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

Title	形式検証ツールの並列化
Author(s)	DO, MINH CANH
Citation	
Issue Date	2022-09
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/18129
Rights	
Description	Supervisor:緒方 和博, 先端科学技術研究科, 博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

Parallelization of Formal Verification Tools

DO, Minh Canh

Supervisor: Kazuhiro Ogata

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]

September, 2022

Abstract

Today, software systems are used in various applications where failure is unacceptable. Among them are airplanes, utilities, telephones, banking & financial systems, commerce, logistics, appliances, houses, and securities. Very important systems, such as operating systems and the Internet that have been used as infrastructures, are typically in the form of concurrent/distributed programs. We are undeniable that the quality of software systems will affect the quality of our life more and more considerably. Therefore, the need for reliable software systems is critical. Model checking is one of the most successful achievements in computer science for hardware and software verification. However, there are still some challenges to tackle. One of them is the state space explosion problem, which can make it impossible to conduct model checking experiments. Many techniques have been proposed to alleviate the problem to some extent, but the problem still remains when dealing with large systems and often prevents model checking experiments from being carried out. Another challenge is to increase the running performance of model checking. One promising approach to this challenge is to parallelize model checking, which can make the best use of multicore architectures. In this thesis, we propose some techniques to mitigate the state space explosion problem (space challenge) and improve the running performance of model checking (time challenge) by parallelization for some formal verification tools. In summary, the thesis describes three non-trivial cases to demonstrate the proposed techniques: (1) parallelization of Java Pathfinder, a software model checker, for testing concurrent programs, (2) parallelization of Maude LTL model checker for checking leads-to properties, and (3) parallelization of Maude-NPA, a logical model checker, for cryptographic protocol analysis. Besides, we describe some shared techniques used for parallelization in this thesis and a generic approach to parallelizing tools used for formal methods.

Studies on testing concurrent programs have been conducted for nearly 40 years or even more. Compared to testing techniques for sequential programs, however, any testing techniques for concurrent programs do not seem mature enough. Moreover, many important software systems, such as operating systems, are in the form of concurrent programs. Therefore, testing techniques for concurrent programs must be worth studying so that they can be matured enough. We propose a specification-based testing technique for concurrent programs. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S. 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. Thus, we propose a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way. A tool is developed to support the proposed technique that can be processed naturally in parallel. Some experiments demonstrate that the proposed technique mitigates the state space explosion and improves the verification time, which cannot be achieved with the straightforward use of JPF.

Our research group has proposed the $L+1$-layer divide & conquer approach to leads-to model checking ($L+1$-DCA2L2MC), which is a new technique to mitigate the state space explosion in model checking. As shown by the name, $L+1$-DCA2L2MC is dedicated to leads-to properties. This thesis describes a parallel version of $L+1$-DCA2L2MC and a tool that supports it. In a temporal logic called UNITY designed by Chandy and Misra, the leads-to temporal connective plays an important role and many case studies have been conducted in UNITY, demonstrating that many systems requirements can be expressed as leads-to properties. Hence, it is worth dedicating to the properties. This thesis also reports on some experiments that demonstrate that the tool can increase the running performance of model checking. Counterexample generation is one of the main tasks in the tool that can be optimized to improve the running performance of the tool to some extent. This thesis then proposes a technique to generate all counterexamples at once that is based on the Tarjan algorithm, implemented in C++, and integrated into Maude, a programming/specification language based on rewriting logic, so that users can use it easily. Some experiments are conducted to demonstrate the power of the technique that can improve the running performance of the tool. Furthermore, layer configuration selection affects the running performance of the tool. Therefore, this thesis then proposes an approach to finding good layer configurations for the tool with an analysis tool that supports the approach. Some experiments are conducted to demonstrate the usefulness of the analysis tool as well as the approach for layer configuration selection.

With the emergence of the Internet and network-based services, many cryptographic protocols, also called security protocols, have been developed over decades to provide information security in an insecure network, such as confidentiality and authentication. The design of cryptographic protocols, such as authentication protocols, is difficult, error-prone, and hard to detect bugs. Therefore, it is important to have automated tools to verify some desired properties of cryptographic protocols. Maude-NPA is a formal verification tool for analyzing cryptographic protocols in the Dolev-Yao strand space model modulo an equational theory defining the cryptographic primitives. It starts from an attack state to find counterexamples or conclude that the attack concerned cannot be conducted by performing a backward narrowing reachability analysis. Although Maude-NPA is a powerful analyzer, its running performance can be improved by taking advantage of parallel and/or distributed computing when dealing with non-trivial protocols whose state space is huge. This thesis describes a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption are parallelized

at each layer. The tool supporting the parallel version has been implemented in Maude with a master-worker model. We report on some experiments of various kinds of protocols that demonstrate that the tool can increase the running performance of Maude-NPA by 44% on average for all non-trivial case studies experimented in which the number of states located at each layer is considerably large.

Keywords: testing concurrent programs; LTL model checking; cryptographic protocol analysis; Java Pathfinder(JPF); Maude; Maude-NPA; state space explosion; divide & conquer; master-worker; parallelization.

Acknowledgments

First and foremost I am extremely grateful to my supervisor, Professor Kazuhiro Ogata, for his invaluable advice, continuous support, and patience during my Ph.D. study at JAIST. His immense knowledge and plentiful experience have encouraged me all the time in my academic research and daily life. I feel so lucky to be a student in his laboratory. He cares so much about my work and gives me invaluable advice, comments, and great ideas to make my research keep on track. He has inspired me to become a good scientific researcher and motivated me to improve myself day by day and always look forward to research. Besides, he has provided a very good environment, which makes me feel free and high motivative for unlimited creativity in research. No words can describe what he has done for me and I am indebted to him.

Second, I would like to express special thanks to Associate Professor Adrián Riesco, Associate Professor Santiago Escobar, Professor Kunihiko Hiraishi, Professor Masashi Unoki, Professor Mizuhito Ogawa, Professor Toshiaki Aoki, Professor Tatsuhiro Tsuchiya, and Professor Shaoying Liu for their encouraging words and thoughtful, detailed comments, and suggestions that have been very important for me to improve my work as well as complete my thesis. Especially, Associate Professor Adrián Riesco and Associate Professor Santiago Escobar have supported me a lot in working with Maude and Maude-NPA on parallelization.

Third, I would like to express special thanks to the Doctoral Research Fellow (DRF) at JAIST, the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP19H04082, and JST SICORP Grant Number JPMJSC20C2, Japan for financial support during my Ph.D. study at JAIST.

