JAIST Repository
https://dspace.jaist.ac.jp/

Title	An Environment for Testing Concurrent Programs Based on Rewrite-theory Specifications
Author(s)	Do, Minh Canh
Citation	
Issue Date	2019-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/17563
Rights	
Description	Supervisor:緒方　和博, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

An Environment for Testing Concurrent Programs Based on Rewrite-theory Specifications

1710458 Do Minh Canh

Supervisor	Kazuhiro Ogata
Main Examiner	Kazuhiro Ogata
Examiners	Kunihiko Hiraishi
	Ryuhei Uehara
	Toshiaki Aoki

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2019

Abstract

Today, software systems are used in various applications where failure is unacceptable. Among are airplanes, vehicles, utilities, telephones, banking & financial systems, commerce, logistics, appliances, houses, and securities. Very important software systems, such as operating systems and the Internet, that have been used as infrastructures are typically in the form of concurrent programs. Major concepts of programming languages that can be used to write concurrent programs emerged in the 1980s and since nearly then studies on testing concurrent programs have been conducted. Arora, et al. have comprehensively surveyed testing concurrent programs. They categorize it into eight classes: (a) reachability testing, (b) structural testing, (c) model-based testing, (d) mutation-based testing, (e) slicing-based testing, (f) formal method-based testing, (g) random testing, and (h) search-based testing. Model checking concurrent programs has been intensively studied, which may be classified into (c) and/or (f). Java Pathfinder (JPF) is such a model checker. Model checking is superior to the other testing techniques in that the former exhaustively checks all possible execution paths (or computations). However, model checking concurrent programs often encounters the notorious state explosion, which has not yet been conquered reasonably well. Therefore, testing techniques for concurrent programs must be worth studying so that they can be matured enough.

For a formal specification S and a concurrent program P, to test P based on S, we can basically take each of the following two approaches: (1) P is tested with test cases generated from S and (2) it is checked that state sequences generated from P can be accepted by S. The two approaches would be complementary to each other. Approach (1) checks if P implements the functionalities specified in S, while approach (2) checks if P never implements what is not specified in S. In terms of simulation, approach (1) checks if P can simulate S, while approach (2) checks if S can simulate P. Approaches (1) and (2) are often used in the program testing community and the refinement-based formal methods community, respectively, while both (1) and (2), namely bi-simulation, are often used in process calculi. This thesis proposes a new testing technique for concurrent programs based on approach (2) mainly because P is a concurrent program and then could produce many different executions due to the inherent nondeterminacy of P.

The proposed technique is basically a specification-based testing one. We suppose that S is specified in Maude and P is implemented in Java. Java Pathfinder (JPF) and Maude are then used to generate state sequences from P and to check if such state sequences are accepted by S, respectively. Even

without checking any property violations with JPF, JPF often encounters the notorious state space explosion while only generating state sequences because there could be a huge number of different states reachable from the initial states, there could be a huge number of different state sequences generated due to the inherent nondeterminacy of concurrent programs and a whole big heap mainly constitutes one state in a program under test by JPF. Thus, we propose a technique to parallelize state sequences generation from P and check if such state sequences are accepted by S in a stratified way. The state space reachable from each initial state is divided into multiple layers. Let us suppose that each layer l has depth d_l. Let d_0 be 0. For each layer l, state sequences $s^l_0, \ldots, s^l_{d_l}$ whose depth is d_l are generated from each state at depth $d_0 + \ldots + d_{l-1}$ from P. Each s^l_i is converted into the state representation $f(s^l_i)$ used in S, where f is a simulation relation candidate from P to S. We conjecture that if S is refined enough, f would be an identity function. There may be adjacent states $f(s^l_i)$ and $f(s^l_{i+1})$ such that $f(s^l_i)$ is the same as $f(s^l_{i+1})$. If so, one of them is deleted. We then have state sequences $f(s^l_0), \ldots, f(s^l_N)$, where the number $N + 1$ of the states in the sequence is usually much smaller than $d_l + 1$ because execution units in P are much finer than those in S. We check if each $f(s^l_0), \ldots, f(s^l_N)$ is accepted by S with Maude. The proposed technique is called a divide & conquer approach to testing concurrent programs, which could be naturally parallelized. We have implemented a tool supporting the proposed technique in Java. Some experiments demonstrate that the proposed technique mitigates the state space explosion instances from which otherwise only one JPF instance cannot suffer.

Keywords: testing concurrent programs; specification-based testing; Java Pathfinder(JPF); divide & conquer; Maude; meta-programming; simulation relations.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor, Professor Kazuhiro Ogata who has supported and kindly guided throughout my study period at Japan Advanced Institute of Science and Technology. He has inspired me to become a good scientific researcher, as well as give me invaluable knowledge of how to deal with problems and critical thinking. Besides, he provided a high research environment and make me feel freedom and high motivative for unlimited creativeness in research.

I also wish to express my special thanks to my second supervisor, Professor Kunihiko Hiraishi as well as my supervisor for minor research, Professor Ryuhei Uehara. They have given advice and comments for my research in which helps me a lot to improve my work.

I would like to express my appreciation to my lab mates. Thank all of you for sharing wonderful moments, interesting ideas, not only in research but daily life. It is an unforgettable memory in my life. I also want to say thank to JAIST Football Club where I have enjoyed playing football with talent players after every tough research time. After every football match, I feel refresh and more high energy to keep going on research.

Finally, I would like to thank my wife Nguyen Thi Thanh as well as whole my family who have been hugely throughout supporting with endless love without any conditions. Without their support, it would be impossible for me to complete this work.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Related Work 3
1.4 Contributions 8
1.5 Thesis Structure 8

2 Preliminaries **10**
2.1 State Machine 10
2.2 Simulation Relations 10
2.3 Meta Programming in Maude 12
2.4 Concurrent Programming in Java 16
2.4.1 Threads 17
2.4.2 Synchronization 18

3 Specification-based Testing with Simulation Relations **22**
3.1 State Sequence Generation from Concurrent Programs 23
3.1.1 Java Pathfinder (JPF) 23
3.1.2 Generating State Sequences by JPF 24
3.2 Checking a finite semi-computation 30
3.3 Summary 32

4 Divide & Conquer Approach to Testing Concurrent Programs **34**
4.1 A Divide & Conquer Approach to Generating State Sequences 34
4.2 Environment Architecture 37
4.3 Summary 42

5 Case Studies **44**
5.1 Simple Communication Protocol (SCP) 44
5.2 Alternating Bit Protocol (ABP) 48
5.3 Summary 52

6 Conclusions and Future Work **53**
6.1 Conclusions 53
6.2 Future Work 54

Bibliography **56**

Publications **59**

This thesis was prepared according to the curriculum for the Collaborative Education Program organized by Japan Advanced Institute of Science and Technology and VNU University of Engineering and Technology, Vietnam National University.

List of Figures

1.1 Specification-based concurrent program testing with a simulation relation . . . 3

2.1 A simulation relation from M_C to M_A . . . 11
2.2 Thread life cycle in Java . . . 17

3.1 JPF Core . . . 24
3.2 JPF Listeners . . . 25
3.3 A way to generate state sequences with JPF . . . 29

4.1 A divide & conquer approach . . . 35
4.2 The architecture of a tool supporting the proposed technique . 37

5.1 A state of SCP . . . 45
5.2 Time taken when the length of each state sequence is fixed (100) and the number of state sequences is changed (100, 1000, 10000, 50000, 100000, 500000 & 1000000) . . . 46
5.3 Time taken when the number of state sequences is fixed (1) and the length of the state sequence is changed (100, 1000, 10000, 50000, 100000, 250000 & 500000) . . . 47
5.4 A state of ABP . . . 49

List of Tables

5.1 Experimental data 50

Chapter 1

Introduction

Many things are controlled by computer software systems nowadays. Among them are airplanes, vehicles, utilities, telephones, banking & financial systems, commerce, logistics, appliances, houses, and securities. Almost all things will be controlled by computer software systems in the near future. Very important software systems, such as operating systems and the Internet, that have been used as infrastructures are typically in the form of concurrent programs. Major concepts of programming languages that can be used to write concurrent programs emerged in the 1980s and since nearly then studies on testing concurrent programs have been conducted. Arora, et al. have comprehensively surveyed testing concurrent programs well because of their inherent nature of non-determinism, which often leads to overlooking subtle flaws lurking in the concurrent programs and/or the notorious state explosion [1]. Therefore, testing techniques for concurrent programs must be worth studying so that they can be matured enough.

1.1 Motivation

Traditional software testing techniques [8] for sequential programs are not adequate for concurrent programs because the inherent nature of non-determinism, being unable to detect subtle flaws lurking in concurrent programs. Model checking is superior to the other testing techniques in that the former exhaustively checks all possible execution paths. However, model checking concurrent programs often encounters the notorious state explosion, which has not yet been conquered reasonably well. Some advanced techniques have proposed by many researchers such as dynamic symbolic execution (DSE) [2], dynamic partial order reduction(DPOR) [3]. Although these approaches have greatly increased the size of the complex systems that can be verified,

but many realistic systems are still too large to handle. JPF is one of the most mature software model checkers [4], [5]. It only can detect subtle flaws lurking in concurrent programs, provided that the flaws are located at shallow positions. On other hands, traditional model checkers are used to testing systems in sequential style but not parallelization. Recently, some model checking techniques with parallelization have studied [6] [7] and the result is impressive. So it is really significant to make a scalable technique to testing concurrent programs so as to detect subtle flaws located at deep positions in large concurrent programs due to many important software systems are in the form of concurrent programs.

1.2 Problem Statement

Java is the most popular programming languages, is used by many researchers as well as supports rich features to deal with concurrent programs. Besides, JPF is one of the most mature software model checkers, is written in Java. On other hands, Maude is a high-performance specification language, is equipped reflective that helps us flexibly building meta applications by using meta programming. So in this research, we focus on testing Java concurrent programs by using JPF and Maude facilities. The problem can be stated as follows:

Input: Given a specification S in Maude, a concurrent program P in Java that is implemented from the specification S and a simulation relation r from P to S.

Output: Detecting any subtle flaw lurking in the concurrent program P based on specification S.

Solution Overview: Fig. 1.1 shows an overview of the flow of our method

1. state sequences $s_0, s_1, \ldots, s_n$ are generated from P by using JPF.
2. state sequences $s''_0, s''_1, \ldots, s''_m$ for S are obtained by converting $s_0, s_1, \ldots, s_n$ with r and
3. it is checked that S can accept $s''_0, s''_1, \ldots, s''_m$ by using meta programming in Maude.

Difficulties: JPF usually is used to model checking Java concurrent programs. In this research, we use JPF to generate state sequences from Java concurrent programs. It makes challenges to analyze and extract state information from the designated Heap memory in JPF. Especially, straightforward use of JPF immediately encounters the state explosion. Even when

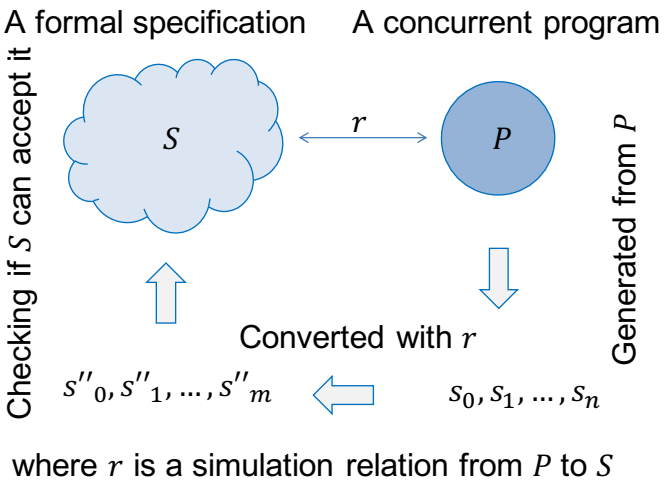


Figure 1.1: Specification-based concurrent program testing with a simulation relation

JPF is used to generate state sequences from Java concurrent programs, we soon encounter the state explosion. It makes more challenges to deal with the state explosion that is the main challenge sharing by model checking applications. That is why we come up with a divide & conquer approach to parallelize state sequence generation, making it possible to generate deeper or longer state sequences that we do without the use of the approach. A scalable technique to testing concurrent programs such that mitigates the state explosion well enough is needed. Besides, the behavior of real current programs is different a little bit to the behavior of specifications, so making the exact same behavior is another challenge. Moreover, we need to combine all parts of our environment to make it work correctly under control is also a challenge.

1.3 Related Work

This section surveys previous work related to specification-based testing. As we know, the testing of concurrent programs is very complex due to coherent non-deterministic threads interleaving that can trigger concurrency errors. Many researchers have been conducting to figure out an effective method to deal with testing concurrent programs. Recently, the automated test generation has emerged to become popular and most relevant to this thesis, many techniques have been proposed that includes the random test generation as

well as non-random techniques.

Random Testing

Exploring all possible non-deterministic threads interleaving that are time-consuming and very expensive. Especially, it seems unfeasible in reality for large programs due to the state space explosion. Random-bases testing is one of the approaches to address this problem. It does not explore all threads interleaving instead of using a randomized search strategy. It would pick up a random thread to execute at every execution point or scheduling point where threads may perform in different behaviors. This approach is simple, less expensive, get a better result when running again and again. But facing another problem that often misses concurrency bugs or does not suffice due to randomized selection. To address this problem, some effective random testings are proposed to improve reliability or confidence in the tested software. One of them is adaptive random testing (ART), an improved random testing method [8]. They used metamorphic testing (MT) and metamorphic distance into ART, which is called metamorphic distance based ART (MD-ART). Metamorphic testing (MT) provides an effective way for mitigating test oracle problem with metamorphic relation (MRs) and additional test cases. While ART is basically a random selection technique, this improved significantly random testing method compared to the traditional random testing method. RM-MT is an existing traditional random testing method to generate original test cases. Their experiments pointed out that MD-ART performs better than RT-MT in this work.

