

www.manaraa.com

```
while $x \leq 5$ do
    Do-something
    $x++$
    while $a \leq 5$ do
        $a++$
        $b--$
    end while
end while
```

(a) Thread 1

```
while $y \leq 5$ do
    Do-something
    $y++$
    $z--$
    while $b \leq 5$ do
        $b++$
        $c--$
    end while
end while
```

(b) Thread 2

```
while $z \leq 5$ do
    Do-something
    $z++$
    $x--$
    while $c \leq 5$ do
        $c++$
    end while
end while
```

(c) Thread 3

Figure 4.4 Case study-4: Nested loops with inter-thread loop dependencies

$k = 2$, Thread 2: An assertion is placed on $\langle 2, 1\rangle$. Selecting Thread 1, Thread 2 for testing, yields an assertion with cnt_{11}, cnt_{12} in the program trace. Updation of graph is done. Selecting Thread 2, Thread 3 for testing, yields no assertion violation. An assertion is placed on $\langle 2, 2\rangle$. Selecting Thread 1, Thread 2 for testing, yields an assertion with cnt_{11}, cnt_{12} in the program trace. Updation of graph is done. Selecting Thread 2, Thread 3 for testing, yields no assertion violation.

$k = 2$, Thread 3: An assertion is placed on $\langle 3, 1\rangle$. Selecting Thread 1, Thread 3 for testing, yields no assertion violation. Selecting Thread 2, Thread 3 for testing, gives an assertion violation with cnt_{21}, cnt_{22} in the program trace. Updation of graph is done. At this point, a non-termination check reveals a simple cycle between the nodes $\langle 1, 1\rangle, \langle 2, 1\rangle, \langle 3, 1\rangle$ as indicated in the *dependency graph*. Hence, non-termination is reported.

4.2.2 Case study-4

Consider, the example Figure.4.4. This corresponds to entry 6 in the Table.4.1. In this example, we have three threads, each with a nested loop. Upon careful observation, we find, there exists no sub group of thread-loop pairs that have a circular *inter-thread loop dependency* amongst them.

$k = 2$, Thread 1: An assertion is placed on $\langle 1, 1\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Selecting Thread 1, Thread 3 for testing, gives an assertion violation with cnt_{31}, cnt_{32} in the program trace. Updation of graph is done. An assertion is placed on $\langle 1, 2\rangle$.

المنارة للاستشارات
www.manaraa.com

Selecting Thread 1, Thread 2 yields no assertion violation. Selecting Thread 1, Thread 3 for testing also yields no assertion violation.

$k = 2$, Thread 2: An assertion is placed on $\langle 2,1\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Selecting Thread 2, Thread 3, also yields no assertion violation. An assertion is now placed on $\langle 2,2\rangle$. Testing Thread 1, Thread 2 gives an assertion violation with cnt_{11}, cnt_{12} in the program trace. Updation of graph is done. Selecting Thread 2, Thread 3 for testing yields no assertion violation.

$k = 2$, Thread 3: An assertion is placed on $\langle 3,1\rangle$. Selecting Thread 1, Thread 3 yields no assertion violaton. Selection Thread 2, Thread 3, gives an assertion violation with cnt_{21}, cnt_{22} in the program trace. Updation of graph is done. At this point, a nontermination check returns true because of the simple cycle between $\langle 1,1\rangle, \langle 3,1\rangle, \langle 2,2\rangle$. This is a false positive, since in reality, recreating such an execution sequence does not lead to a non-termination.

When there exists an *inter-thread loop dependency* between a loop x in a thread, from a loop in a set of nested loops y, z in a different thread, causing a *count violation* in x, it is not feasible to deduce if the *count violation* occurs due to the inner loop or the outer loop. This is because, unless the inner loop is inside a conditional statement, the execution of the outer loop will inevitably cause the execution of the inner loop.