Fourth, I would like to express my appreciation to my lab mates. Thank all of you for sharing wonderful moments and interesting ideas, not only in research but also in daily life. It is an unforgettable memory in my life. Besides, I also would like to express special thanks to my Japanese teachers: Atsushi Asamoto, Masako Tsutsui, and Yamaguchi Michiyo who have tried their best to teach me the Japanese language. I had a fun and unforgettable time in Japanese classes with friends and teachers that help me to have an enjoyable life in Japan.

Last but not least, my warm and heartfelt thanks go to my family for the tremendous support and hope they had given to me. Especially, I would like to express my greatest thanks to my wife and my children for always being by my side and understanding me to overcome the difficult time. Without their support, understanding, and encouragement, it would be impossible for me to complete my Ph.D. study at JAIST.

Contents

Abstract **i**

Acknowledgments **iv**

1 Introduction **1**
- 1.1 Testing for Concurrent Programs 2
- 1.2 Linear Temporal Logic Model Checking 4
- 1.3 Cryptographic Protocol Analysis 5
- 1.4 Contributions 6
- 1.5 Thesis Structure 7

2 Preliminaries **10**
- 2.1 State Machine 10
- 2.2 Simulation Relations 10
- 2.3 Kripke Structure 12
- 2.4 Büchi Automata 13
- 2.5 Rewriting Logic 15
- 2.6 Meta-programming in Maude 16

3 Related Work **18**
- 3.1 Testing Concurrent Programs 18
- 3.2 Linear Temporal Logic Model Checking 22
- 3.3 Cryptographic Protocol Analysis 24

4 Techniques to Parallelize Formal Verification Tools **27**
- 4.1 A Master-Worker Model 27
- 4.2 Dividing the Reachable State Space into Multiple Layers 28
- 4.3 Tackling Multiple Sub-State Spaces in Parallel 28
- 4.4 Caching to Avoid State Duplications 28
- 4.5 A Generic Approach to Parallelizing Tools Used for Formal Methods 29

5 Parallel Specification-based Testing for Concurrent Programs 31
5.1 Specification-based Concurrent Program Testing with a Simulation Relation . . 31
5.2 State Sequence Generation from Concurrent Programs 33
5.2.1 Java Pathfinder (JPF) 33
5.2.2 Generating State Sequences by JPF 33
5.3 A Divide & Conquer Approach to Generating State Sequences 36
5.4 A Divide & Conquer Approach to Testing Concurrent Programs and Its Support Tool 39
5.5 Case Studies 43
5.5.1 Alternating Bit Protocol (ABP) 43
5.5.2 Revised CloudSync 50
5.5.3 NSLPK 54
5.5.4 Original CloudSync 56
5.5.5 Threats to Bug Detection 59
5.6 Extending Our Technique to Test Concurrent Programs with JPF 60
5.7 Limitations 61
5.8 Summary 62

6 A Parallel Version of a Support Tool for the Divide & Conquer Approach to Leads-to Model Checking 63
6.1 Introduction 64
6.2 The $L+1$-Layer Divide & Conquer Approach to Leads-to Model Checking . . . 66
6.3 Parallel $L+1$-DCA2L2MC and Its Tool Support 69
6.4 Outline of Parallel $L+1$-DCA2L2MC 74
6.5 Experiments 81
6.6 All Counterexample Generation 84
6.7 Layer Configuration Selection 92
6.8 Scalability Testing 101
6.9 Limitations 103
6.10 Summary 104

7 Parallel Maude-NPA for Cryptographic Protocol Analysis 105
7.1 Introduction 105
7.2 Maude 106
7.3 Maude-NPA 114
7.3.1 A Protocol Specification in Maude-NPA 114
7.3.2 How Maude-NPA Works 120
7.4 Parallel Maude-NPA and Its Tool Support 122
7.4.1 How to Parallelize Maude-NPA 122

7.4.2 Job Scheduling by the Master 126
7.4.3 Job Handing by Workers 134
7.5 Experiments 136
7.6 Summary 142

8 Conclusions and Future Work **143**
8.1 Conclusions 143
8.2 Future Work 145

Bibliography **148**

First-author Publications **165**

The Other Co-author Publications **166**

List of Figures

2.1 A simulation relation from M_C to M_A 11

5.1 Specification-based concurrent program testing with a simulation relation 32
5.2 A way to generate state sequences with JPF 35
5.3 A divide & conquer approach to generating state sequences 37
5.4 The architecture of a tool supporting the proposed technique 40
5.5 A state of ABP 44
5.6 Verification time for ABP programs with various numbers of workers 50
5.7 gotval transition 51
5.8 updated1 transition 52
5.9 updated2 transition 52
5.10 gotoidle transition 52
5.11 modval transition 57
5.12 Backward and cross transitions 59
5.13 A state sequence in a layer l 60

6.1 Split of the reachable state space into $L + 1$ layers 67
6.2 The reachable state space of TAS 76
6.3 Three layers of TAS reachable state space 77
6.4 Strong scaling speedup for TAS 103

7.1 A fragment of one possible execution trace of the client-server system 111
7.2 The execution trace of the system with a server and a meta-interpreter for incrementing a natural number. 113
7.3 Conducting the backward narrowing in step (1) from the states at layer l in parallel. 123
7.4 Conducting the transition subsumption in step (2.1) for the states at layer $l + 1$ in parallel. 124
7.5 Conducting the transition subsumption in step (2.1) for the states at layer $l + 1$ in parallel (continuously). 124
7.6 Conducting the transition subsumption in step (2.2) for states with history states at layer $l + 1$ in parallel. 125

List of Tables

5.1 Experimental data for ABP program in which one lock is used 48
5.2 Experimental data for ABP program in which two locks are used 48
5.3 Experimental data for ABP program with various numbers of workers 49
5.4 Experimental data for Revised CloudSync 53
5.5 Experimental data for NSLPK 57
5.6 Experimental data for Original CloudSync 58

6.1 Experimental data with 2GB memory restriction 81
6.2 The use of the shared and local caches 82
6.3 Experimental data for Maude LTL model checker, $L+1$-DCA2L2MC, and Parallel $L+1$-DCA2L2MC 83
6.4 Experimental data for parallelizing the model checking in the final layer only . . 84
6.5 Experimental data for generating all counterexamples at once in Parallel $L+1$-DCA2L2MC 91
6.6 Experimental data for only generating states up to the final layer in $L+1$-DCA2L2MC with all counterexample generation 91
6.7 Parallel $L+1$-DCA2L2MC with different layer configurations 100
6.8 Analysis time to find good layer configurations 100
6.9 Strong scaling data for our tool with TAS 102

7.1 Experimental results of Maude-NPA and Par-Maude-NPA-2. 137
7.2 Experimental results of Maude-NPA and Par-Maude-NPA-2. 138
7.3 The effectiveness of the use of meta-interpreters and the parallelization of step (2.1) and step (2.2). 139
7.4 Par-Maude-NPA-1 and Par-Maude-NPA-2 with various numbers of workers. . . 140