Another technique is CovCon [9] that is a coverage-guided approach to generating concurrent tests that can detect arbitrary kinds of concurrency bugs due to independent of any particular bug pattern. They follow the philosophy of coverage-based approaches to guide random-baed test generation toward uncovered threads interleaving. The new idea is to use concurrent method pairs that represent the set of interleaving events. They measure how often method pairs have been executed concurrently and use that coverage information to generate tests on method pairs that have not been covered or have been covered less frequently than others. Thereby, it makes it less random and explores misbehavior from programs. They have reported that the use of CovCon to 18 thread-safe Java classes and it detects 17 concurrency bugs of them while requiring less time. In other words, this approach may speed up in some cases to find concurrency bugs.

Testing concurrent programs are difficult as well as extremely time-consuming due to model checkers need to exhaustively traverse all possible behaviors of concurrent programs in their large state spaces. A big gap between

completion and deployment of programs is regarded as another problem. Metzler, et al. [10] proposed a novel iterative relaxed scheduling (IRS) approach to verification of concurrent programs that reduces such that time. IRS introduces a set of admissible schedules and a suitable execution environment. It iteratively verifies each trace that is generated by the scheduling. As soon as a single trace is verified, it will be added to the set of admissible schedules. While continuously verify individual schedules or sets of schedules, IRS execution environment does not need to wait until the program is fully verified. It may execute selected schedules from the set of admissible schedules that make the program be able to be safely used.

Systematic Testing

The automated test generation techniques that do not use randomness, model checking is a superior method than others due to it systematically explores all thread interleaving schedules to give a promise in the correctness of the system. But the most challenge of model checking is the state explosion problem because of very large state space. We use model checking JPF in our tool, so we mainly discuss model checking here. Symbolic execution and Partial Order Reduction are old techniques, and very famous to address state space explosion problem in model checking.

Symbolic execution is used in many purposes, one of them is to generate test inputs. The main idea behind symbolic execution is to use symbolic values, instead of program variables. Inputs are a program, then the outputs computed by a program are expressed as a function of the symbolic inputs. The state of symbolically executed programs consists of the values of programs variables (symbolic), a path condition (PC) and a program counter. The path condition is a boolean formula over the symbolic inputs, it accumulates constraints which the inputs must satisfy in order for execution to follow the particular associated path. The program counter defines the next statement to be executed. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

By using symbolic execution and a model checker, namely JPF, they have generated automatically test suites at the black-box level and the white-box level [11]. This work says that the efficient test input generation can be done for code manipulating complex data, especially the red-black tree instead of simple data types such as integers. They have used an algorithm for generalizing traditional symbolic execution to handle dynamically allocated structures, primitive data and concurrency. Then build up a symbolic execution framework on top of JPF model checking tool. From an original

program translates to another source that adds nondeterminism and support for manipulating formulas that represent path conditions. JPF checks the new program using its state exploration techniques. In other words, exploring the symbolic execution tree of the program. A state includes a heap configuration, a path condition and thread scheduling. Whenever a path condition is updated, it is checked satisfaction under some conditions. If the path condition is unsatisfiable, JPF backtracks. If a path is satisfiable, concrete test inputs are generated that will obligate execution to follow such a path.

Based on symbolic execution, another technique has proposed, namely dynamic symbolic execution (DSE) [2]. DSE generates automatically test input by executing a program with concrete and symbolic values simultaneously. Executing all possible program path is impossible due to exponential or infinite number of paths. This is the main challenge of DSE problem and then it presents a method to increase the coverage achieved by the presence of input-data dependent loops and loop dependent branches. They combine DSE with abstract interpretation to find indirect control dependencies, including loop and branch indirect dependencies. In other words, using abstract interpretation to solve 2 things in this case: 1) to calculate in advance how many iterations are needed to enter that subsequent branch and 2) invariant generation that relates program inputs to other program variables in the conditional statement of the branch. Compared to dynamic symbolic execution without abstract interpretation, this approach is better coverage and requires less time.

Maximal causality reduction (MCR) [12] is a new technique for stateless model checking to efficiently reduce state spaces. MCR takes a trace as input and generates a set of new interleaving events. It exploits the events of thread execution in a trace to drive new execution that reaches a new distinct state. Thereby, MCR minimizes redundant interleaving events better than classical techniques. By using existing MCR with dynamic symbolic execution, a new technique called Maximal Path Causality is proposed to explore both the input space and the schedule space at the same time [13].

Partial order reduction (POR) is a general theory to mitigate the state space explosion by exploiting the independence of concurrently executed events. If two events p and t are independent of each other, it is sufficient to analyze only one of them. But if events p and t are in a race, we must take into account both executions p and t. Early POR algorithms depend on static over approximations to detect possible future conflicts. The dynamic partial order reduction (DPOR) algorithm as a state of the art due to it does not need to look at the future [3]. The main idea of DPOR is to dynamically construct two types of sets at each scheduling point: 1) the *sleep*

set that contains processes that should not be selected due to it is explored to be redundant and 2) the *backtrack set* that contains the processes that have not proven independent with previous explored steps. In this paper, they introduced a new notion of conditional independence, which ensures the commutativity of the considered events p and t under certain conditions that can be evaluated in the explored state with a DPOR algorithm to mitigate the state space explosion problem [14].

Scalable Testing

Although the symbolic execution and the partial order reduction have greatly increased the size of the system that can be verified, many realistic systems are still too large to handle. The use of parallelism is used to attack this problem by utilizing more hardware resources to solve the model checking problem. In 2013, Barnat, et al. have proposed DiVinE 3.0, an Explicit-State Model Checker for Multithreaded C & C++ Programs [6]. 5 years later, they have proposed a Parallel Model Checking Algorithms for Linear-Time Temporal Logic [7].

DiVinE 3.0 is an explicit-state model checker for multithreaded C & C++ programs. In version 3.0, this tool can directly verify C/C++ programs based on newly developed LLVM bitcode interpreter. At the same time, DiVinE 3.0 used partial order reduction and path compression techniques to reduce the state space, combined with parallel and distributed-memory processing. Thereby, DinVinE3.0 may verify large systems, when compared to sequential model checkers as well as traditional techniques.

LTL model checking heavily depend on the search strategy such as Depth-First Search (DFS) and Breadth-First Search (BFS), it is hard to parallel. In this paper [7], Barnat, et al. applied parallelism to both two search strategies and give different characteristics. DFS depends on heuristics for good parallelization, but exhibit a low complexity and good on-the-fly behavior. And BFS offers good parallel scalability and support distributed parallelism for both two search strategies.

Another one for scalable testing, FlyMC [15] proposed a fast and scalable testing approach for datacenter/cloud systems such as Cassandra, Hadoop, Spark, and ZooKeeper. The approach is able to overcome the state space explosion problem with complex interleaving of messages and faults. Three algorithms in FlyMC have proposed: 1) state symmetry, 2) event independence and 3) parallel flips that make their approach speed up on average 16x (up to 78x) faster than other solutions. All were done systematically without random walks or manual scheduling checking points.

Our approach has used the model checker JPF for state sequences gen-

eration, we focus on a scalable testing to mitigate the state space explosion problem. Those techniques seem related to our proposed technique, especially scalable testing section. Some of them could be incorporated into our tool.

1.4 Contributions

The main challenge in model checking for testing concurrent programs is dealing with the state explosion problem due to their inherent nature of non-determinism. Our main contributions consist of:

- Firstly, we propose a new testing technique for concurrent programs. The technique is basically a specification-based testing one. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S.
- Secondly, we propose a technique to parallelize state sequences generation from P and check if such state sequences are accepted by S in a stratified way. Some experiments demonstrate that the proposed technique mitigates the state space explosion instances from which otherwise only one JPF instance cannot suffer.
- Lastly, we combine all methods above to develop a tool to completely testing Java concurrent programs.

1.5 Thesis Structure

The thesis is organized into six chapters. Chapter 1 is the introduction, the main content of the next five chapters are summarized as follows:

- **Chapter 2** presents some background knowledge of State Machine, Simulation Relations, Meta Programming in Maude as well as Concurrent Programming in Java that are applied in the scope of this thesis.
- **Chapter 3** explains how Specification-based Testing with Simulation Relations work in-depth. It will show how to generate state sequences from concurrent programs and then check such state sequences with a specification, namely checking a finite semi-computation.
- **Chapter 4** presents mainly our Divide & Conquer Approach to Testing Concurrent Programs from generating state sequences to fully the architecture of a tool supporting the proposed technique.

- **Chapter 5** shows the practical experiments result over 2 case studies ABP and SCP protocols. Both of them are communication protocol.
- **Chapter 6** summarizes the main contributions of the thesis and the advantages as well as remaining drawbacks. From that, some future works are mentioned to improve and extend our proposed technique.

Chapter 2

Preliminaries

This chapter presents some fundamental existing techniques used in our method. For specifications, a state machine is used to specify systems as finite state machines. Simulation relations is to find out a simulation relation between programs to specifications. Developers are mandatory to have a profound understanding of programs and specification as well. For concurrent programs, we need to use implement by using concurrent programming mechanism in Java such as threads, synchronizations. Lastly, the meta programming in Maude is to check if state sequences are generated from program P whether are accepted by specification S.

2.1 State Machine

A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states, the set $I \subseteq S$ of initial states and a binary relation $T \subseteq S \times S$ over states. $(s, s') \in T$ is called a state transition and may be written as $s \rightarrow_M s'$. Let $\rightarrow_M^*$ be the reflexive and transitive closure of $\rightarrow_M$. The set $R_M \subseteq S$ of reachable states w.r.t. M is inductively defined as follows: (1) for each $s \in I$, $s \in R$ and (2) if $s \in R$ and $(s, s') \in T$, then $s' \in R$. A state predicate p is called invariant w.r.t. M iff $p(s)$ holds for all $s \in R_M$. A finite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$ of states is called a finite semi-computation of M if $s_0 \in I$ and $s_i \rightarrow_M^* s_{i+1}$ for each $i = 0, \ldots, n-1$. If that is the case, it is called that M can accept $s_0, \ldots, s_i, s_{i+1}, \ldots, s_n$.

2.2 Simulation Relations

Given two state machines M_C and M_A, a relation r over R_C and R_A is called a simulation relation from M_C to M_A if r satisfies the following two

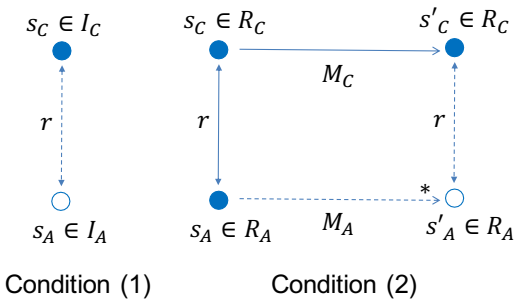


Figure 2.1: A simulation relation from M_C to M_A

conditions: (1) for each $s_C \in I_C$, there exists $s_A \in I_A$ such that $r(s_C, s_A)$ and (2) for each $s_C, s'_C \in R_C$ and $s_A \in R_A$ such that $r(s_C, s_A)$ and $s_C \rightarrow_{M_C} s'_C$, there exists $s'_A \in R_A$ such that $r(s'_C, s'_A)$ and $s_A \rightarrow^*_{M_A} s'_A$ [15] (see Fig. 2.1). If that is the case, we may write that M_A simulates M_C with r. There is a theorem on simulation relations from M_C to M_A and invariants w.r.t M_C and M_A: for any state machines M_C and M_A such that there exists a simulation relation r from M_C to M_A, any state predicates p_C for M_C and p_A for M_A such that $p_A(s_A) \Rightarrow p_C(s_C)$ for any reachable states $s_A \in R_{M_A}$ and $s_C \in R_{M_C}$ with $r(s_C, s_A)$, if $p_A(s_A)$ holds for all $s_A \in R_{M_A}$, then $p_C(s_C)$ holds for all $s_C \in R_{M_C}$ [15]. The theorem makes it possible to verify that p_C is invariant w.r.t. M_C by proving that p_A is invariant w.r.t. M_A, M_A simulates M_C with r and $p_A(s_A)$ implies $p_C(s_C)$ for all $s_A \in R_{M_A}$ and $s_C \in R_{M_C}$ with $r(s_C, s_A)$.

States are expressed as braced soups of observable components, where soups are associative-commutative collections and observable components are name-value pairs in this paper. The state that consists of observable components oc_1, oc_2 and oc_3 is expressed as $\{oc_1\ oc_2\ oc_3\}$, which equals $\{oc_3\ oc_1\ oc_2\}$ and some others because of associativity and commutativity. We use Maude [16], a rewriting logic-based computer language, as a specification language because Maude makes it possible to use associative-commutative collections. State transitions are specified in Maude rewrite rules.