4.2.3 Case study-5

Consider, the example Figure.4.5. This corresponds to the entry 7 in the Table.4.1. The sample program consists of three threads, each having nested loops in them. Upon careful observation, we find, that there exist circular *inter-thread loop dependencies* in the sub groups $\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,1\rangle$ and $\langle 1,2\rangle, \langle 2,1\rangle, \langle 3,2\rangle$.

$k = 2$, Thread 1: An assertion is placed on $\langle 1,1\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Testing Thread 1, Thread 3 yields an assertion violation with cnt_{31}, cnt_{32} (both count variables appear because of the inherent dependency between the outer loop and inner loop in nested loops). Updation of graph is done. An assertion is placed on $\langle 1,2\rangle$. Testing Thread 1, Thread 2 yields no assertion violation. Testing Thread 1, Thread 3 gives an assertion violation with cnt_{31}, cnt_{32}. Updation of graph is done.

www.manaraa.com

```
1: while $x \leq 5$ do
2:     Do-something
3:     $x++$
4:     $b--$
5:     while $a \leq 5$ do
6:         $a++$
7:         $y--$
8:     end while
9: end while
```

(a) Thread 1

```
1: while $y \leq 5$ do
2:     Do-something
3:     $y++$
4:     $c--$
5:     while $b \leq 5$ do
6:         $b++$
7:         $z--$
8:     end while
9: end while
```

(b) Thread 2

```
1: while $z \leq 5$ do
2:     Do-something
3:     $z++$
4:     $x--$
5:     while $c \leq 5$ do
6:         $c++$
7:         $a--$
8:     end while
9: end while
```

(c) Thread 3

Figure 4.5 Case study-5: Nested loops with inter-thread loop dependencies

$k = 2$, Thread 2: An assertion is placed on $\langle 2, 1\rangle$. Selecting Thread 1, Thread 2 for testing, yields an assertion with cnt_{11}, cnt_{12} in the program trace. Updation of graph is done. Selecting Thread 2, Thread 3 for testing, yields no assertion violation. An assertion is placed on $\langle 2, 2\rangle$. Selecting Thread 1, Thread 2 for testing, yields an assertion with cnt_{11}, cnt_{12} in the program trace. Updation of graph is done. Selecting Thread 2, Thread 3 for testing, yields no assertion violation.

$k = 2$, Thread 3: An assertion is placed on $\langle 3, 1\rangle$. Selecting Thread 1, Thread 3 for testing, yields no assertion violation. Selecting Thread 2, Thread 3 for testing, gives an assertion violation with cnt_{21}, cnt_{22} in the program trace. Updation of graph is done. At this point, a non-termination check reveals a simple cycle between the nodes $\langle 1, 1\rangle, \langle 2, 1\rangle, \langle 3, 1\rangle$ as indicated in the *dependency graph*. Hence, non-termination is reported.

A careful observation reveals, that this is the exact same sequence of computations by EXPLORE that concluded the likelihood of non-termination in Case study-3. Again, we attribute this symmetrical behavior to be due to the inherent dependency between the inner and outer loop in nested loops. It is therefore, not feasible to isolate the cause of the dependency leading to a *count violation*.

www.manaraa.com

```
**while** $x \leq 5$ **do**
    Do-something
    $x++$
    **if** z $= 2$ **then**
        $y--$
    **end if**
**end while**
```

(a) Thread 1

```
**while** $y \leq 5$ **do**
    Do-something
    $y++$
    $z--$
**end while**
```

(b) Thread 2

```
**while** $z \leq 5$ **do**
    Do-something
    $z++$
    $x--$
**end while**
```

(c) Thread 3

Figure 4.6 Case study-6: Loops and conditionals with inter-thread loop dependencies

4.3 Loops and conditionals with *inter-thread loop dependencies*

In this case, non-termination is caused due to scenarios, where *inter-thread loop dependencies* causing *count violations* on a $\langle i, j\rangle$ occur due to two thread-loop pairs $\langle i', j'\rangle$, $\langle i'', j''\rangle$. Here, $i' \neq i''$. Examples include entries 8, 9, 10 in Table.4.1.