Chapter 1

Introduction

Today, software systems are used in various applications where failure is unacceptable. Among them are airplanes, utilities, telephones, banking & financial systems, commerce, logistics, appliances, houses, and securities. We are undeniable that the quality of software systems will affect the quality of our life more and more considerably. With rapidly expanding software systems as well as their complexity nowadays, the likelihood of subtle bugs lurking in software systems is much greater. Some of these bugs may cause tragic loss of money, time, or even human life. Software testing has been the standard technique for identifying software bugs for decades. However, the purpose of software testing is not to prove that there are no bugs in software systems but rather to find them. As E.W. Dijkstra has remarked [1] “Program testing can be used to show the presence of bugs, but never to show their absence.” Although software testing is very useful to detect bugs in software systems, it is insufficient to make software systems reliable, especially, for testing concurrent/distributed systems where non-nondeterminism is naturally presented. To complement testing techniques, formal methods are one promising way to make software systems more trustworthy by using mathematically based languages, techniques, and tools for specifying and verifying systems.

Specification is the process of describing a system and its desired properties, which uses a language with a mathematically defined syntax and semantics. Based on the formal specification, formal verification is conducted to verify that the system satisfies its desired properties. One of the most successful approaches to formal verification in the past few decades is model checking [2]. That is why Edmund M. Clarke, together with E. Allen Emerson and Joseph Sifakis, was honored with the 2007 A. M. Turing Award[1] for their role in developing model checking into a highly effective verification technology. Model checking is an automatic verification technique for verifying that a finite-state system satisfies some desired properties by exploring the entire reachable state space and checking whether there are any violated states. If so, it produces counterexamples, which usually represent subtle bugs in the system. However, there are still some challenges to tackle. One of them is the state space explosion problem,

[1] `https://amturing.acm.org/award_winners/clarke_1167964.cfm`

the most annoying one, which can make it impossible to conduct model checking experiments. Many techniques have been proposed to alleviate the problem. Among them are partial order reduction [3], abstraction [4, 5, 6], ordered binary decision diagrams (OBDD) [7], and compositional reasoning [8, 9]. Such techniques alleviate the problem to some extent, but the problem still remains when dealing with large systems and often prevents some model checking experiments from being carried out. Another challenge is to increase the running performance of model checking. Because computer systems continue to scale, e.g., individual machines have more memory, more core, and larger disk capacity, one promising approach to this challenge is to parallelize model checking [10], which can make the best use of multi-core architectures.

This thesis proposes some techniques to mitigate the state space explosion problem (space challenge) and improve the running performance of model checking (time challenge) to some extent by parallelization. The thesis describes three non-trivial cases to demonstrate the proposed techniques: (1) parallelization of Java Pathfinder, a software model checker, for testing concurrent programs, (2) parallelization of Maude LTL model checker for checking leads-to properties, and (3) parallelization of Maude-NPA, a logical model checker [11], for cryptographic protocol analysis. We select three formal verification tools because of the following reasons: Java Pathfinder (JPF) [12] is one of the most popular software model checkers for Java programs, which is developed by NASA; Maude LTL model checker [13] is comparable to Spin [14] in terms of both running performance and memory consumption [15]; and Maude-NPA [16] is one formal verification tool dedicated to security protocol analysis that is very crucial for our current and future open network world, such as the Internet.

1.1 Testing for Concurrent Programs

Studies on testing concurrent programs [17] have been conducted for nearly 40 years or even more. Compared to testing techniques for sequential programs, however, any testing techniques for concurrent programs do not seem mature enough. Moreover, many important software systems, such as operating systems, are in the form of concurrent programs. 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. The thesis focuses on approach (2) mainly because P is a concurrent program and then could produce many different executions due to the inherent nondeterminacy of P, which often leads to subtle bugs in the program.

In the correct-by-construction system or software development, a system is initially specified in a very abstract formal specification and then incrementally developed to a concrete formal specification through a sequence of refinement steps. In each refinement step, we add details about data structures and algorithms so that the final formal specification is closer to the implementation or program from which the program can be implemented by human programmers or generated by automatic code generators [18, 19, 20]. Among formal methods in which stepwise refinement plays a crucial role are Vienna Development Method (VDM) (or VDM++) [21], Z method [22], Abstract State Machine (ASM) [23], B method [24], and Event-B [25]. The final specification can be verified by verifying each individual refinement step, while the program generated needs to be verified by proving the very final refinement step from the final formal specification to the program implemented or generated based on the final formal specification or the automatic code generator, respectively. It is very hard to verify the correctness of automatic code generators [26] and it is also hard to verify the very final refinement step partly because we need to have full precise formal semantics of programming languages. However, we need to have a reasonably good technique that can be used to check the program implemented or generated against the final formal specification. We propose a specification-based testing technique in this thesis that can be regarded as a complement to the very last step in correct-by-construction software development to guarantee the correctness of the program with the specification.

We suppose that a formal specification S is specified in Maude [13] while a program P developed based on the formal specification is implemented in Java. Java Pathfinder (JPF) [12] is an extensible model checker for Java programs so that we can interact with JPF while model checking a program. Hence, we use JPF to generate state sequences from P. Maude is equipped with reflective programming (meta-programming) facilities, making it possible to check whether a state sequence can be accepted by a formal specification [27], and so we use Maude to check if such state sequences are accepted by S. Most model checkers suffer from the notorious state space explosion problem and JPF is no exception. JPF often encounters the notorious state space explosion while generating state sequences even without checking any property violations. Because there could be a huge number of different states reachable from an initial state, which could make a huge number of different state sequences generated due to the inherent nondeterminacy of concurrent programs. In addition, a whole big heap mainly constitutes one state in a program under test by JPF. Thus, we propose a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way, which can be processed naturally in parallel, to mitigate the state space explosion as well as increase the running performance of model checking.

1.2 Linear Temporal Logic Model Checking

To verify (concurrent) systems by model checking, formal specification involves two specification languages: one is system specifications and the other one is property specifications. In linear temporal logic (LTL) model checking, a system specification formalizes the behavior of the system as a state transition graph also known as a Kripke structure K, and a property specification is expressed as a temporal formula φ. An LTL model checker decides whether the Kripke structure K is a model of the formula φ, which is actually a linear temporal formula. If so, we denote $K \models \varphi$. Otherwise, we denote $K \not\models \varphi$ and a counterexample that witnesses the violation of φ by K is returned. There are several approaches to the LTL model checking problem, such as the automata-theoretic approach [28, 29], BDD-based approach [30, 31], and SAT/SMT-based approach [32, 33, 34]. Many model checkers are implemented based on such approaches, such as SPIN [14], NuSMV [35], TCBMC [36], and Lazy-CSeq [37, 38].