Let us consider as an example a mutual exclusion protocol (the test&set protocol) in which the atomic instruction test&set is used. The protocol written in an Algol-like pseudo-code is as follows:

Loop : "RemainderSection(RS)"
  rs : **repeat while** test&set($lock$) = true;
    "CriticalSection(CS)"
  cs : $lock$:= false;

lock is a Boolean variable shared by all processes (or threads) participating in the protocol. test&set(*lock*) does the following atomically: it sets *lock* false and returns the old value stored in *lock*. Each process is located at either rs (remainder section) or cs (critical section). Initially, each process is located at rs and *lock* is false. When a process is located at rs, it does something (which is abstracted away in the pseudo-code) that never requires any shared resources; if it wants to use some shared resources that must be used in the critical section, then it performs the **repeat while** loop. It waits there while test&set(*lock*) returns true. When test&set(*lock*) returns false, the process is allowed to enter the critical section. The process then does something (which is also abstracted away in the pseudo-code) that requires to use some shared resources in the critical section. When the process finishes its task in the critical section, it leaves there, sets *lock* false and goes back to the remainder section.

When there are three processes p1, p2 and p3, each state of the protocol is formalized as a term $\{(\text{lock} : b)\ (\text{pc[p1]} : l_1)\ (\text{pc[p2]} : l_2)\ (\text{pc[p3]} : l_3)\}$, where b is a Boolean value and each l_i is either rs or cs. Initially, b is false and each l_i is rs. The state transitions are formalized as two rewrite rules. One rewrite rule says that if b is false and l_i is rs, then b becomes true, l_i becomes cs and any other l_j (such that $j \neq i$) does not change. The other rewrite rule says that if l_i is cs, then b becomes false, l_i becomes rs and any other l_j (such that $j \neq i$) does not change. The two rules are specified in Maude as follows:

```
rl [enter] : {(lock: false) (pc[I]: rs) OCs}
  => {(lock: true) (pc[I]: cs) OCs} .
rl [leave] : {(lock: B) (pc[I]: cs) OCs}
  => {(lock: false) (pc[I]: rs) OCs} .
```

where `enter` and `leave` are the labels (or names) given to the two rewrite rules, `I` is a Maude variable of process IDs, `B` is a Maude variable of Boolean values and `OCs` is a Maude variable of observable component soups. `OCs` represents the remaining part (the other processes but process `I`) of the system. Both rules never change `OCs`. Let $S_{\text{t\&s}}$ refer to the specification of the test&set protocol in Maude.

2.3 Meta Programming in Maude

Maude is a high-level language and a high-performance system [16]. It supports both equational and rewriting logic computation. Rewriting logic is logic of concurrent change, therefore a concurrent system can be specified in

Maude. Moreover, rewriting logic is reflective. This makes possible many advanced meta programming and meta language applications.

A meta program is a program that takes programs as inputs and performs some useful computations such as it may transform one program into another. Or it may analyze such a program with respect to some properties, or perform other useful programs independent computation. Obviously, it is very useful and very powerful. In Maude, meta programming has a logical reflective semantics. Essentially, It has 2 possible term representation, one is the object level and another one is the meta level. The object level representation can correctly simulate the relevant meta level representation and vice versa. We can easily write Maude meta programs by importing $META-LEVEL$ module into a module that defines such key functionality of meta programs as functions that have Module as one of their arguments. In $META-LEVEL$ module, this includes the modules $META-MODULE$ and $META-TERM$. The following describes shortly overview about three modules.

- in the module $META-TERM$, terms are meta represented as elements of a data type $Term$ of terms.
- in the module $META-MODULE$, modules are meta represented as terms in a data type $Module$ of modules.
- in the module $META-LEVEL$
  - operations upModule, upTerm, downTerm and others allow moving between reflection levels.
  - functions metaReduce, metaApply, metaXapply, metaRewrite, metaFrewrite, metaMatch and metaXmatch are called descent functions.
  - the process of searching for a term satisfying some conditions starting from an initial term is built-in functions metaSearch and metaSearchPath.

In this section, we present three meta functions examples that can be helpful as a basic guide to start with meta programming tasks.

First of all, we tackle to meta representation Modules. Any meta functions require a Module as one of their arguments. But such a Module must be represented in meta level, upModule function is a built-in function in META-LEVEL module that may transform a module in object level to meta level representation. The upModule function is explicitly declared as follows:

```
op upModule : Qid Bool ~> Module [special (...)]
```

where

- Qid is the name of a module.
- Bool is a boolean value. If it is called with true as its second argument. All modules, its equations are shown. Otherwise, only the current module is shown.

Example, we specify a module PNAT in Maude as in the listing 2.1. To get meta representation of PNAT module we will feed a command to Maude interpreter as follows:

```
red in META-LEVEL : upModule('PNAT, false) .
```

The command says that we want to get the meta representation of PNAT module up to the current module level.

Listing 2.1: Module PNAT in object level representation

```
fmod PNAT is
  sorts Zero PNat .
  subsorts Zero < PNat .
  op 0 : -> Zero [ctor] .
  op s : PNat -> PNat [ctor] .
  op _+_ : PNat PNat -> PNat [comm] .
  vars N M : PNat .
  eq 0 + N = N .
  eq s (N) + M = s (N + M) .
endfm
```

The listing 2.2 shows the corresponding meta level representation of PNAT in the object level when you use that such command. Although, two kinds of representation are different, but it is transformed and represented consistently.

Listing 2.2: Module PNAT in meta level representation

```
fmod 'PNAT is
  including 'BOOL .
  sorts 'PNat ; 'Zero .
  subsort 'Zero < 'PNat .
  op '0 : nil -> 'Zero [ctor] .
  op '_+_ : 'PNat 'PNat -> 'PNat [comm] .
  op 's : 'PNat -> 'PNat [ctor] .
  none
  eq '_+_['0.Zero,'N:PNat] = 'N:PNat [none] .
  eq '_+_['M:PNat,'s['N:PNat]] = 's['_+_['N:PNat,'M:PNat]] [
     none] .
endfm
```

Secondly, we introduce about meta representation Terms. upTerm and downTerm are frequently used in meta programming. They are declared explicitly as follows:

```
op upTerm : Universal -> Term [poly(1) special (...)] .
op downTerm : Term Universal -> Universal [poly (2 0) special
(...)] .
```

upTerm function is used to transform a term in object level to its meta level representation and downTerm function to convert a meta level representation to its object level representation. It means we can switch between in the reflective way. Listing 2.3 shows how to use upTerm function, it converts s(s(s(0))) term in object level to meta level. Using downTerm function if you want to reconvert to object level representation.

Listing 2.3: Converting a term in object level to its meta representation by upTerm function

```
Maude> red in META-LEVEL : upTerm(s(s(s(0)))) .
reduce in META-LEVEL : upTerm(3) .
rewrites: 1 in 0ms cpu (0ms real) (100000 rewrites/second)
result GroundTerm: 's_^3['0.Zero]
```

Lastly, searching is the most important functionality that allows us to check whether or not programs satisfy some properties. Maude supports search command. It allows us to explore the reachable state space in different ways. Its syntax is in the form of the following general scheme.

```
search [ n, m ] in <ModId> : <Term-1> <SearchArrow> <Term-2>
  such that <Condition>
```

where

- n is an optional argument providing a bound on the number of the desired solution.
- m is another optional argument stating the maximum depth of the search.
- the module $<$ ModId $>$ where the search takes place can be omitted.
- $<$ Term1 $>$ is the starting term.
- $<$ Term2 $>$ is the pattern that has to be reached.
- $<$ SearchArrow $>$ is an arrow indicating the form of the rewriting proof from $<$ Term $-$ 1 $>$ until $<$ Term $-$ 2 $>$.

  - => 1 means a rewriting proof consisting of exactly one step.
  - => + means a rewriting proof consisting of one or more steps.
  - => $*$ means a proof consisting of none, one, or more steps, and
  - =>! indicates that only canonical final states are allowed, that is, states that cannot be further rewritten.

- < Condition > states an optional property that has to be satisfied by the reached state.

For example, the search command below checks if s2 is reachable from s1 by depth 2 in given module ABP. In this case, we do not use any condition.

```
search [1,2] in ABP : s1 =>* s2 .
```

At meta level, the searching strategy used by metaSearch coincides with that of the object level search command in Maude. With the above example, we can use metaSearch function instead of search command as follows:

```
metaSearch(upModule('ABP, false), upTerm(s1), upTerm(s2), nil,
'*, 2, 0)
```

where

- upModule converts a module to its meta-representation.
- upTerm converts a term to its meta-representation.
- nil for a condition.
- 2 for depth.
- 0 for the solution number.

2.4 Concurrent Programming in Java

Java is the most popular programming languages, has been used by many researchers and then been matured enough. Besides, it supports rich features to deal with concurrent programs. So we chose Java as a programming language in our environment. In our document, concurrent programming is in the form of multithreading in Java that is a process of executing multiple threads simultaneously.

2.4.1 Threads

A thread is a lightweight process, the smallest unit of processing that provides an execution environment. Threads exist within a process, every process has at least one. Each process has its own memory space, but threads share the process's resources. This makes for efficient, but potentially communication. A thread goes through various stages in its life cycle. For example, a thread is created, started, runs, and then terminated. Fig. 2.2 shows the complete life cycle of a thread.

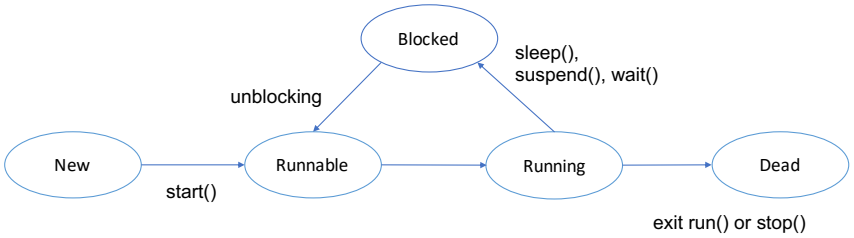


Figure 2.2: Thread life cycle in Java

Following are the stages of the life cycle.

- **new** - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
- **Runnable** - The thread is a runnable state after the invocation of start() method, but the thread scheduler has not selected it to be the running thread.
- **Running** - The thread is running state if the thread scheduler has selected it.
- **Non-Runnable(Blocked)** - This is the state when the thread still alive, but is currently not eligible to run.
- **Dead** - A thread is in terminated or dead state when its run() method exits.

To create an instance of Thread must define the code that will run in that thread. There are two ways to do so. The first way is to provide a class that implements Runnable interface as the listing 2.4 .

Listing 2.4: Defining a thread by implementing the Runnable interface

```
public class HelloRunnable implements Runnable {

  public void run() {
    System.out.println("Hello from a thread!");
  }

  public static void main(String args[]) {
    (new Thread(new HelloRunnable())).start();
  }

}
```

The HelloRunnable class must implement the *run* method from the Runnable interface where comprises the code executed in the thread. To create a new thread from the HelloRunnable class, we need to pass a Runnable object to Thread constructor and then call the *start* method. The thread will execute the code in the run method.

The second way is to define a subclass of *Thread* as listing 2.5. Basically, the Thread class itself implements the Runnable interface, though its run method does nothing. So the subclass Thread needs to provide its implementation of the run method as the HelloThread class in the listing 2.5. A bit different when you create a new thread in this way, just create a new object from the subclass Thread and then call the start method.

Listing 2.5: Defining a subclass of Thread

```
public class HelloThread extends Thread {

  public void run() {
    System.out.println("Hello from a thread!");
  }

  public static void main(String args[]) {
    (new HelloThread()).start();
  }

}
```

2.4.2 Synchronization

Multithreading is extremely efficient to multiprocessing by sharing access to the same resources, but makes two kinds of errors possible: thread interference and memory consistency errors. The listing 2.6 shows an error multithreading program in Java due to multi-threads accessing and modifying the same resource simultaneously without any synchronization mechanism.

Listing 2.6: Errors in a multi-thread program

```
class NonatomicCounter {

  private int count = 0;

  public void inc() {
    count ++;
  }

  public int get() {
    return count;
  }
}

public class UnsafeInc extends Thread {

  private NonatomicCounter counter;
  private int times;

  public UnsafeInc(NonatomicCounter c, int n) {
    this.counter = c;
    this.times = n;
  }

  public void run() {
    for (int i = 0; i < times; i ++) {
      counter.inc();
    }
  }

  public static void main(String[] args) throws
     InterruptedException {
    NonatomicCounter c = new NonatomicCounter();
    Thread t1 = new UnsafeInc(c, 1000000);
    Thread t2 = new UnsafeInc(c, 1000000);
    Thread t3 = new UnsafeInc(c, 1000000);
    t1.start(); t2.start(); t3.start();
    t1.join(); t2.join(); t3.join()
    System.out.println("Counter: " + c.get());
  }
}
```

In the beginning, we initialize 3 threads with a counter time is 1000000 and a NonatomicCounter c object that is shared by 3 threads. Each thread will count with the designated times and we outcome that the final counter value will be accumulated from 3 threads. But a launch of the application (UnsafeInc) does not display the desired result Counter: 3000000 instead it actually displayed as follows: Counter: 1697864 Counter: 1700446 Counter:

2737760. Each time the application is launched, a different result is displayed.

The reason is the code line *count* + +; in NonatomicCounter class is not atomic and at least consists of three basic things: (1) read count (fetching the content v of count), (2) compute (calculate v+1), and (3) write count (store the result of v + 1 into count). *count* + +; is processed by three threads simultaneously without any protection or in an arbitrary way. When each thread t_i(i = 1,2,3) performs $read_i$ count, $compute_i$ and $write_i$ count. There are $C_3^9 x C_3^6$(1680) possible scenarios. One possible scenario:

```
read_1 count, read_2 count, read_2 count, compute_1, compute_2,
compute_3, write_1 count, write_2count, write_3 count
```

After the scenario, what is stored in *count* is 1, not 3, though each thread increases count. The effects of two increments are lost. This problem is called Race condition that is a situation in which objects (or resources) shared by multiple threads are used without any protection (or in an arbitrary way) by those threads, which may cause a different outcome each time when the program is launched.