4.3.1 Case study-6

Consider the example in Figure.4.6. This corresponds to the entry 8 in Table.4.1. The program has three threads, with a circular *inter-thread loop dependency* between $\langle 1, 1\rangle$, $\langle 2, 1\rangle$, $\langle 3, 1\rangle$. Here, for $\langle 2, 1\rangle$ to encounter a *count violation*, dependencies are required from both $\langle 1, 1\rangle$, $\langle 3, 1\rangle$.

$k = 2$, Thread 1: An assertion violation is placed on $\langle 1, 1\rangle$. Selecting Thread 1, Thread 2 for testing yields no assertion violation. Selecting Thread 1, Thread 3 for testing, gives an assertion violation with cnt_{31} in the program trace. Updation of graph is done.

$k = 2$, Thread 2: An assertion violation is placed on $\langle 2, 1\rangle$. Selecting Thread 1, Thread 2 for testing yields no assertion violation. Similarly, selecting Thread 2, Thread 3 for testing, yields no assertion violation.

$k = 2$, Thread 3: An assertion violation is placed on $\langle 3, 1\rangle$. Selecting Thread 1, Thread 3 for testing yields no assertion violation. Selecting Thread 2, Thread 3 for testing gives an assertion violation with cnt_{21} in the program trace. Updation of graph is done. Since, no non-teminating execution sequence was found yet, EXPLORE repeats testing for k = 3.

$k = 3$: Selecting all the threads, placing an assertion on $\langle 1, 1\rangle$, gives an assertion violation with cnt_{21}, cnt_{31} in the program trace. Updation of graph is done. Selecting all the threads,

المنارة للاستشارات
www.manaraa.com

placing an assertion on $\langle 2, 1\rangle$, gives an assertion violation with cnt_{11}, cnt_{31} in the program trace. Updation of graph is done. A non-term check returns true, due to a complex cycle as shown in Table.4.1.

4.3.2 Experience with CONCREST

We have incorporated two types of techniques in our framework: con2colic testing technique as implemented in CONCREST tool (Farzan et al. (2013)) and simple random testing. As noted in Section.2.3, con2colic testing performs branch condition analysis and inter-thread interference analysis (race conditions) to systematically explore the possible executions of a concurrent program. This technique is efficient in verifying properties encoded as assertions. However, our experience reveals that increase in the number of branch points can result in significant overhead in terms of exploring interference scenarios. In particular, consider Figure.4.1. This program is a simple concurrent program with two threads, each having a loop of size 5. With CONCREST, our tool took 122s (as compared to 2.3s with random testing)to terminate, due to exhaustive exploration of an exorbitant number of irrelevant scenarios. Further, with larger sized loops, or programs with more number of threads, the technique fails to scale.

4.4 Discussion

Our approach successfully detects several different possible scenarios that could cause a non-terminating behavior in multi-threaded programs. In the presence of a non-terminating execution sequence, our tool is able to detect the non-termination in a reasonable amount of time. We observe, that the execution time of our tool is a function of the number of threads in the program of interest.

Further, the execution time is also dependent on the type of patterns that lead to non-termination. Patterns involving more conjunctive dependencies typically require more time. That is, when *inter-thread loop dependencies* on a thread-loop pair, require more than one thread-loop pair to cause a *count violation*, our experiments show, that it takes longer to discover such scenarios. This is evident from the entries 2, 8 in Table 4.1. Entry 2 shows a pattern in which, non-termination occurs due to simple *inter-thread loop dependencies*, a thread-

المنارة للاستشارات
www.manaraa.com

loop pair depends on another single thread-loop pair and hence it takes less time(100.6s). Entry 8 shows a pattern with a conjunctive dependency in which, for non-termination to occur, two thread-loop pairs affect a thread-loop pair to cause a *count violation*. Hence, in this case non-termination detection takes more time than the previous case(119.4s). This is because, testing to identify scenarios involving such complex dependencies typically requires exploring a larger number of thread schedules.