In hardware and software verification, model checking [2] is one of the most advanced methods for automatic formal verification. However, the most challenge in model checking is the state space explosion problem, which can make it impossible to conduct model checking experiments. Another challenge is to increase the running performance of model checking. One promising approach to this challenge is to parallelize model checking, which can make the best use of multicore architectures. Any LTL model checker suffers from the state space explosion, making the running performance of model checking degrade. Hence, in this thesis, we focus on improving the running performance of an LTL model checker by parallelization.

Our research group has proposed the $L+1$-layer divide & conquer approach to leads-to model checking ($L+1$-DCA2L2MC) [39] that is a new technique to mitigate the state explosion problem in model checking. As shown by the name, $L+1$-DCA2L2MC is dedicated to leads-to properties whose informal description is that whenever something becomes true, something else will eventually become true. Chandy and Misra [40] designed a temporal logic called UNITY in which the leads-to temporal connective plays an important role and demonstrated that many important systems requirements can be expressed as leads-to properties. Besides, Dwyer et al. [41] showed some statistics on the usage distribution of the various patterns in property specifications in which the leads-to property (or the response pattern) had the highest proportion. Therefore, it is worth focusing on leads-to properties.

$L+1$-DCA2L2MC has been demonstrated as one promising approach to mitigating the state space explosion in model checking for leads-to properties. However, the running performance of $L+1$-DCA2L2MC can be improved by taking advantage of parallel and/or distributed computing from which we can mitigate the state space explosion problem (space challenge) and improve the running performance of model checking (time challenge) as well. In this thesis, we demonstrate that $L+1$-DCA2L2MC can be naturally parallelized by describing a parallel version of $L+1$-DCA2L2MC and a tool that supports it. Counterexample generation is one main task in the tool that can be optimized to improve the running performance of the tool.

We propose a technique to generate all counterexamples at once in model checking with a new model checker. Furthermore, layer configuration selection affects the running performance of the tool. Therefore, we propose an approach to finding good layer configurations for the tool with an analysis tool.

1.3 Cryptographic Protocol Analysis

Cryptographic protocols (or security protocols), such as Transport Layer Security (TLS) [42, 43], are extremely important so as to implement secure, safe, and reliable communication over an open network, such as the Internet. It is also extremely hard to design such protocols and detect flaws lurking in them [44]. Some protocols designed by security experts had flaws and it took time to discover flaws in such protocols after their publication. For example, Lowe found a flaw [45] in the Needham-Schroeder public key (NSPK) authentication protocol 17 years later since NSPK was designed and published by Needham and Schroeder [46], security experts. TLS is an improved version of SSL, which is commonly used in securing HTTPS connections. Since SSL version 1.0 was first introduced in 1994, many attacks were quickly discovered, and so SSL versions 2.0 and 3.0 were then proposed in 1995 and 1996, respectively. By 2015, SSL 3.0 became deprecated and TLS was then used as an upgrade version because of some attacks [47], such as the POODLE attack [48]. Therefore, techniques and tools that help researchers and engineers find out flaws lurking in cryptographic protocols and/or formally verify that such protocols enjoy some desired properties are indispensable. Tools dedicated to formal verification of such protocols have been developed. Among them are Athena [49], ProVerif [50], Avispa [51], Scyther [52], Tamarin [53], DEEPSEC [54], Verifpal [55], and CPSA [56]. Because the running performance is crucial, such tools adopt several optimization techniques (e.g., the partial order reduction [57]) like those used by general-purpose model checkers, such as Spin [14] and NuSMV [35]. Except for DEEPSEC, which supports protocols with a bounded number of sessions, none of the dedicated formal verification tools for cryptographic protocols have been parallelized.

Maude-NPA is a formal verification tool for analyzing cryptographic protocols. It uses a backward narrowing reachability analysis modulo an equational theory and the Dolev-Yao strand space model [58, 59], which gives intruders capable of intercepting, modifying, and injecting messages to impersonate other protocol principals. Narrowing is a generalization of term rewriting that allows logical variables in terms and replaces pattern matching by unification. Hence, it supports symbolic execution. The backward narrowing reachability analysis starts from a final insecure state, an attack state, to determine whether it is reachable from an initial state, which has no further backward steps. If so, the attack concerned from the attack state can be conducted for the protocol under verification; otherwise, the attack cannot. The advantage of Maude-NPA is that it supports protocols with an unbounded session model

thanks to symbolic execution, and different equational theories; as a counterpart, these theories often lead to a bigger state space that requires more time to conduct formal verification. Although some techniques were devised to reduce the state space and improve the running performance, such as grammar-based techniques, giving priority to input messages in strands, early detection of inconsistent states (never reaching an initial state), a relation of transition subsumption (to discard transitions and states already being processed in another part of the search space), and the super lazy intruder (to delay the generation of substitution instances as much as possible) [60, 61], the state space explosion problem is inevitable in some cases, degrading the running performance. Therefore, it is worth improving the running performance of Maude-NPA. In this thesis, we describe a parallel version of Maude-NPA that can largely improve the running performance of Maude-NPA.

1.4 Contributions

In summary, we focus on the parallelization of three formal verification tools: (1) parallelization of Java Pathfinder for testing concurrent programs, (2) parallelization of Maude LTL model checker for checking leads-to properties, and (3) parallelization of Maude-NPA for cryptographic protocol analysis. Besides, we describe some shared techniques used for parallelization in (1), (2), and (3), and a generic approach to parallelizing tools used for formal methods.

For testing concurrent programs, we make the following contributions:

- A new testing technique for concurrent programs to check the correctness of a program with a specification that can also be used to complement the very last step in correct-by-constructions software development. For programmers who prefer developing a program based on a specification from scratch, our technique is fruitful to be used to verify whether the program conforms to the specification.

- A divide & conquer approach to testing concurrent programs that can mitigate the state space explosion problem and be naturally parallelized to improve the running performance of model checking. A tool is developed to support the technique that is written in Java with a master-worker model.

- Some experiments are conducted to demonstrate the usefulness of the tool as well as the approach. The tool and case studies are publicly available at the webpage.[2]

For LTL model checking, we make the following contributions:

- A parallel version of $L+1$-DCA2L2MC and a support tool that is dedicated to leads-to model checking. The tool is written in Maude with a master-worker model.

[2] `https://github.com/canhminhdo/spec-based`

- Counterexample generation technique that can generate all counterexamples at once and a new model checker to support the technique, which is implemented in C++ and integrated into Maude, to improve the running performance of our tool. The source code of the new model checker is publicly available at the webpage.[3]

- An approach to finding good layer configurations for $L + 1$-DCA2L2MC technique and an analysis tool that supports the approach.