One possible remedy is to use synchronization mechanisms supported by Java. It controls threads such that at most one thread is allowed to use shared objects (or resources) at any given moment. Each object is equipped with one lock that can be used to synchronize threads such that a thread that has acquired such a lock is allowed to enter a section in which shared objects (or resources) can be used. We have 2 ways to use such locks.

Listing 2.7: Synchronized Methods

```
public class AtomicCounter {

  private int counter = 0;

  public synchronized void inc() {
    counter ++;
  }

  public synchronized int get() {
    return counter;
  }
}
```

1. Synchronized methods: ... synchronized ... m(...) { ... }

   When a thread t executes $o.m(...)$, t first tries to acquire the lock l associated with an object o. If t has acquired l, t is allowed to invoke $m(...)$. Otherwise, t waits until t has acquired l. When t finishes

executing $m(...)$, t releases l . The listing 2.7 shows how to use synchronized methods to fix the current race condition by replacing class NonatomicCounter to class AtomicCounter.

2. Synchronized statements: synchronized (o) { ... } When a thread t executes the statement, t first tries to acquire the lock l associated with an object o. If t has acquired l, t is allowed to enter the body ... Otherwise, t waits until t has acquired l. When t leaves the body ... , t releases l . The listing 2.8 shows how to use synchronized statements to fix the current race condition by replacing class UnsafeInc to class SafeInc.

Listing 2.8: Synchronized Statements

```
public class SafeInc extends Thread {

  private NonatomicCounter counter;
  private int times;

  public UnsafeInc(NonatomicCounter c, int n) {
    this.counter = c;
    this.times = n;
  }

  public void run() {
    for (int i = 0; i < times; i ++) {
      synchronized(counter) {
        counter.inc();
      }
    }
  }

  public static void main(String[] args) throws
     InterruptedException {
    NonatomicCounter c = new NonatomicCounter();
    Thread t1 = new UnsafeInc(c, 1000000);
    Thread t2 = new UnsafeInc(c, 1000000);
    Thread t3 = new UnsafeInc(c, 1000000);
    t1.start(); t2.start(); t3.start();
    t1.join(); t2.join(); t3.join()
    System.out.println("Counter: " + c.get());
  }
}
```

Chapter 3

Specification-based Testing with Simulation Relations

We have proposed a concurrent program testing technique that is a specification-based one and uses a simulation relation candidate from a concurrent program to a formal specification [1]. The technique is depicted in Fig. 1.1. Let S be a formal specification of a state machine and P be a concurrent program. Let us suppose that we know a simulation relation candidate r from P to S. The proposed technique does the following: (1) finite state sequences $s_1, s_2, \ldots, s_n$ are generated from P, (2) each s_i of P is converted to a state s'_i of S with r, (3) one of each two consecutive states s'_i and s'_{i+1} such that $s'_i = s'_{i+1}$ is deleted, (4) finite state sequences $s''_1, s''_2, \ldots, s''_m$ are then obtained, where $s''_i \neq s''_{i+1}$ for each $i = 1, \ldots, m-1$ and (5) it is checked that $s''_1, s''_2, \ldots, s''_m$ can be accepted by S.

We suppose that programmers write concurrent programs based on formal specifications, although it may be possible to generate concurrent programs (semi-)automatically from formal specifications in some cases. The FeliCa team has demonstrated that programmers can write programs based on formal specifications and moreover use of formal specifications can make programs high-quality [18]. Therefore, our assumption is meaningful as well as feasible. If so, programmers must have profound enough understandings of both formal specifications and concurrent programs so that they can come up with simulation relation candidates from the latter to the former. Even though consecutive equal states except for one are deleted, generating $s''_1, s''_2, \ldots, s''_m$ such that $s''_i \neq s''_{i+1}$ for each $i = 1, \ldots, m-1$, there may not be exactly one transition step but zero or more transition steps so that s''_i can reach s''_{i+1} w.r.t. P. Therefore, we need to know the maximum number of such transition steps. We suppose that programmers can guess the maximum number. If programmers cannot, we can start with 1 as the maximum

number b and gradually increment b as we find consecutive states s_i'' and s_{i+1}'' such that s_i'' cannot reach s_{i+1}'' in b transition steps unless the unreachability is caused by some flaws lurking in P.

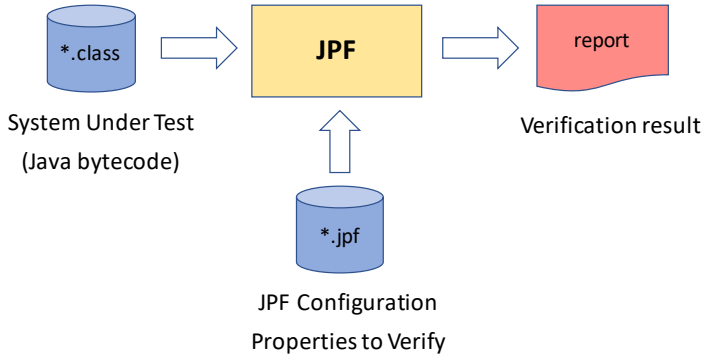
The paper [1] focuses on the left part of Fig. 1.1 but does not describe the right part of Fig. 1.1, namely how to generate state sequences from P. This thesis describes how to generate state sequences from P as well, where P is a concurrent program written in Java and Java Pathfinder (JPF) is mainly used to generate state sequences from P.

3.1 State Sequence Generation from Concurrent Programs

3.1.1 Java Pathfinder (JPF)

JPF is an extensible software model checking framework for Java bytecode programs that are generated by a standard Java compiler from programs are written in Java. JPF has a special Virtual Machine (VM) in it to support model checking of concurrent Java programs, being able to detect some flaws lurking in Java concurrent programs, such as race conditions and deadlocks, when it reports a whole execution leading to the flaw. JPF core is depicted in Fig. 3.1, given a system under test that is in the form of Java bytecode along with JPF configuration. JPF will verify follow such that configuration and return a verification result. Theoretically, JPF explores all potential executions of a program under test systematically, while an ordinary Java VM executes the code in only one possible way. JPF is basically able to identify points that represent execution choices in a program under test from which the execution could proceed differently.

Although JPF is a powerful model checker for concurrent Java programs, its straightforward use does not scale well and often encounters the notorious state space explosion. We anticipated in the paper [1] that we might mitigate the state space explosion if we do not check anything while JPF explores a program under test to generate state sequences. It is, however, revealed that we could not escape the state space explosion just without checking anything during the exploration conducted by JPF. This is because a whole big heap mainly constitutes one state in a program under test by JPF, while one state is typically expressed as a small term in formal specifications. This thesis then proposes a divide & conquer approach to state sequence generation from a concurrent program, which generates state sequences in a stratified way.



Source: `https://github.com/javapathfinder/jpf-core`

Figure 3.1: JPF Core

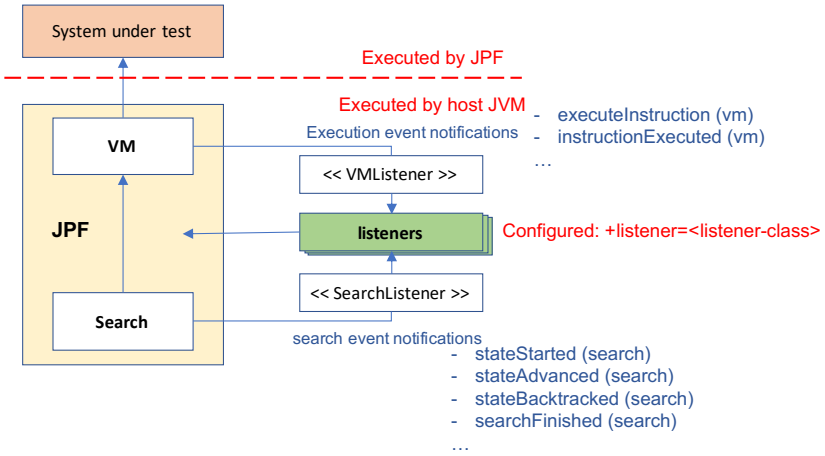
3.1.2 Generating State Sequences by JPF

Two main components of JPF are (1) a VM and (2) a search component. The VM is a state generator. It generates state representations by interpreting bytecode instructions. A state is mainly constituted of a heap and threads plus an execution history (or path) that leads to the state. Each state is given a unique ID number. The VM implements state management that makes it possible to do state matching, state storing and execution backtracking that are useful to explore a state space. Three key methods of the VM are employed by the search component:

- **forward** - it generates the next state and reports if the generated state has a successor; if so, it stores the successor on a backtrack stack for efficient restoration.
- **backtrack** - it restores the last state on the backtrack stack.
- **restoreState** - it restores an arbitrary state.

At any state, the search component is responsible for selecting the next state from which the VM should proceed, either by directing the VM to generate the next state (forward) or by telling it to backtrack to a previously generated one (backtrack). The search component works as a driver for the VM. We have some strategies used to traverse the state space. By default, the search component uses depth-first search, we can configure to use different strategies, such as breadth-first search.

The most important extension mechanism of JPF is listeners that are depicted in Fig. 3.2. They provide a way to observe, interact with and extend JPF execution. We can configure JPF with many of our own listener



Source: `https://github.com/javapathfinder/jpf-core`

Figure 3.2: JPF Listeners

classes that extend the ListenerAdapter class. The ListenerAdapter class consists of all event notifications from the VMListener and SearchListener classes. It allows us to subscribe to VMListener and SearchListener event notifications by overriding some methods, such as:

- **searchStarted** - it is invoked when JPF has just entered the search loop but before the first forward.
- **stateAdvanced** - it is invoked when JPF has just got the next state.
- **stateBacktracked** - it is invoked when JPF has just backtracked one step.
- **searchFinished** - it is invoked when JPF is just done.

Listing 3.1 shows a piece of code of class SequenceState that extends from class ListenerAdapter is made to observe and interact with JPF execution. In class SequenceState, we override the two most important methods stateAdvanced and stateBacktracked as well as searchStarted, searchFinished. As described above, the stateAdvanced method is invoked when JPF has just got the next state. We need to retrieve all the necessary information about the next state at this step. We use an instance Path of class ArrayList to keep up with the path on which we are staying. Each element of Path corresponds to a state in JPF and is encapsulated as an instance of a class Configuration we prepare. Each element of Path only stores the information for our testing purpose, which is mainly the values of observable components.

For example, the information for the test&set mutual exclusion protocol is as follows:

- **stateId** - the unique id of state.
- **depth** - the current depth of search path.
- **lock** - a Lock object that contains the lock observable component value that is true or false.
- **threads** - an ArrayList object of threads, each of which consists of the current location information that is rs or cs.

Listing 3.1: SequenceState class - a JPF listener

```
public class SequenceState extends ListenerAdapter {
  // ...
  @Override
  public void stateAdvanced(Search search) {
    if (STARTUP == 1) {
      STARTUP ++;
      startup(search.getVM());
    }
    Configuration<String> config = getConfiguration(search);
    if (config == null) {
      search.requestBacktrack();
      Logger.log("Finish program at " + search.getDepth());
      COUNT ++;
      writeSeqStringToFile();
    } else {
      seq.add(config);
      if (search.isEndState() || !search.isNewState()) {
        COUNT ++;
        writeSeqStringToFile();
      } if (DEPTH_FLAG && search.getDepth() >= DEPTH) {
        search.requestBacktrack();
        COUNT ++;
        writeSeqStringToFile();
      }
    }
    if (BOUND_FLAG && COUNT >= BOUND) {
      search.terminate();
    }
  }
  @Override
  public void stateBacktracked(Search search) {
    while (seq.size() > 0 && seq.get(seq.size() - 1).
       getStateId() != search.getStateId()) {
```

```
      seq.remove(seq.size() - 1);
    }
  }
}
```

We need to keep up with the change of observer components in each state stored in Path. Observer components are implemented as object data in JPF. So we need to look inside the heap of JPF. The heap contains a dynamic array of ElementInfo objects where the array indices are used as object reference values. An ElementInfo object contains a Fields object that actually stores the values we need. Hereby we can gather the values of observer components and create a new Configuration object and append it to Path as the stateAdvanced method is invoked. To accelerate the speed to access to heap memory, at the beginning of stateAdvanced function we gather information of object references and store into a hash table, namely lookupTable with a key is the name of object references and value is the index of its object references that is allocated in the heap of JPF. lookupTable hash table is regarded as a cached to speed up our performance. Getting observer components at each state have done by calling getConfiguration function in the listing 3.1.