Our experiments with nested loops reveal that there may be false positives. This was due to the inherent dependency between the outer loop and the inner loop in nested loops. Inevitably, the execution of the outer loop causes the inner loop to be executed too. As a result of this, when a non-terminating execution sequence occurs due to an *inter-thread loop dependency* from the nested loops to a loop in a different thread, it is not possible to distinguish the source of the *count violation*.

www.manaraa.com

CHAPTER 5. CONCLUSION

5.1 Summary

Testing is primarily used to for verifying assertional properties. However, non-termination cannot be directly expressed as an assertional property. Also, testing techniques require that a program terminates, *how can we utilize testing techniques, that are more scalable than static analysis techniques, to detect non-termination in multi-threaded programs?* In this thesis, we have addressed this question. We have presented a novel, testing based technique, that detects non-termination in multi threaded programs due to unwanted race conditions.

Our technique involves the insertion of carefully selected assertions, violations of which, indicate *inter-thread loop dependencies.* We presented a specialized data structure, the *dependency graph* to encode these dependencies. We developed a reduction of non-termination in concurrent programs, to detecting cycles in this *dependency graph* and devised an algorithm for the same. To aid users in developing appropriate countermeasures to the detected non-termination, we display the scenario leading to the unbounded behavior along with the *dependency graph.* We realized our technique in a modular framework. We have validated the feasibility of our approach by experimental evaluation on systematic enumeration of tailored sample programs, that exhibit different types of execution scenarios leading to non-termination.

In this thesis, we focused on developing a generic, language agnostic methodology to detect non-termination instead of providing a solution that is specific to a programming language. By simply using a testing engine corresponding to the programming language of interest and by minor modifications to the instrumentation tool, our tool can be adapted to any programming language. The modularity of our framework, allows our technique to be used in conjunction with any testing technique. Hence, our technique can be further leveraged by pairing it with testing

المنارة للاستشارات

www.manaraa.com

techniques providing better soundness and completeness guarantees. Lastly, our technique being testing based, does not require explicit adjustments to consider the non-determinism introduced by memory models adopted.

5.2 Future Work

5.2.1 Investigation of methods to reduce false positives

Typical real world programs have many nested loops. There exists an inherent dependency between the parent loop and the child or nested loop, this causes the child loop to be executed every time the parent loop executes. This could again cause unforeseen *inter-thread loop dependencies* and *count violations* causing a false positive. A possible direction for future work, could be in investigating methods, that could reduce these false positives. For instance, when a non-termination is reported, testing could be used to recreate the scenario to indicate the presence of a non-termination, or in other words, whether the scenario causes a divergence in the program execution.

5.2.2 Improved identification of the scenario leading to non-termination

In the event, that our framework detects a non-terminating execution sequence, our framework reports a scenario, which consists of a sequence of thread - loop pairs along with their *inter-thread loop dependencies*, that led to the non-terminating program behavior. We are interested in identifying and reporting a more refined scenario, which is a sequence of program locations, local variable and global variable valuations that led to the non-terminating program behavior.

5.2.3 Automated generation of remedies for non-termination

Upon the identification of more refined scenarios leading to non-termination, a possible avenue for future work could be to facilitate the automatic generation of possible remedies to weed out the non-termination execution path, without compromising the advantages of concurrency.

www.manaraa.com

5.2.4 Development of guided testing strategies

Our tool currently adopts a naive testing strategy, where the course of testing is the same irrespective of the discovered dependencies. One avenue for future work could be to develop strategies that guide testing. That is, the *inter-thread loop dependencies* observed could be used to determine the most optimal course of testing.

المنارة للاستشارات

www.manaraa.com

BIBLIOGRAPHY

Atig, M. F., Bouajjani, A., Emmi, M., and Lal, A. (2012a). Detecting fair non-termination in multithreaded programs. In *Computer Aided Verification*, pages 210–226.