- Several experiments are conducted to demonstrate the usefulness of each proposed technique/approach. All source code files and case studies are publicly available at the webpage.[4]

For cryptographic protocol analysis, we make the following contributions:

- A parallel version of Maude-NPA in which the backward narrowing and the transition subsumption are conducted in parallel at each layer.

- A tool is developed to support the parallel version that is written in Maude using meta-interpreters with a master-worker model.

- Some experiments on various kinds of protocols are reported to demonstrate that the tool can increase the running performance of Maude-NPA by 44% on average for all non-trivial case studies experimented. All source code files and case studies are publicly available at the webpage.[5]

1.5 Thesis Structure

We organize the structure of this thesis into eight chapters. We summarize each chapter as follows:

- **Chapter 1** introduces model checking and summarizes the problems of model checking, including the state space explosion problem and improving the running performance of model checking. This chapter introduces and addresses the problems in testing concurrent programs, LTL model checking, and cryptographic protocol analysis by parallelization followed by our contributions and the thesis structure.

- **Chapter 2** provides preliminaries to this thesis. This chapter introduces some background to help readers understand our work, including State Machine, Simulation Relations, Kripke Structure, Büchi Automata, Rewriting Logic, and Meta-programming in Maude.

[3]`https://github.com/canhminhdo/Maude`

[4]`https://github.com/canhminhdo/parallel-dca2l2mc`

[5]`https://github.com/canhminhdo/parallel-maude-npa`

- **Chapter 3** investigates some advancements in parallel model checking for testing concurrent programs, LTL model checking, and cryptographic protocol analysis. For testing concurrent programs, bounded model checking, abstraction refinement, and some work in correct-by-construction software development, such as High-Level Language (HLL) to translate Event-B model checks to C code, EventB2Java to generate JML-annotate Java programs from Even-B models, are mentioned. For LTL model checking, DiVinE, Garakabu2, a multicore extension of SPIN, assume-guarantee verification, and parallel CTL model checking are mentioned. For cryptographic protocol analysis, other analysis tools than Maude-NPA are mentioned, such as Scyther, Tamarin, Proverif, and DEEPSEC.

- **Chapter 4** describes techniques that are used to parallelize three formal verification tools in this thesis. Besides, we describe a generic approach to parallelizing tools used for formal methods.

- **Chapter 5** proposes a new testing technique for testing concurrent programs that is a specification-based testing. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S. In the proposed technique, S is specified in Maude and P is implemented in Java. JPF and Maude are then used to generate state sequences of P and check if such state sequences are accepted by S, respectively. This chapter then proposes a divide & conquer approach to generating state sequences from P and checking if such state sequences are accepted by S on the fly in a stratified way to avoid the state space explosion while generating state sequences. A tool is implemented in Java to support the proposed technique that is also described in detail. Some case studies are given to demonstrate the usefulness of the tool as well as the proposed technique. This chapter also describes how to extend the technique to test concurrent programs with JPF without checking whether execution sequences generated from Java programs can be accepted by Maude specifications.

- **Chapter 6** introduces $L+1$-DCA2L2MC that is a new technique to mitigate the state space explosion in model checking and then proposes a parallel version of the $L+1$-DCA2L2MC and its support tool. Counterexamples generation is one of the main tasks in the tool that can be optimized to improve the running performance of the tool. This chapter then proposes a technique to generate all counterexamples at once and a new model checker that supports the technique. Furthermore, layer configuration selection affects the running performance of the tool, this chapter then proposes an approach to finding good layer configurations for the tool and an analysis tool that supports the approach. Several experiments are conducted to demonstrate the usefulness of each proposed technique/approach.

- **Chapter 7** introduces Maude-NPA for cryptographic protocol analysis and describes how Maude-NPA works in detail from which we propose a parallel version of Maude-NPA and

its support tool. Several experiments on various kinds of protocols are conducted to demonstrate the usefulness of the tool for cryptographic protocol analysis.

- **Chapter 8** summarizes the main contributions of the thesis and mentions several lines of our future work.

Chapter 2

Preliminaries

This chapter presents some backgrounds to help readers understand our work, including State Machine, Simulation Relations, Kripke Structure, Büchi Automata, Rewriting Logic, and Meta-programming in Maude.

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 $s \to_M s'$. Let $\to_M^*$ be the reflexive and transitive closure of $\to_M$. 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 $s_i \to_M^* s_{i+1}$ for each $i = 0, \ldots, n-1$. If that is the case, it is said 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 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 $s_C \to_{M_C} s'_C$, there exists $s'_A \in R_A$ such that $r(s'_C, s'_A)$ and $s_A \to_{M_A}^* s'_A$ [62] (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 $s_C \in R_{M_C}$ [62]. The theorem makes it possible to verify that p_C is invariant w.r.t. M_C by proving that p_A is

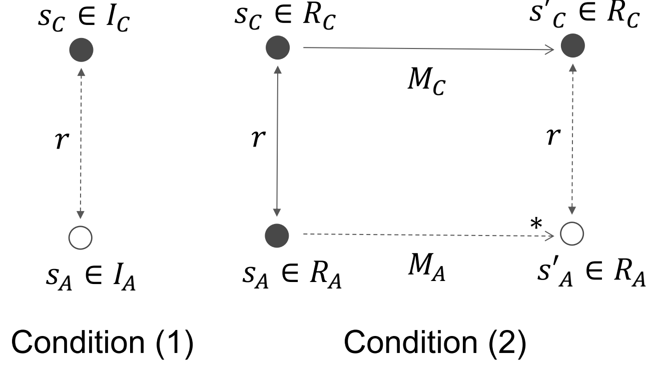


Figure 2.1: A simulation relation from M_C to M_A

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 work. 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 [13], 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 $\{(\text{lock} : b)\ (\text{pc}[\text{p1}] : l_1)\ (\text{pc}[\text{p2}] : l_2)\ (\text{pc}[\text{p3}] : l_3)\}$, 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 Kripke Structure

A Kripke structure K is a tuple $\langle S, I, T, A, L\rangle$ [2], where S is a set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a left-total binary relation over S, A is a set of atomic propositions and L is a labeling function whose type is $S \to 2^A$. Each element $(s, s') \in T$ is called a state transition from s to s' and may be denoted $s \to_K s'$ or $s \to s'$. For a state $s \in S$, $L(s)$ is the set of atomic propositions that hold in s. A path π is an infinite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots$ such that $s_i \to_K s_{i+1}$ for each i. We use the following notations for paths: $\pi^i \triangleq s_i, s_{i+1}, \ldots$; $\pi_i \triangleq s_0, \ldots, s_i, s_i, s_i, \ldots$; and $\pi(i) \triangleq s_i$. π^i is a postfix of π. π_i is a path obtained by adding $\pi(i)$, the ith state of π, to a prefix of π at the end infinitely many times. Let $\mathcal{P}$ be the set of all paths. π is called a computation if $\pi(0) \in I$. Let $\mathcal{C}$ be the set of all computations.