In the listing 3.2 shows concretely how to get the data of a specific observer component in ABP case study. For this example, we get the data of an observer component, namely *packetsReceived*. From *getConfiguration* function, we look up the *Receiver* object from the heap of JPF by the index that is acquired by searching in the *lookupTable* hash table (cached). An *ElementInfo* ei object is returned. Finding *FieldInfo* list in the *ElementInfo ei* object, there exists a *FieldInfo* with name equal to packetsReceived , we start proceeding to get object reference value to the observer component packetsReceived in order by calling *getPacketsReceived* function. To understand this function, we should know that *packetsReceived* is maintained by a queue, each element in such a queue is a String or an array of characters in other words. Starting with *ElementInfo ei_packetsReceived* that is regarded as packetsReceived reference object. We find exactly where packetsReceived value is by accessing to the field value object with name *elementData*, reading JPF core source code to know more in detail. The data of packetsReceived is a queue, so we get a *ReferenceArrayFields* by *getArrayFields* function. We already known that each element in such a queue is a string or an array of characters. So with each element in *ReferenceArrayFields raf* object by *getValues* function, we get its *ElementInfo* by *getFieldValueObject* with given parameter is *value*, and then calling *getArrayFields* to get a *CharArrayFields* object that absolutely contains the value of an element in such a queue.

Lastly, we call *getValues* function, an array of characters is returned. Eventually, we have done to get the value of packetsReceived for ABP case study by accessing to the heap of JPF.

Listing 3.2: A piece of code to get observer components data

```
public class SequenceState extends ListenerAdapter {
  // ...
  private Configuration<String> getConfiguration(Search
     search) {
    // ...
    // Receiver
    ElementInfo ei = heap.get(lookupTable.get("main.Receiver"
       ));
    if (ei == null) {
      return null;
    }
    FieldInfo[] fis = ei.getClassInfo().getInstanceFields();
    for (FieldInfo fi : fis) {
      switch (fi.getName()) {
        // ...
        case "packetsReceived":
          ElementInfo ei_packetsReceived = (ElementInfo)fi.
             getValueObject(ei.getFields());
          ArrayList<String> packetsReceived =
             getPacketsReceived(search.getVM(),
             ei_packetsReceived);
          config.setPacketsReceived(packetsReceived);
        // ...
      }
    }
  }

  // ...
  private ArrayList<String> getPacketsReceived(VM vm,
     ElementInfo ei_packetsReceived) {
    ElementInfo ei_elementData = (ElementInfo)
       ei_packetsReceived.getFieldValueObject("elementData");
    ArrayList<String> packetsReceived = new ArrayList<String
       >();
    if (ei_elementData != null) {
      ReferenceArrayFields raf = (ReferenceArrayFields)
         ei_elementData.getArrayFields();
      for (int i : (int[])raf.getValues()) {
        if (i > 0) {
          ElementInfo ei_rf = vm.getHeap().get(i);
          ElementInfo ei_value = (ElementInfo) ei_rf.
             getFieldValueObject("value");
```

```
            CharArrayFields caf = (CharArrayFields) ei_value.
               getArrayFields();
            packetsReceived.add(String.valueOf((char[])caf.
               getValues()));
          }
        }
      }
      return packetsReceived;
    }
    // ...
}
```

Whenever JPF hits an end state, a state that has been already visited or a depth bound, we write the current path to a file after we already have deleted consecutive same states except for one. We also check if the current path has already been stored in some files. If so, we do not write the path into any files. When writing the current path into a file, we make the formats of each state and the path (state sequence) conform to those used in rewrite-theory specifications written in Maude so that Maude can check if the path can be accepted by such a rewrite-theory specification. Right after, JPF request to backtrack, we need to update the path up to date by removing elements in the path from tail until reach to the element such that its node id is the same with the node id of the backtracked node.

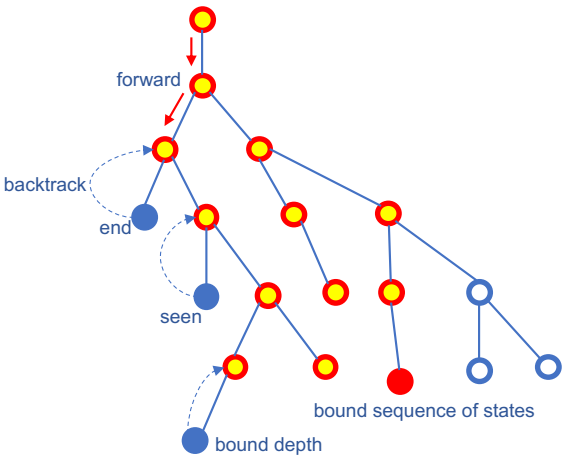


Figure 3.3: A way to generate state sequences with JPF

Because the state space could be huge even if it is bounded, we manage two bound parameters in order to prevent JPF from diverging. The two bound parameters are as follows:

- **DEPTH** - the maximum depth from the initial state; once JPF reaches

any state whose depth from the initial state is DEPTH, we send a backtrack message to the search component for backtracking.

- **BOUND** - the maximum number of paths (or state sequences); we count the number of paths generated; when the number reaches to BOUND, we send a terminate message to the search component for stopping JPF.

Each of DEPTH and BOUND could be set unbounded, meaning that we ask JPF to generate as deep state sequences as possible and/or as many state sequences as possible. Every time JPF performs backtracking, we delete the last state from Path in the stateBacktracked method to keep up with the change if the last state does not have any more successor states.

The way to generate state sequences from concurrent programs with JPF is depicted in Fig. 3.3. Yellow nodes with a thick border in red are those that have been visited by JPF. So are blue ones but they cause backtracking because the node (or state) does not have any more successor states, the node has been seen (or visited) before or the depth of the node reaches DEPTH. The red node means that when JPF arrives at the node, it has just generated the BOUND number of state sequences and then it does not need to explore the remaining part of the state space composed of the white nodes with a thick border in blue.

While gathering the data, we realize that the behavior of Java programs is a little different from the behavior of specification. Some parts of Java programs we need to make it atomic. JPF supports Atomicity control that helps us to execute a piece of code by JPF in only one transition, it is helpful for us to reduce the number of states as well as make some codes atomic as the specification.

3.2 Checking a finite semi-computation

In the paper [1], given a finite sequence $s_0, s_1, \ldots, s_n$ of states generated from S_C, each state is converted into a state in S_A with r, generating $s'_0, s'_1, \ldots, s'_n$, where $s'_i = r(s_i)$ for each i and r is used as a function from S_C states to S_A states. There may be two consecutive states s'_i and s'_{i+1} such that $s'_i = s'_{i+1}$. If so, one of them is deleted. We then generate a sequence $s''_0, s''_1, \ldots, s''_m$ of states in S_A such that there does not exists i such that $s''_i = s''_{i+1}$. We finally check if $s''_0, s''_1, \ldots, s''_m$ is a finite semi-computation of S_A.

Currently, from a concurrent program P, we generate a finite sequence $s_0, s_1, \ldots, s_n$ of states, each state already is converted into a state in S with r and removed two consecutive states s_i and s_{i+1} such that $s_i = s_{i+1}$, we just

keep one of them. We check if $s_0, s_1, \ldots, s_n$ is a finite semi-computation of S (see the left part of Fig. 1.1).

Given a module `mQid` in which a state machine is specified, two states `S1` & `S2` and the depth `B`, the function `checkSttTrans` checks if `S2` is reachable from `S1` in `B` w.r.t. the state machine, which is defined as follows:

```
ceq checkSttTrans(mQid, S1, S2, B)
= if sttTrans? :: ResultTriple
  then true else false fi
if sttTrans? :=
   metaSearch(upModule(mQid, false),
    upTerm(S1), upTerm(S2), nil, '*, B, 0) .
```

`metaSearch` is used to check if `S2` is reachable from `S1` in `B` with respect to the state machine specified as `mQid`. If there exists a path from state S1 to state S2, the checkSttTrans function returns true. Otherwise false is returned.

Given a module `mQid` in which a state machine is specified, a sequence of the state machine states and a depth `B`, the function `checkConform` checks if the state sequence is a finite semi-computation of the state machine, which is defined as follows:

```
eq checkConform(mQid, S1 | S2 | L, B)
= $checkConform(mQid, S2 | L, S1, 0, B).
eq $checkConform(mQid, nil, S, N, B)
= success .
eq $checkConform(mQid, S2 | L, S1, N, B)
= if checkSttTrans(mQid, S1, S2, B)
  then $checkConform(mQid, L, S2, N + 1, B)
  else {msg: "Failure",from: S1,to: S2,
        index: N, bound: B} fi .
```

`checkSttTrans(mQid, S1, S2, B)` checks if `S1` $\rightarrow^*_{\texttt{mQid}}$ `S2` in the depth `B`. if that such a sequence of state is a finite semi-computation of the state machine. The `checkConform` function returns success. Otherwise, it will return a message in the form of as follows:

```
{msg: "Failure",from: S1,to: S2, index: N, bound: B}
```

where *msg* specifies "Failure" message, *from* and *to* indicate that from state S1 to state S2 there does not exist any path with maximum search depth is B, *index* indicates where state S1 is located in such a sequence of states.

Given a module `mQid` in which a state machine is specified, a list of sequences of the state machine states and a depth `B`, the function `$tester` checks if all the list of state sequences is finite semi-computations of the state machine, which is defined as follows:

```
eq $tester(mQid, empty, B, N) = success .
ceq $tester(mQid, (LC , LS), B, N)
=  if R4C? :: Result4Conform
 then $tester(mQid, LS, B, N + 1)
 else {
   seq: N,
   msg: getMsg(R4C?),
   from: getFrom(R4C?),
   to: getTo(R4C?),
   index: getIndex(R4C?),
   bound: getBound(R4C?)
 } fi
 if R4C? := checkConform(mQid, LC, B) .
```

The `checkConform` function is used in the `$tester` function by iterating through such a list of sequences of states, each a sequence of states is checked by `checkConform` function. If all state sequences pass successfully, it returns success. Otherwise, there exists a state sequence does not satisfy a finite semi-computation, the program will terminate and a message is returned in the form of as follows:

```
{ seq: N, msg: getMsg(R4C?), from: getFrom(R4C?), to: getTo(R4C?),
index: getIndex(R4C?),bound: getBound(R4C?) }
```

where *seq* indicates where such an error sequence of states is located in the list state sequences where it is located. *msg*, *from*, *to*, *index* and *bound* are the same information as the output of `checkConform` function.

3.3 Summary

We have presented how to generate state sequences by JPF as well as how to check a semi-computation by meta programming in Maude. The most advantage of model checking is to systematically explore entire state space in theory. We exploit itself functionality in JPF model checker to cover all possible path execution. To accomplish it we need to observe and interact with JPF via Listeners that are supported by JPF. Note that JPF often encounters the state space explosion, so we need to set two bound parameters to ensure JPF may terminate. Of course, each of DEPTH and BOUND could be set unbounded if you want to explore entire state space. The meta programming in Maude is most exciting to build many meta applications, we have used metaSearch as a core function to check if a state S2 is reachable from a state S1 in a depth B with respect to the state machine specified

as mQid. Thereby, we can check a state sequence whether it is a semi-computation.

Chapter 4

Divide & Conquer Approach to Testing Concurrent Programs

State sequence generation from concurrent programs by using JPF that often encounters the notorious state space explosion even without checking any property violation while searching due to the coherent non-determinism threads interleaving. This is the main challenge that is shared by model checking software. From our point of view, the traditional techniques often use model checkers in sequential testing. Besides, some model checking techniques with parallelization have studied [6] [7] and the result is impressive. So we come up with a divide & conquer approach to generate state sequences from concurrent programs and check if such state sequences are accepted by specifications in a stratified way. The approach aims are to alleviate the state space explosion problem as well as building up a scalable testing technique. In this chapter, we explain in order how to implement exactly the approach in our environment.

4.1 A Divide & Conquer Approach to Generating State Sequences

A divide & conquer approach is the basis of efficient algorithms for many kinds of problems in computer sciences. The advantage of the approach is to break down a problem into two or more sub problems, until these become simple enough to be solved directly. The solutions of the sub-problem are combined together to give a final solution. Coming back to state sequence generation from concurrent programs by using JPF. When you do not set each of DEPTH and BOUND to a moderately small number and ask JPF to exhaustively (or almost exhaustively) explore all (or a huge number of)

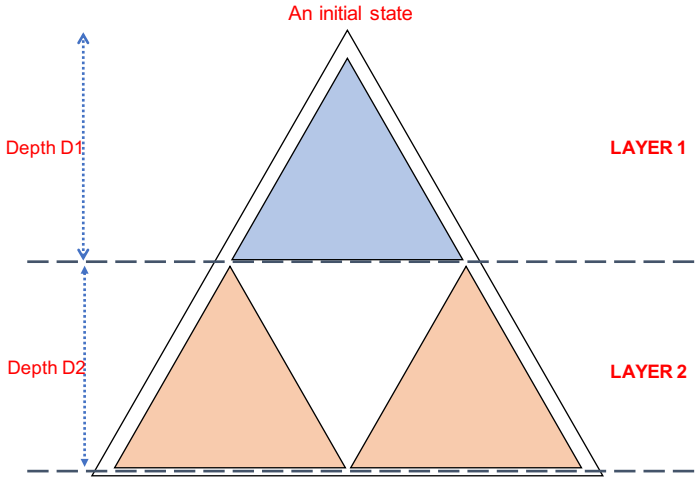


Figure 4.1: A divide & conquer approach

possible states, JPF may not finish the exploration and may lead to out of memory. To mitigate the situation, the present thesis proposes a technique to generate state sequences from concurrent programs in a stratified way, which is called a divide & conquer approach to generating state sequences. We firstly generate state sequences from each initial state, where DEPTH is D1 (see Fig. 4.1). Note that BOUND may be set unbounded. If D1 is small enough, it is possible to do so. We then generate state sequences from each of the state at depth D1, where DEPTH is D2 (see Fig. 4.1). If D2 is small enough, it is also possible to do so. Given one initial state, there is one sub-state space in the first layer explored by JPF, while there are as many sub-state spaces in the second layer as the states at depth D1 (see Fig. 4.1). Combining each state sequence seq_1 in layer 1 and each state sequence seq_2 in layer 2 such that the last state of seq_1 equals the first state of seq_2, we are to generate state sequences, where DEPTH is D1 + D2 (see Fig. 4.1), which can be done even though D1 + D2 is too large. Although we have described the divide & conquer approach to generating state sequences such that there are two layers, the technique could be generalized such that the number of layers is $N \geq 2$. For example, we could generate state sequences from each of the states at depth D1 + D2, where DEPTH is D3; Combining each state sequence seq_1 in layer 1, each state sequence seq_2 in layer 2 and each state sequence seq_2 in layer 3 such that the last state of seq_1 equals the first state of seq_2 and the last state of seq_2 equals the first state of seq_3, we are to generate state sequences, where DEPTH is D1 + D2 + D3.