Atig, M. F., Bouajjani, A., Kumar, K. N., and Saivasan, P. (2012b). Linear-time model-checking for multithreaded programs under scope-bounding. In *International Symposium on Automated Technology for Verification and Analysis*, pages 152–166. Springer.

Bouajjani, A., Emmi, M., and Parlato, G. (2011). On sequentializing concurrent programs. In *International Static Analysis Symposium*, pages 129–145. Springer.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., and OHearn, P. (2014). Proving nontermination via safety. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–171. Springer.

Cook, B., Fuhs, C., Nimkar, K., and O'Hearn, P. (2014). Disproving termination with overapproximation. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 67–74. IEEE.

Cook, B., Kroening, D., Rümmer, P., and Wintersteiger, C. M. (2010). Ranking function synthesis for bit-vector relations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250. Springer.

Cook, B., Podelski, A., and Rybalchenko, A. (2005). Abstraction refinement for termination. In *International Static Analysis Symposium*, pages 87–101. Springer.

www.manaraa.com

Cook, B., Podelski, A., and Rybalchenko, A. (2006). Termination proofs for systems code. In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 41:6, pages 415–426. ACM.

Cook, B., Podelski, A., and Rybalchenko, A. (2007). Proving thread termination. In *ACM SIGPLAN Notices*, volume 42:6, pages 320–330. ACM.

Cook, B., See, A., and Zuleger, F. (2013). Ramsey vs. lexicographic termination proving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 47–61. Springer.

Emmi, M., Qadeer, S., and Rakamarić, Z. (2011). Delay-bounded scheduling. *ACM SIGPLAN Notices*, 46(1):411–422.

Farzan, A., Holzer, A., Razavi, N., and Veith, H. (2013). Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 37–47. ACM.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40:6, pages 213–223. ACM.

Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A., and Xu, R.-G. (2008). Proving non-termination. *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 43(1):147–158.

Inverso, O., Tomasco, E., Fischer, B., La Torre, S., and Parlato, G. (2014). Bounded model checking of multi-threaded c programs via lazy sequentialization. In *International Conference on Computer Aided Verification*, pages 585–602. Springer.

Jones, C. B. (1981). *Development methods for computer programs including a notion of interference*.

Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., and Rubio, A. (2014). Proving non-termination using max-smt. In *International Conference on Computer Aided Verification*, pages 779–796. Springer.

المنارة للاستشارات
www.manaraa.com

Morse, J., Cordeiro, L., Nicole, D., and Fischer, B. (2011). Context-bounded model checking of LTL properties for ANSI-C software. In *Software Engineering and Formal Methods*, pages 302–317. Springer.

Musuvathi, M. and Qadeer, S. (2008). Fair stateless model checking. In *ACM Programming Language Design and Implementation*, volume 43:6, pages 362–371. ACM.

Podelski, A. and Rybalchenko, A. (2004a). A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer.

Podelski, A. and Rybalchenko, A. (2004b). Transition invariants. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 32–41. IEEE.

Popeea, C. and Rybalchenko, A. (2012). Compositional termination proofs for multi-threaded programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 237–251. Springer.

Qadeer, S. and Rehof, J. (2005). Context-bounded model checking of concurrent software. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 93–107. Springer.

Rodrigues, R. E. (2013). Non-termination attacks based on integer overflows. *WTDSOFT 2013*, page 86.

Sen, K. and Agha, G. A. (2006a). Concolic testing of multithreaded programs and its application to testing security protocols.

Sen, K. and Agha, G. A. (2006b). Concolic testing of multithreaded programs and its application to testing security protocols.

Sen, K., Marinov, D., and Agha, G. (2005). CUTE: a concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272.

المنارة للاستشارات
www.manaraa.com

Velroyen, H. and Rümmer, P. (2008). Non-termination checking for imperative programs. In *International Conference on Tests and Proofs*, pages 154–170. Springer.

www.manaraa.com