The syntax of a formula φ in LTL for K is as follows:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\, \varphi \mid \varphi_1\, \mathcal{U}\, \varphi_2$$

where $p \in A$. Let $\mathcal{F}$ be the set of all formulas in LTL for K. For an arbitrary path $\pi \in \mathcal{P}$ of K and an arbitrary LTL formula $\varphi \in \mathcal{F}$ of K, $K, \pi \models \varphi$ is inductively defined as follows:

- $K, \pi \models \top$,
- $K, \pi \models p$ if and only if $p \in L(\pi(0))$,
- $K, \pi \models \neg\varphi_1$ if and only if $K, \pi \not\models \varphi_1$,

- $K, \pi \models \varphi_1 \wedge \varphi_2$ if and only if $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$,
- $K, \pi \models \bigcirc \varphi_1$ if and only if $K, \pi^1 \models \varphi_1$,
- $K, \pi \models \varphi_1 \,\mathcal{U}\, \varphi_2$ if and only if there exists a natural number i such that $K, \pi^i \models \varphi_2$ and for all natural numbers $j < i$, $K, \pi^j \models \varphi_1$,

where φ_1 and φ_2 are LTL formulas. Then, $K \models \varphi$ if and only if $K, \pi \models \varphi$ for each computation $\pi \in \mathcal{C}$ of K. $\bigcirc$ and $\mathcal{U}$ are called the next temporal connective and the until temporal connective, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond \varphi \triangleq \top \,\mathcal{U}\, \varphi$, $\Box \varphi \triangleq \neg(\Diamond \neg\varphi)$, and $\varphi_1 \leadsto \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond \varphi_2)$. $\Diamond$, $\Box$, and $\leadsto$ are called the eventually temporal connective, the always temporal connective, and the leads-to temporal connective, respectively. LTL formulas that do not have any temporal connectives at all are called state propositions in this thesis.

There are multiple possible ways to express states. In this work, a state is expressed as a braced associative-commutative (AC) collection of name-value pairs, where a name may have parameters. The order of elements is irrelevant in AC collections, such as sets. AC collections are called soups and name-value pairs are called observable components. That is, a state is expressed as a braced soup of observable components. The juxtaposition binary operator that is associative and commutative is used as the constructor of soups. There are multiple possible ways to specify state transitions. In this thesis, rewrite rules are used to specify them. Concretely, we use Maude [13], a programming/specification language based on rewriting logic to specify Kripke structures or systems formalized as Kripke structures. Maude makes it possible to specify complex systems flexibly and is also equipped with an LTL model checker.

2.4 Büchi Automata

A finite automaton $\mathcal{A}$ over finite words is a tuple $\langle \Sigma, Q, \Delta, Q^0, F \rangle$, where Σ is a finite alphabet that is a finite set of letters, Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $Q^0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final (accepting) states. Let Σ^* be the set of finite words over Σ, including an empty word. A *run* of $\mathcal{A}$ over a word $v \in \Sigma^*$ of length $|v|$ is a function $p : \{0, 1, \ldots, |v|\} \to Q$ such that the first state is the initial state $p(0) \in Q^0$ and states are related by transition relation $\forall 0 \leq i \leq |v| \cdot (p(i), v(i), p(i+1)) \in \Delta$, where $v(i)$ is the ith letter of v. A run can be interpreted as a finite state sequence in $\mathcal{A}$ from $p(0)$ to the state $p(|v|)$ such that if the sequence is regarded as a graph, the edges are labeled with the letters in v. A run is accepting if it ends in an accepting state: $p(|v|) \in F$. An accepting run is also called a computation in $\mathcal{A}$. We say that $\mathcal{A}$ accepts v if and only if there exists an accepting run of $\mathcal{A}$ on v. The language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ is the set of all words in Σ^* accepted (or recognized) by $\mathcal{A}$. A set of words is *regular* if it is the language of a finite automaton. An automaton is *deterministic*

if the transition relation is deterministic for every letter in the alphabet: $\forall q, q', q'' \in Q, \forall a \in \Sigma$ such that if $(q, a, q') \in \Delta \wedge (q, a, q'') \in \Delta$, then $q' = q''$. Otherwise, it is *non-deterministic*.

Büchi automata are finite automata over infinite words that have the same structure as automata on finite words, but the notion of acceptance is different. Let Σ^ω be the set of infinite words over Σ such that each infinite word contains some characters that appear infinitely often. Büchi automata recognize words from Σ^ω (not Σ^*). A run p of a Büchi automaton $\mathcal{A}$ is over an infinite word $v \in \Sigma^\omega$. The domain of the run is the set of all natural numbers. Let $\mathit{inf}(p)$ be the set of states that appear infinitely often in p: $\mathit{inf}(p) = \{q \mid \forall i \in N, \exists j \geq i, p(j) = q\}$. p must have a cycle because of a finite alphabet and q must appear in such a cycle. A run p is accepting (Büchi accepting) if and only if $\mathit{inf}(p) \cap F \neq \emptyset$. A set of infinite words is ω-regular if and only if it is recognizable by a Büchi automaton.

Büchi automata are closed under projection, union, intersection, and complement [63]. For example, given two Büchi automata $\mathcal{B}_1 = \langle \Sigma, Q_1, \Delta_1, Q_1^0, F_1 \rangle$ and $\mathcal{B}_2 = \langle \Sigma, Q_2, \Delta_2, Q_2^0, F_2 \rangle$ such that all states of $\mathcal{B}_1$ are accepting, which is the special case of the intersection of two automata concerned in this thesis. We define $\mathcal{B}_\cap = \langle \Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2 \rangle$, where $((s_1, s_2), a, (s_1', s_2') \in \Delta'$ if and only if $(s_i, a, s_i') \in \Delta_i, i = 1, 2$. Then, $\mathcal{L}(\mathcal{B}_\cap) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Given a system in form of a Kripke structure K that is $\langle S, I, T, A, L \rangle$, we can transform the Kripke structure $\langle S, I, T, A, L \rangle$ into the Büchi automaton $\mathcal{A} = \langle \Sigma, S \cup \{l\}, \Delta, \{l\}, S \cup \{l\} \rangle$, where

- l is a dummy state regarded as the sole initial state,
- $\Sigma = 2^A$,
- $(l, \alpha, s) \in \Delta$ if and only if $s \in I$ and $\alpha = L(s)$,
- $(s, \alpha, s') \in \Delta$ if and only if $(s, s') \in T$ and $\alpha = L(s')$.