Generating state sequences for each sub-state space is independent from that for any other sub-state space. Especially for sub-state spaces in one

layer, generating state sequences for each sub-state space is totally independent from that for each other. This characteristic of the proposed technique makes it possible to generate state sequences from concurrent programs in parallel. For example, once we have generated state sequences in layer 1, we can generate state sequences for all sub-state spaces in layer 2 simultaneously. This is another advantage of the divide & conquer approach to generating state sequences from concurrent programs.

Let us consider the test&set protocol and suppose that we write a concurrent program (denoted $P_{\text{t\&s}}$) in Java based on the specification $S_{\text{t\&s}}$ of the protocol. We suppose that there are three processes participating in the protocol. $S_{\text{t\&s}}$ has one initial state and so does $P_{\text{t\&s}}$. Let each of D1 and D2 be 50 and use the proposed technique to generate state sequences from $P_{\text{t\&s}}$. One of the state sequences (denoted seq_1) generated in layer 1 is as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} |
{(pc[p1]: rs) (pc[p2]: cs) (pc[p3]: rs) (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} | nil
```

where `_|_` is the constructor for non-empty state sequences and `nil` denotes the empty state sequence. Note that atomic execution units used in $P_{\text{t\&s}}$ are totally different from those used in $S_{\text{t\&s}}$. Therefore, the depth of layer 1 is 50 but the length of the state sequence generated is 3. One of the state sequences (denoted seq_2) generated in layer 2 is as follows:

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} |
{(pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs) (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} | nil
```

Note that the last state in the first state sequence is the same as the first state in the second state sequence. Combining them the two state sequences, we get the following state sequence (denoted seq_3) :

```
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} |
{(pc[p1]: rs) (pc[p2]: cs) (pc[p3]: rs) (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} |
{(pc[p1]: cs) (pc[p2]: rs) (pc[p3]: rs) (lock: true)} |
{(pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (lock: false)} | nil
```

This is one state sequence generated from $P_{\text{t\&s}}$, where DEPTH is 100. So using the divide & conquer approach to generating state sequences, we are able to generate longer or deeper state sequences that are unfeasible to use JPF only without our approach.

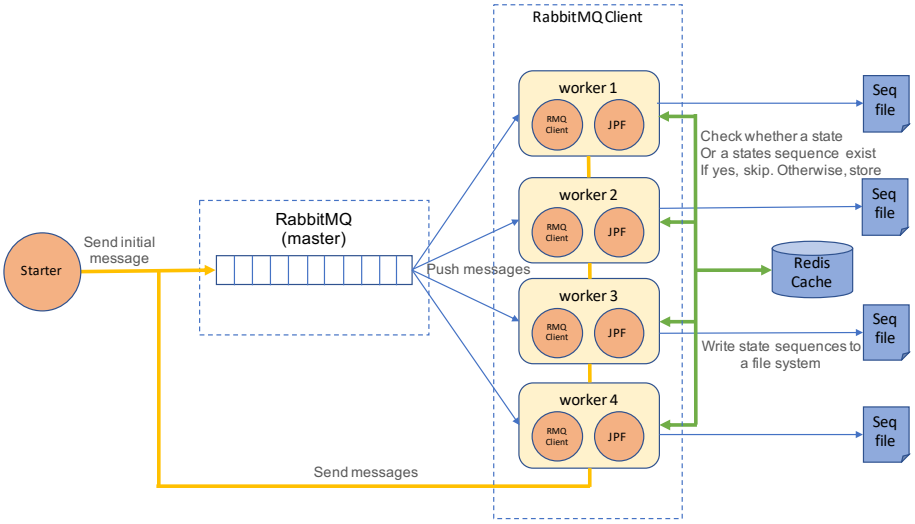


Figure 4.2: The architecture of a tool supporting the proposed technique

4.2 Environment Architecture

Once state sequences are generated from a concurrent program P, we check if a formal specification S can accept the state sequences with Maude. For example, we can check if seq_3 can be accepted by $P_{\text{t\&s}}$ with Maude. Instead of checking if seq_3 can be accepted by $P_{\text{t\&s}}$, however, it suffices to check if each of seq_1 and seq_2 can be accepted by $P_{\text{t\&s}}$.

For each layer l, we generate state sequences starting from each state located at depth $\text{D}1 + \ldots + \text{D}(l-1)$ from a concurrent program P with JPF and check if each state sequence generated in layer l can be accepted by a formal specification S with Maude. We could first generate all (sub-)state sequences from P in the stratified way and then could check if each state sequence can be accepted by S. But, we do not combine multiple (sub-)state sequences to generate a whole state sequence of P because we do not need to do so and it suffices to check if each (sub-)state sequence can be accepted by S in order to check if a whole state sequence can be accepted by S. This way to generate (sub-)state sequences from P and to check if each (sub-)state sequence is accepted by S is called a divide & conquer approach to testing concurrent programs.

Our tool that supports the divide & conquer approach to testing concurrent programs has been implemented in Java. The tool architecture is depicted in Fig. 4.2. As shown in Fig. 4.2, the architecture is a master-worker model (or pattern), where there is one master and four workers. We use

Redis [19] and RabbitMQ [20] to quickly develop in our tool.

- Redis is an advanced key-value store and supports many different kinds of data structures such as strings, lists, maps, sets, sorted sets, ... It holds its database entirely in memory. We can imagine it as a big hash table in memory. Redis is used as an effective cache to avoid duplicating states and state sequences when generating state sequences.

- RabbitMQ is used as a message broker. The RabbitMQ master maintains a message queue to dispatch messages to RabbitMQ (RMQ) clients. Each worker consists of a RabbitMQ client and JPF.

Initially, we run a starter program to send an initial message to the RabbitMQ master for kicking off the tool. The listing 4.1 shows how such a starter program works. Basically, the starter program is a worker in our environment. But it is just specialized to send an initial message. First of all, we flush all keys and values from the Redis server cache to clean up data in memory. Secondly, we make a connection to RabbitMQ master server with a designated configuration. After making sure that it is connected, then we prepare initial data to push an initial job to message queue. The data is encapsulated into a *Configuration* object, namely *config*. Before publishing the data to the master server, we use *SerializationUtils* to *serialize* the *config* object. It is easy to unserialize to the original object at the receiver side without doing any extra thing and effectively convey messages from the worker nodes to the master server.

Listing 4.1: A starter program in Java

```
public class Starter extends RabbitMQ {

  public static void main(String[] argv) {
    // ...
    // Flush all keys and values from redis server
    RedisClient.getInstance().getConnection().flushAll();
    // Push an initial job to message queue
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(getHost());
    try (Connection connection = factory.newConnection();
        Channel channel = connection.createChannel()) {
      channel.queueDeclare(QUEUE_NAME, false, false, false,
          null);
      // prepare to send a message to queue
      Configuration<String> config = new Configuration<String
          >();
      List<String> sentPackets = Arrays.asList(Env.PACKETS);
      List<String> recPackets = new ArrayList<String>();
```

```
      main.Channel<Pair<String,Boolean>> ch1 = new main.
         Channel<Pair<String,Boolean>>(Env.BOUND);
      main.Channel<Boolean> ch2 = new main.Channel<Boolean>(
         Env.BOUND);
      Cell<Boolean> f = new Cell<Boolean>(false);
      // 1st argument: packetsToBeSent
      config.setPacketsToBeSent(sentPackets);
      // 2nd argument: packetsReceived
      config.setPacketsReceived(recPackets);
      // 3rd argument: index of packetsToBeSent
      config.setIndex(0);
      // 4th argument: finish flag
      config.setFinish(f);
      // 5th argument: flag1
      config.setFlag1(true);
      // 6th argument: flag2
      config.setFlag2(true);
      // 7th argument: channel1
      config.setChannel1(ch1);
      // 8th argument: channel2
      config.setChannel2(ch2);
      byte[] data = SerializationUtils.serialize(config);
      channel.basicPublish("", QUEUE_NAME, null, data);
    }
    // ...
  }
}
```

As soon as the master server has received a message from a worker, it will be stored in a queue. By default, RabbitMQ master server will pop a message from the queue and then dispatch to a worker, in sequence. On average every worker will get the same number of messages due to using round-robin scheduling for distributing messages. The listing 4.2 shows us how it works. Firstly, workers make a connection to the master server. After having connected, workers are ready for receiving messages from the master server. Whenever a worker receives a message from the master, the worker deserializes a message to get the original object by using *SerializationUtils*. It says that we have a *config* object belong to class *Configuration*. Passing *config* object to the constructor of *RunJPF* and call *start* function. The worker internally starts JPF instance with a configuration that is built from the message. To know how to internally start JPF in our independent Java program, you may look at in the listing 4.4. A brief description, we get a configuration and pass to the constructor of JPF, then add our own Listener class to JPF and then run it.

Listing 4.2: A worker program in Java

```
public class Receiver extends RabbitMQ {
  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(getHost());
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare(QUEUE_NAME, false, false, false,
       null);
    DeliverCallback deliverCallback = (consumerTag, delivery)
        -> {
      Configuration<String> config = SerializationUtils.
         deserialize(delivery.getBody());
      RunJPF runner = new RunJPF(config);
      runner.start();
      try {
        runner.join();
        channel.basicAck(delivery.getEnvelope().
           getDeliveryTag(), false);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    };
    boolean autoAck = false;
    channel.basicConsume(QUEUE_NAME, autoAck, deliverCallback
       , consumerTag -> { });
  }
}
```

Note that all workers as well as JPF programs are using the same a Redis instance as a shared cache. JPF program traverses the (sub-)state space designated in the message. Whenever JPF reaches the designated depth or finds the current state have no more successor states, our listener class does the following.

1. Removing all consecutive same states but except one from the state sequence.

2. Converting the state sequence to a string representation by *seqToString* function in the listing 4.3 and using the SHA256 algorithm by *getSHA* function support by *GFG* class to generate a certain unique signature for the string.

3. Asking the Redis cache whether the state sequence exists by sending *exists* message to *jedis* object; If yes, skipping what follows; Otherwise, saving the signature as the key and the string as the value into the Redis cache and writing the string into the file maintained by the worker.

4. Obtaining the last state from the state sequence, converting it to a string representation and using the SHA256 algorithm to hash the string to a unique signature.

5. Asking the Redis cache whether the state exists; If yes, skipping what follows; Otherwise, asking Redis to save the signature as the key and the string as the value into the Redis cache and sending a message that contains the last state's information to the RabbitMQ master, which then prepares a message that asks a worker to generate state sequences from the state.

The listing 4.3 shows how to avoid the state exists, the state sequence exists, write state sequences to files as well as sending a new state that is encapsulated into a message to the RabbitMQ master. We have used SHA256 algorithm, Redis, RabbitMQ to accomplish it as described above.

Listing 4.3: A worker program in Java

```
public void writeSeqStringToFile() {
  try {
    if (seq.size() > 0) {
      String seqString = seqToString();
      String seqSha256 = GFG.getSHA(seqString);
      if (!jedis.exists(seqSha256)) {
        jedis.set(seqSha256, seqString);
        graph.write(seqString + " , ");
        graph.newLine();
        SEQ_UNIQUE_COUNT ++;
      }
      Configuration<String> lastElement = seq.get(seq.size()
          - 1);
      if (lastElement != null) {
        String elementSha256 = GFG.getSHA(lastElement.
            toString());
        if (!jedis.exists(elementSha256)) {
          jedis.set(elementSha256, lastElement.toString());
          // TODO :: submit job to the queue broker
          if (lastElement.getFinish().get() == false) {
            mq.Sender.getInstance().sendJob(lastElement);
          }
        }
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listing 4.4: How to internally start JPF

```
public class RunJPF extends Thread {
  public void run() {
    try {
      String[] configString = configList.toArray(new String[
          configList.size()]);
      Config conf = JPF.createConfig(configString);
      conf.setProperty("report.console.finished", "result");
      JPF jpf = new JPF(conf);
      SequenceState seq = new SequenceState();
      jpf.addListener(seq);
      jpf.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The current tool has not yet been integrated to Maude. It requires human users to feed state sequences generated into Maude. Because each file maintained by a worker could be huge, however, it is necessary to split the file into multiple smaller ones, each of which is fed into Maude, because it would take much time to feed a huge file into Maude. The tool will be integrated to Maude, including splitting of huge files into multiple smaller files.