Every state is accepting in the Büchi automaton. Therefore, we only consider the special case of the intersection of the two automata described above. Given an LTL formula φ, we can build a corresponding Büchi automaton [64] $\mathcal{S} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$, where

- $\Sigma = 2^{Prop}$, where $Prop$ is a set of atomic propositions from which φ is built,
- $|Q| \leq 2^{O(|\varphi|)}$, where $|\varphi|$ is the length of the formula,

such that $\mathcal{L}(\mathcal{S})$ is exactly the set of computations satisfying the formula φ. The system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$, meaning that each computation of the system $\mathcal{A}$ is among the allowed computations of the specification $\mathcal{S}$. Alternatively, let $\overline{\mathcal{L}(\mathcal{S})}$ be the language $\Sigma^\omega - \mathcal{L}(\mathcal{S})$, then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$ if and only if $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$, meaning that no computation of $\mathcal{A}$ is prohibited by $\mathcal{S}$. Verifying the system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ becomes the emptiness problem in the product automaton of the system automaton and the complement

of the specification automaton. If the product automaton is not empty, any computation in it corresponds to a counterexample. Counterexamples are always of the form uv^ω, where u and v are finite words.

2.5 Rewriting Logic

We follow the classical notation and terminology from [65] for term rewriting and from [66, 67] for rewriting logic and order-sorted notions. We assume an *order-sorted signature* Σ with a finite poset of sorts $(\mathbf{S}, \leq)$ and a finite number of function symbols. We furthermore assume that: (i) each connected component in the poset ordering has a top sort, and for each $\mathbf{s} \in \mathbf{S}$ we denote by $[\mathbf{s}]$ the top sort in the component of $\mathbf{s}$; and (ii) for each operator declaration $f : \mathbf{s_1} \times \ldots \times \mathbf{s_n} \to \mathbf{s}$ in Σ, there is also a declaration $f : [\mathbf{s_1}] \times \ldots \times [\mathbf{s_n}] \to [\mathbf{s}]$. We assume an $\mathbf{S}$-sorted family $\mathcal{X} = \{\mathcal{X}_\mathbf{s}\}_{\mathbf{s} \in \mathbf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathbf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathbf{s}$ is the set of terms of sort $\mathbf{s}$, and $\mathcal{T}_{\Sigma,\mathbf{s}}$ is the set of ground terms of sort $\mathbf{s}$. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding term algebras. The set of positions of a term t is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. The root of a term is Λ. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the subterm $t|_p$ in t replaced by u. A *substitution* σ is a sorted mapping from a finite subset of $\mathcal{X}$, written $Dom(\sigma)$, to $\mathcal{T}_\Sigma(\mathcal{X})$. The set of variables introduced by σ is $Ran(\sigma)$. The identity function substitution is id. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The restriction of σ to a set of variables V is $\sigma|_V$.

A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathbf{s}$ for some sort $\mathbf{s} \in \mathbf{S}$. Given Σ and a set E of Σ-equations such that $\mathcal{T}_{\Sigma,\mathbf{s}} \neq \emptyset$ for every sort $\mathbf{s}$, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [67]). We assume that $\mathcal{T}_{\Sigma,\mathbf{s}} \neq \emptyset$ for every sort $\mathbf{s}$. The *Esubsumption* order on terms $\mathcal{T}_\Sigma(\mathcal{X})_\mathbf{s}$, written $t \preccurlyeq_E t'$ (meaning that t' is more general than t), holds if $\exists \sigma : t =_E \sigma(t')$. The E-renaming equivalence on term $\mathcal{T}_\Sigma(\mathcal{X})_\mathbf{s}$, written $t \approx_E t'$, holds if $t \preccurlyeq_E t'$ and $t' \preccurlyeq_E t$. We extend $=_E$, $\approx_E$, and $\preccurlyeq_E$ to substitutions in an expected way. An *E-unifier* for a Σ-equation $t = t'$ is a substitution σ s.t. $\sigma(t) =_E \sigma(t')$. A *complete* set of of E-unifiers of an equation $t = t'$ is written $CSU_E(t = t')$. We say $CSU_E(t = t')$ is *finitary* if it contains a finite number of E-unifiers. This notion can be extended to several equations, written $CSU_E(t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n)$.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathbf{s}$ for some sorts $\mathbf{s} \in \mathbf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. A *topmost rewrite theory* is a rewrite theory s.t. for each $l \to r \in R, l, r \in \mathcal{T}_\Sigma(\mathcal{X})_\mathbf{State}$ for a top sort **State**, $r \notin \mathcal{X}$, and no operator in Σ has **State** as an argument sort. The rewriting relation $\to_R$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \xrightarrow{p}_R t'$ (or $\to_R$) if $p \in Pos_\Sigma(t), l \to r \in R, t|_p = \sigma(l)$, and $t' = t[\sigma(r)]_p$ for some σ. The relation $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$. Note that $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\to_{R/E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \to_{R/E} [t']_E$ iff $t \to_{R/E} t'$. When $\mathcal{R} = (\Sigma, E, R)$ is a topmost rewrite theory

we can safely restrict ourselves [68] to the rewriting relation $\to_{R,E}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$, where $t \xrightarrow{\Lambda}_{R,E} t'$ (or $\to_{R,E}$) if $l \to r \in R, t =_E \sigma(l)$, and $t' = \sigma(r)$. Note that $\to_{R,E}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ induces a relation $\to_{R,E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \to_{R,E} [t']_E$ iff $\exists w \in \mathcal{T}_{\Sigma}(\mathcal{X})$ s.t. $t \to_{R,E} w$ and $w =_E t'$.

The narrowing relation $\rightsquigarrow_R$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is $t \overset{p,\sigma}{\rightsquigarrow}_R t'$ (or $\overset{\sigma}{\rightsquigarrow}_R, \rightsquigarrow_R$) if $p \in Pos_{\Sigma}(t)$, $l \to r \in R, \sigma \in CSU_{\emptyset}(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Assuming that E has a finitary and complete unification algorithm, the narrowing relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is $t \overset{p,\sigma}{\rightsquigarrow}_{R,E} t'$ (or $\overset{\sigma}{\rightsquigarrow}_{R,E}, \rightsquigarrow_{R,E}$) if $p \in Pos_{\Sigma}(t), l \to r \in R, \sigma \in CSU_E(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Note that $\rightsquigarrow_{R,E}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ induces a relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \overset{\sigma}{\rightsquigarrow}_{R,E} [t']_E$ iff $\exists w \in \mathcal{T}_{\Sigma}(\mathcal{X}) : t \overset{\sigma}{\rightsquigarrow}_{R,E} w$ and $w =_E t'$.