4.3 Summary

We have explained how to apply our divide & conquer approach to testing concurrent programs. Typically, a divide & conquer approach to generate state sequences and expose completely our environment architecture. As mentioned above, If we only use one JPF instance, JPF program cannot terminate or even reach to out of memory soon due to traverse on a large state space. To alleviate this problem to apply to our tool, we come up with the divide & conquer approach. If $\mathrm{D1} + \ldots + \mathrm{D}(l-1)$ are small enough, we can generate all (sub-)state sequences from each state located at depth $\mathrm{D1} + \ldots + \mathrm{D}(l-1)$ from a concurrent program P in the stratified way. But, we do not combine multiple (sub-)state sequences to generate a whole state sequence of P because it suffices to check if each (sub-) state sequence can be accepted by S in order to imply that if a whole state sequence can be accepted by S. Leveraging the master-worker model, we parallelize state sequence generation to avoid JPF running out of memory. Each worker will conquer a sub space in a whole state space, it effectively reduces the burden for JPF. Besides, we use a Redis instance as a big hash table, it stores distinct

states as well as state sequences to eliminate duplication while generating state sequences.

Chapter 5

Case Studies

The Transmission Control Protocol (TCP), one of the most important protocols of the Internet protocol suites, is used widely in almost our application program nowadays. It is very worth to study this kind of protocol, so in our experiments, we choose Alternating Bit Protocol (ABP) that is a simplified version of TCP and another one is Simple Communication Protocol (SCP) that is a simplified version of ABP. Instead of using a queue as ABP, SCP uses a cell to specify and maintain data channel and ask channel. Both ABP and SCP protocols are a communication protocol, they ensure that a sender may send correct messages in order to a receiver over an unreliable network. In the paper [1], SCP and ABP are regarded as an abstract specification and a concrete specification, respectively. In this chapter, we present completely the experiment results of our proposed method over SCP and ABP protocols. The experiment results indicate that the proposed techinique is able to mitigate the state space explosion problem from which otherwise only one JPF instance cannot suffer.

5.1 Simple Communication Protocol (SCP)

Simple Communication Protocol (SCP), a communication protocol, is used as one running example. SCP consists of a sender, a receiver and two channels between them. One channel called dc (data channel) is a cell that is used to transfer pairs $\langle d, b\rangle$, where d is a data value and b is a Boolean value, to the receiver from the sender, and the other channel called ac (ack channel) is a cell that is used to deliver Boolean values (as ack) to the sender from the receiver. Both cells are unreliable in that the contents may drop. The sender maintains two pieces of information that are sb (sender bit) and data. sb is a Boolean value and data is the data to be delivered next to the receiver. The

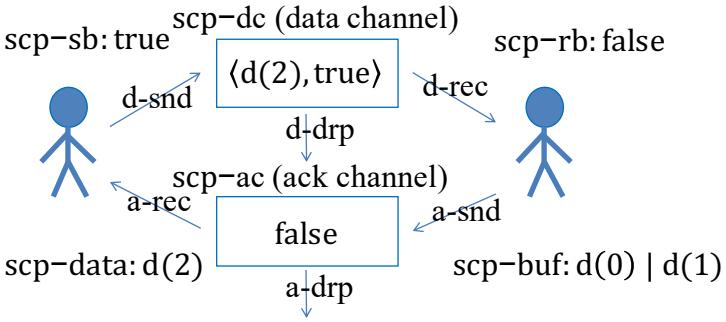


Figure 5.1: A state of SCP

receiver maintains two pieces of information that are rb (receiver bit) and buf. rb is Boolean value and buf is the list of data received so far. Initially, sb is true, data is d(0), rb is true, buf is empty, dc is empty and cc is empty. The sender has two actions to do that are d-snd and d-rec. d-snd does the following: the pair $\langle \text{data}, \text{sb} \rangle$ is put into dc. d-rec does the following: if ac has a Boolean value b, then b is extracted and if $b \neq$ sb, then data is set to the next data and sb is negated and otherwise, nothing changes. The receiver has two actions to do that are a-snd and a-rec. a-snd does the following: rb is put into ac. a-rec does the following: if dc has $\langle d, b \rangle$, then $\langle d, b \rangle$ is extracted and if $b =$ rb, then d is added to buf at the end and rb is negated and otherwise nothing changes. There are two more actions that are d-drp and a-drp. d-drp does the following: if dc is not empty, dc becomes empty. a-drp does the following: if ac is not empty, ac becomes empty. Fig. 5.1 shows a state of SCP.

A state of SCP is expressed as follows:

```
{(scp-sb: b1) (scp-data: d(n)) (scp-rb: b2)
 (scp-buf: dl) (scp-dc: cell1) (scp-ac: cell2)}
```

Each of the six actions in SCP is formalized as state transitions, which are described in Maude (conditional) rewrite rules (or rules) as follows:

```
rl [d-snd] : {(scp-sb: B)(scp-data: D) (scp-dc: DC) OCs}
=> {(scp-sb: B)(scp-data: D) (scp-dc: (< D,B >)) OCs} .

crl [a-rec1] : {(scp-sb: B)(scp-data: d(N)) (scp-ac: B') OCs}
=> {(scp-sb: (not B))(scp-data: d(N + 1)) (scp-ac: empc) OCs}
if B =/= B' .
```

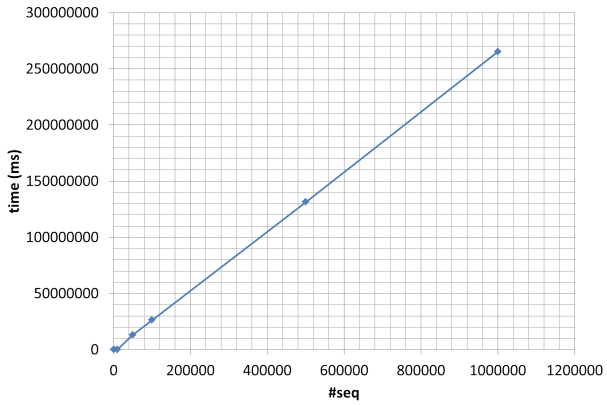


Figure 5.2: Time taken when the length of each state sequence is fixed (100) and the number of state sequences is changed (100, 1000, 10000, 50000, 100000, 500000 & 1000000)

```
crl [a-rec2] : {(scp-sb: B)(scp-data: D) (scp-ac: B') OCs}
=> {(scp-sb: B)(scp-data: D)(scp-ac: empc) OCs}
if B = B' .

rl [a-snd] : {(scp-rb: B)(scp-ac: AC) OCs}
=> {(scp-rb: B)(scp-ac: B) OCs} .

crl [d-rec1] : {(scp-rb: B)(scp-buf: Ds) (scp-dc: (< D,B' >)) OCs}
=> {(scp-rb: (not B))(scp-buf: (Ds | D)) (scp-dc: empc) OCs}
if B = B' .

crl [d-rec2] : {(scp-rb: B)(scp-buf: Ds) (scp-dc: (< D,B' >)) OCs}
=> {(scp-rb: B)(scp-buf: Ds) (scp-dc: empc) OCs}
if B =/= B' .

rl [d-drp] : {(scp-dc: P) OCs}
=> {(scp-dc: empc) OCs} .

rl [a-drp] : {(scp-ac: B) OCs}
=> {(scp-ac: empc) OCs} .
```

SCP is a simplified version of ABP, SCP uses Cells to maintain data channel and ack channel, but ABP uses Queues instead of. On other words, ABP is general of SCP. When we set the maximum number of elements may be in the queue to 1, ABP is SCP.

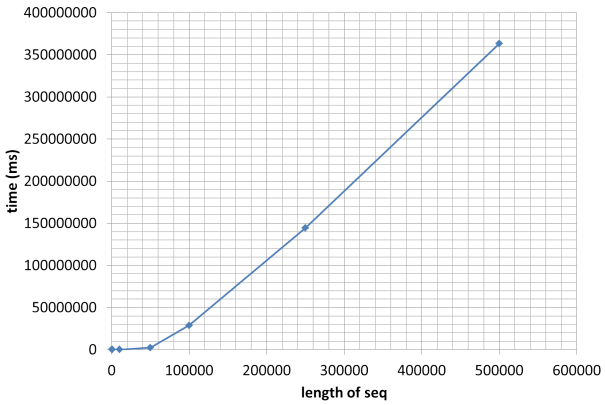


Figure 5.3: Time taken when the number of state sequences is fixed (1) and the length of the state sequence is changed (100, 1000, 10000, 50000, 100000, 250000 & 500000)

In our paper [1], we used the SCP as a abstract specification and ABP as a concrete specification in Maude to measure time taken to generate state sequences from the ABP specification, transform them with the simulation relation from ABP to SCP to other state sequences, and check if the state sequences obtained can be accepted by the SCP specification. We used one node of SGI UV3000 that carries 2.90GH microprocessor and 256GB memory for the experiments. Two sets of experiments were conducted. One set is to fix the length of each state sequence, which is 100, and modify the number of state sequences generated, which is one of 100, 1000, 10000, 50000, 100000, 500000 and 1000000. The other set is to fix the number of state sequences generated, which is one, and modify the length of the state sequence, which is one of 100, 1000, 10000, 50000, 100000, 250000 and 500000. For both sets of experiments, 2 was used as the depth of state transitions. Fig. 5.2 shows the experimental results for the first set. The time taken increases almost linearly as the number of state sequences generated increases. Fig. 5.3 shows the experimental results for the second set. The time taken increases a bit greater than linearly as the length of the state sequence generated increases. If we use 1 as the depth instead of 2, we get the following an error result:

```
{msg: "Failure",from: {scp-sb: true scp-data:
d(0) scp-rb: false scp-buf: d(0) scp-dc:
< d(0),true  > scp-ac: false},to: {scp-sb:
false scp-data: d(1) scp-rb: false scp-buf:
d(0) scp-dc: < d(0),true > scp-ac: false},
index: 20,bound: 1}
```

This is because the transition between those two states is impossible with depth 1. In other words, the state following `to:` is not reachable from the state following `from:` with depth 1 in SCP specification.

5.2 Alternating Bit Protocol (ABP)

We study on another case study in which ABP has been used. ABP is a communication protocol and can be regarded as a simplified version of TCP. ABP makes it possible to reliably deliver data from a sender to a receiver even though two channels between the sender and receiver are unreliable in that elements in the channels may be dropped and/or duplicated. The sender maintains two pieces of information: sb that stores a Boolean value and $data$ that stores the data to be delivered next. The receiver maintains two pieces of information: rb that stores a Boolean value and buf that stores the data received. One channel dc from the sender to the receiver carries pairs of data and Boolean values, while the other one ac from the receiver to the sender carries Boolean values. There are two actions done by the sender: (sa1) the sender puts the pair $(data, sb)$ into dc and (sa2) if ac is not empty, the sender extracts the top Boolean value b from ac and compares b with sb; if $b \neq sb$, $data$ becomes the next data and sb is complemented; otherwise nothing changes. Actions (sa1) and (sa2) done by the sender are denoted d-snd and a-rec, respectively. There are two actions done by the receiver: (ra1) the receiver puts rb into ac and (ra2) if dc is not empty, the sender extracts the top pair (d, b) from dc and compares b with rb; if $b = sb$, d is stored in buf and rb is complemented; otherwise nothing changes. Actions (ra1) and (ra2) done by the receiver are denoted a-snd and d-rec, respectively. There are four more actions to dc and ac because the channels are unreliable. If dc is not empty, the top element is dropped (d-drp) or duplicated (d-dup), and if ac is not empty, the top element is dropped (a-drp) or duplicated (a-dup). Fig. 5.4 shows a graphical representation of a state of ABP.

Each state of ABP is formalized as a term $\{(\mathrm{sb} : b_1)\ (\mathrm{data} : \mathrm{d}(n))\ (\mathrm{rb} : b_2)\ (\mathrm{buf} : dl)\ (\mathrm{dc} : q_1)\ (\mathrm{ac} : q_2)\}$, where b_1 and b_2 are Boolean values, n is a natural number, dl is a data list, q_1 is a queue of pairs of data and Boolean values and q_2 is a queue of Boolean values. $\mathrm{d}(n)$ denotes data to be delivered from the sender to the receiver. Initially, b_1 is true, b_2 is true, n is 0, dl is the empty list, q_1 is the empty queue and q_2 is the empty queue. The state transitions that formalize the actions are specified in rewrite rules as follows:

```
rl [d-snd] : {(sb: B)(data: D)(dc: Ps) OCs}
=> {(sb: B)(data: D)(dc:(Ps | < D,B >)) OCs} .
```

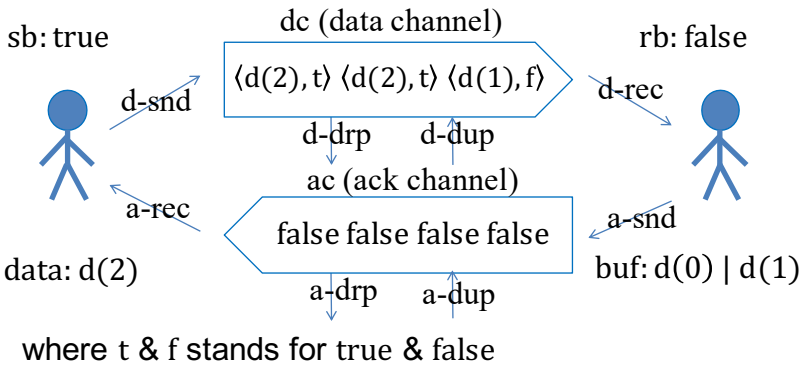


Figure 5.4: A state of ABP

```
crl [a-rec1] : {(sb: B)(data: d(N))(ac: (B' | Bs)) OCs}
=> {(sb:(not B))(data: d(N + 1))(ac: Bs) OCs} if B =/= B' .

crl [a-rec2] : {(sb: B)(data: D)(ac: (B' | Bs)) OCs}
=> {(sb: B)(data: D)(ac: Bs) OCs} if B = B' .

rl [a-snd] : {(rb: B)(ac: Bs) OCs}
=> {(rb: B) (ac: (Bs | B)) OCs} .

crl [d-rec1] : {(rb: B)(buf: Ds)(dc: (< D,B' > | Ps)) OCs}
=> {(rb: (not B))(buf: (Ds | D))(dc: Ps) OCs} if B = B' .

crl [d-rec2] : {(rb: B)(buf: Ds)(dc: (< D,B' > | Ps)) OCs}
=> {(rb: B)(buf: Ds)(dc: Ps) OCs} if B =/= B' .

rl [d-drp] : {(dc: (Ps1 | P | Ps2)) OCs}
=> {(dc: (Ps1 | Ps2)) OCs} .

rl [d-dup] : {(dc: (Ps1 | P | Ps2)) OCs}
=> {(dc: (Ps1 | P | P | Ps2)) OCs} .

rl [a-drp] : {(ac: (Bs1 | B | Bs2)) OCs}
=> {(ac: (Bs1 | Bs2)) OCs} .

rl [a-dup] : {(ac: (Bs1 | B | Bs2)) OCs}
=> {(ac: (Bs1 | B | B | Bs2)) OCs} .
```

Words that start with a capital letter, such as B, D, Ps and OCs, are Maude

Table 5.1: Experimental data

Channel size	Worker	Time1 (d:h:m)	#seqs	#small files	Total size (MB)	Time2 (s)
1	Worker 1	1:0:57	1,381	2	0.5	13
	Worker 2	1:0:57				
	Worker 3	1:0:57				
	Worker 4	0:18:43				
2	Worker 1	2:17:51	8,879	9	4.3	118
	Worker 2	2:11:45				
	Worker 3	2:10:52				
	Worker 4	2:11:51				
3	Worker 1	3:23:53	24,416	25	13.5	355
	Worker 2	3:20:27				
	Worker 3	3:20:37				
	Worker 4	3:17:41				

- Time1 – time taken to generate state sequences with JPF.
- Time2 – time taken to check if state sequences are accepted by the formal specification with Maude.
- Total size – the total size of all state sequences generated.

variables. B, D, Ps and OCs are variables of Boolean values, data, queues of (Data,Bool)-pairs and observable component soups, respectively. The types (or sorts) of the other variables can be understood from what have been described. The two rewrite rules `a-rec1` and `a-rec2` formalize action a-rec. What rewrite rules formalize what actions can be understood from what have been described. Let S_{ABP} refer to the specification of ABP in Maude. A concurrent program P'_{ABP} is written in Java based on S_{ABP}, where one thread performs two actions d-snd and a-rec, one thread performs two actions a-snd and d-rec, one thread performs two actions d-drp and a-drp and one thread performs two actions d-dup and a-dup. We intentionally insert one flaw in P'_{ABP} such that when the receiver gets the third data, it does not put the third data into buf but puts the fourth data into buf.

We suppose that the sender is to deliver four data to the receiver, the maximum state transition bound is 2, DEPTH is 100 for each layer and BOUND is unbounded for each layer. The simulation relation candidate from P'_{ABP} to S_{ABP} is essentially the identity function. We change each channel size as follows: 1, 2 and 3. We do not need to fix the number of layers in advance, but the number of layers can be determined by the tool on the fly. For each experiment, however, the number of layers is larger than 2. The experiments were carried out by an Apple iMac Late 2015 that had

Processor 4GHz Intel Core i7 and Memory 32 GB 1867 MHz DDR3. The experimental data are shown in Table 5.1. Each small file contains at most 1,000 state sequences.

When each channel size is 1, it takes about 1 day to generate all state sequences with four workers. The number of the state sequences generated is 1,381, which is split into two groups (files): one file has 1,000 state sequences and the other file has 381 sequences. Each file is fed into Maude to check if each state sequence is accepted by S_{ABP} with Maude. It takes 13s to do all checks. Maude detects that some state sequences have an adjacent states s and s' such that s cannot reach s' by S_{ABP} in two state transitions. If that is the case, a tool component [1] implemented in Maude shows us some information as follows:

```
Result4Driver?: {seq: 31,msg: "Failure",
from: {sb: true data: d(2) rb: true buf: (d(0) | d(1))
       dc: < d(2),true > ac: nil},
to:{sb: true data: d(2) rb: false buf: (d(0) | d(1) | d(3))
    dc: nil ac: nil},index: 3,bound: 2}
```

This is because although the receiver must put the third data `d(2)` into buf when `d(2)` is delivered to the receiver, the receiver instead puts the fourth data `d(3)` into buf, which is the flaw intentionally inserted into P'_{ABP}. This demonstrates that our tool can detect the flaw.

When each channel size is 2, it takes about 2.75 days to generate all state sequences with four workers. The number of the state sequences generated is 8,879, which is split into nine groups (files): eight files have 1,000 state sequences and one file has 879 sequences. Each file is fed into Maude to check if each state sequence is accepted by S_{ABP} with Maude. It takes 118s to do all checks. As is the case in which each channel is 1, Maude detects that some state sequences have an adjacent states s and s' such that s cannot reach s' by S_{ABP} in two state transitions due to the flaw intentionally inserted in P'_{ABP}.

When each channel size is 3, it takes about 4 days to generate all state sequences with four workers. The number of the state sequences generated is 24,416, which is split into 25 groups (files): 24 files have 1,000 state sequences and one file has 416 sequences. Each file is fed into Maude to check if each state sequence is accepted by S_{ABP} with Maude. It takes 355s to do all the checks. As is the case in which each channel is 1, Maude detects that some state sequences have an adjacent states s and s' such that s cannot reach s' by S_{ABP} in two state transitions due to the flaw intentionally inserted in P'_{ABP}.

Note that when each channel is 3, the straightforward use of JPF did not complete the model checking but caused out of memory after it spent about 4 days to try exploring the reachable state space with almost the same computer as the one used in the experiments reported in the present paper [21]. Therefore, the proposed technique can alleviate the out-of-memory situation due to the state space explosion.

The experimental results say that it takes a few days to generate state sequences from a concurrent Java program with our tool in which JPF plays the main role, while it only takes several minutes to check if state sequences are accepted by a formal specification with Maude.

5.3 Summary

We have conducted to experiment on 2 case studies SCP and ABP. SCP is a simplified version of ABP. When the channel size is assigned to 1 in the ABP case study, ABP may be regarded as SCP. Our experiment results say that we can generate all state sequences from ABP program with the channel size from 1 up to 3. When the channel size of ABP is 1, 2 or 3. It took about 1 day, 2.75 days, 4 days, respectively. The path execution is so large that one JPF instance cannot terminate and reach to out of memory soon. One notice that we need to take into account how much the DEPTH value for each layer should be. A good DEPTH value makes sure that JPF may explore entire (sub)state sequences and JPF program has to complete in a reasonable time. In our experiments, we use DEPTH is 100 for each layer by heuristic, of course, BOUND is unbounded for each layer. The experiment results have demonstrated that the proposed technique can mitigate the state space explosion instances from which otherwise only one JPF instance cannot suffer while generating state sequences.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have proposed a new testing technique for testing concurrent programs. The technique is a specification-based testing one. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S. We have used S that is specified in Maude and P that is implemented in Java. Java Pathfinder (JPF) and Maude are then used to generate state sequences from P and to check if such state sequences are accepted by S, respectively. Even without checking any property violations with JPF, JPF often encounters the notorious state space explosion while only generating state sequences. Thus, we have proposed a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way. Using only one JPF instance, JPF cannot terminate and reach to out of memory soon. Leveraging the master-worker model, we use several JPF instances to parallelize state sequences generation from P in a stratified way. Our implementation has contributed to three main modules:

1. *Semi-computation Checker:* we have used meta programming in Maude to develop a checker module. Given a specification S, a state sequences Seq. Simi-computation Checker module automatically detects whether the state sequences Seq conforms to the specification S to give a report.

2. *State Sequences Generation:* we have generated state sequences from concurrent program P by using the advantages of model checker JPF.

3. *Divide & Conquer Approach to Generate State Sequences:* we have built up a scalable environment for state sequences generation by our divide & conquer approach with the master-worker model.

Some experiments have demonstrated that the proposed technique mitigates the state space explosion instances from which otherwise only one JPF instance cannot suffer.

Advantages:

1. *Specification-based testing:* We may check if a program P never implements what is not specified in a specification S.

2. *Parallelization:* Using the divide & conquer approach that helps us to parallelize state sequences generation in a stratified way.

3. *A scalable technique:* The proposed approach may mitigate state explosion problem while generating state sequences.

Drawbacks:

1. *Consuming resources and not alleviation well enough:* Essentially, exploring the entire state space by JPF. This is quite expensive, as some execution may not be relevant to the properties to check. Especially, the proposed approach does not mitigate the state explosion well enough when increasing a large number of threads in concurrent programs.

2. *Manually semi-computation checking:* As soon as all state sequences have generated, we manually check it with specifications. It is manual and time-consuming to wait for checking. But we are confident to completely conquer these problems in our future work.

3. *Understanding specifications as well*: Generating state sequences from program P requires developers need to understand about specifications to make a correct format that is satisfied by the specification inputs.

6.2 Future Work

Up to now, our implementation can completely be testing a proper number of threads in concurrent programs with specifications. Because of time limitation, so based on the advantages and drawbacks. In future work, we would like to conduct more useful case studies to evaluate the performance of our proposed technique as well as improve our environment more scalability. Our plan is the following:

1. Integrating the current tool implemented in Java to the tool component [1] implemented in Maude, it makes our environment that can detect subtle flaws lurking in the real-time.

2. Accelerating state sequences generation from concurrent programs. For example, instead of the use of files system (disk) storage, it would be better to only use in-memory storage, which we anticipate would be feasible.

3. Finding out a good criterion that can be used to systematically choose a next state which should be explored at scheduling points for state sequences generation. It is possible to help us quickly reaching errors lurking in concurrent programs.

4. Lastly, we would like to conduct more case studies, it is most important to evaluate the performance of our tool as well as our proposed method.

Bibliography

[1] Vinay Arora, Rajesh Kumar Bhatia, and Maninder Singh. A systematic review of approaches for testing concurrent programs. *Concurrency Computat.: Pract. Exper.*, 28(5):1572–1611, 2016.

[2] Eman Alatawi, Harald Søndergaard, and Tim Miller. Leveraging abstract interpretation for efficient dynamic symbolic execution. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 619–624, Piscataway, NJ, USA, 2017. IEEE Press.

[3] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, January 2005.

[4] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[5] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[6] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. Divine 3.0 – an explicit-state model checker for multithreaded c & c++ programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 863–868, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Jiri Barnat, Vincent Bloemen, Alexandre Duret-Lutz, Alfons Laarman, Laure Petrucci, Jaco van de Pol, and Etienne Renault. *Parallel Model Checking Algorithms for Linear-Time Temporal Logic*, pages 457–507. Springer International Publishing, Cham, 2018.

[8] Zhan-Wei Hui and Song Huang. Md-art: A test case generation method without test oracle problem. In *Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs*, SCTDCP 2016, pages 27–34, New York, NY, USA, 2016. ACM.

[9] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *39th ICSE*, pages 266–277, 2017.

[10] Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. Quick verification of concurrent programs by iteratively relaxed scheduling. In *32nd ASE*, pages 776–781, 2017.

[11] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[12] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *36th PLDI*, pages 165–174, 2015.

[13] Qiuping Yi and Jeff Huang. Concurrency verification with maximal path causality. In *26th FSE/17th ESEC*, pages 366–376, 2018.

[14] Miguel Isabel. Conditional dynamic partial order reduction and optimality results. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 433–437, New York, NY, USA, 2019. ACM.

[15] Kazuiro Ogata and Kokichi Futatsugi. Simulation-based verification for invariant properties in the OTS/CafeOBJ method. In *Refine 2007*, ENTCS 201, pages 127–154. Elsevier, 2007.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude*. LNCS 4350. Springer, 2007.

[17] Canh Minh Do and Kazuhiro Ogata. Specification-based testing with simulation relations. In *31st International Conference on Software Engineering and Knowledge Engineering (31st SEKE)*, pages 107–112. KSI Research Inc., July 2019.

[18] Taro Kurita, Miki Chiba, and Yasumasa Nakatsugawa. Application of a formal specification language in the development of the "Mobile FeliCa"

IC chip firmware for embedding in mobile phone. In *FM 2008*, LNCS 5014, pages 425–429. Springer, 2008.

[19] Open source. Redis. `https://redis.io/`, 2009. [Online; accessed 05-August-2019].

[20] Open source. Rabbitmq. `https://www.rabbitmq.com/`, 2007. [Online; accessed 05-August-2019].

[21] Kazuhiro Ogata. Model checking designs with CafeOBJ – a contrast with a software model checker, Workshop on Formal Method and Internet of Mobile Things, ECNU, Shanghai, China, 2014.

Publications

[1] Canh Minh Do and Kazuhiro Ogata. Specification-based testing with simulation relations. In *31st International Conference on Software Engineering and Knowledge Engineering (31st SEKE)*, pages 107–112. KSI Research Inc., July 2019.