2.6 Meta-programming in Maude

Maude is a high-level language and a high-performance system [13] that supports both equational and rewriting logic computation. Rewriting logic is the logic of concurrent change, therefore a concurrent/distributed system can be specified in Maude. Moreover, rewriting logic is a reflective logic that can be faithfully interpreted in itself, making it possible to develop many advanced meta-programming and meta-language applications. Some descriptions on meta-programming from [13] are borrowed in this section. Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way so that the object-level representation correctly simulates the relevant metatheoretical aspect. Rewriting logic is reflective in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}$, $\overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$. Because $\mathcal{U}$ is representable in itself, we can achieve a reflective tower with an arbitrary number of levels of reflection:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

In this chain of equivalences, we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In Maude, the key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META−LEVEL`. This module includes the modules `META−VIEW`, `META−MODULE`, `META−STRATEGY`, and `META−TERM`. As an overview,

- in the module `META−TERM`, Maude terms are metarepresented as elements of a data type `Term` of terms.
- in the module `META−STRATEGY`, the Maude strategy language is metarepresented as terms in a data type `Strategy` of strategy expressions.
- in the module `META−MODULE`, Maude modules are metarepresented 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;
  - the process of reducing a term to a canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
  - the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;
  - the process of applying (without extension) a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
  - the process of applying (with extension) a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
  - the process of matching (without extension) two terms at the top is reified by a built-in function `metaMatch`;
  - the process of matching (with extension) a pattern to any subterm of a term is reified by a built-in function `metaXmatch`;
  - the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`;
  - the processes of rewriting a term using Maude's `srewrite` and `dsrewrite` commands are metarepresented by built-in functions `metaSrewrite` and `metaDsrewrite`; and
  - parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

Meta-programing is a programming technique in which programs have the ability to treat other programs as their data. 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 a program with respect to some properties. Hence, it is very useful and very powerful. In Maude, meta-programming has a logical reflective semantic. A term can be represented in the object-level or meta-level. The object-level representation can correctly simulate the relevant meta-level representation and vice versa. We can write Maude meta-programs by simply importing `META−LEVEL` module in our programs in which we can use the built-in functions supported, such as `metaReduce` and `metaSearch`.

Chapter 3

Related Work

This chapter investigates some advancements in parallel model checking for testing concurrent programs, LTL model checking, and cryptographic protocol analysis. For testing concurrent programs, bounded model checking, abstraction refinement, and some work in correct-by-construction software development, such as High-Level Language (HLL) to translate Event-B model checks to C code, EventB2Java to generate JML-annotate Java programs from Even-B models, are mentioned. For LTL model checking, DiVinE, Garakabu2, a multicore extension of SPIN, assume-guarantee verification, and parallel CTL model checking are mentioned. For cryptographic protocol analysis, other analysis tools than Maude-NPA are mentioned, such as Scyther, Tamarin, Proverif, and DEEPSEC.

3.1 Testing Concurrent Programs

The state space explosion is the main challenge to concurrent program verification due to inherent thread interleavings. Several techniques have recently been studied to overcome this problem, such as bounded model checking and abstraction refinement. Nevertheless, these existing techniques cannot overcome the problem reasonably well, especially for large concurrent program verification. To take full advantage of multiprocessor architectures, parallel processing is one mainstream to deal with this problem.

Bounded model checking (BMC) is an efficient technique for sequential program analysis that uses a symbolic representation to formalize the program verification problem into an equi-satisfiable CNF formula that can be analyzed by a SAT/SMT solver. Among existing studies for BMC, CBMC is a famous bounded model checker to verify sequential C programs [69]. Based on CBMC, a SAT-based bounded verification technique is then proposed to support multi-threaded C programs, called TCBMC [36]. For Java programs, JBMC is a bounded model checking tool for verifying Java bytecode [70], which is also based on CBMC. However, JBMC only supports sequential Java programs. The developers of JBMC are currently extending JBMC to support multi-threaded Java programs. Another extension of SAT/SMT-based BMC

to model check concurrent programs is Lazy Sequentialization (Lazy-CSeq) [37, 38]. Given a concurrent program P together with two parameters u and r that are the loop unwinding bound and the number of round-robin schedules, respectively, they first generate an intermediate bounded program P_u by unwinding all loops and inlining all function calls in P with u as a bound except for those used for creating threads. P_u then is transformed into a sequential program $Q_{u,r}$ that simulates all behaviors of P_u within r round-robin schedules. $Q_{u,r}$ is then transformed into a propositional formula that can be analyzed by a SAT/SMT solver. To take advantage of parallelization, they then propose a method [71] to decompose the set of execution traces of concurrent programs into symbolic subsets that can be separately explored by multiple instances of a SAT solver in parallel. Given a number of threads and a number of execution contexts, their approach needs to calculate symbolic partitioning in which a single formula is divided into multiple propositional sub-formulas. After that threads are spawned to check such sub-formulas by using SAT in parallel. They build a prototype tool that can be deployed on a single machine as well as in a distributed environment. To use much larger compute and memory resources than those of one single ordinary computer, distributed bounded model checking is one promising way to make verification scalable. Prantik Chatterjee et al. [72] then propose an algorithm that dynamically unfolds the call graph of a program and frequently splits it into sub-tasks that can be solved by many instances of an SMT solver in parallel. In detail, the technique splits the set of program paths into disjoins subsets that are searched independently. The splitting is done by picking a control node (splitting node) and considering (1) the set of paths that go through the node, and (2) the set of paths that do not. They create multiple processes, each of which has access to the input program, and are deployed in a distributed environment. One process is designated as the server while the rest are called clients. The search starts sequentially on one of the clients. A client chooses a splitting node from which two partitions are created. The client continues verification on one of the partitions and sends the other partition to the server. The server is in charge of collecting, prioritizing partitions from the clients, and distributing them to the clients subsequently. Note that clients can split multiple times. This process continues until a client reports a counterexample or there is no partition left in the server and all clients are idle. Their architecture is basically a master-worker model that is similar to the architecture used in our tool. However, the way they partition the reachable state space is different from our approach, which splits the reachable state space into multiple smaller sub-state spaces based on depth information. In recent years, exploiting the power of GPU in parallel computation with a huge number of cores, some researchers have used GPU to speed up the model checking verification [73, 74, 75], which may be one of our interesting future directions to improve our approach.

Regarding abstraction refinement, the scheduling constraint based abstraction refinement (SCAR) is an effective method for concurrent program verification [76]. SCAR is built on top of CBMC that is a bounded model checker for C and C++ programs. However, instead of using the scheduling constraint [77], SCAR ignores the scheduling constraint and uses a

scheduling constraint based abstraction refinement method that makes the constraints in the initial abstraction to be reduced significantly. From the initial abstraction, they use graph-based algorithms over an event order graph (EOG) for counterexample validation. Given that abstraction, a counterexample π is produced, which is a set of assignments to the variables in the abstraction. To validate the feasibility of π, they validate its corresponding EOG, which captures all the order requirements among the events of π. Some order requirements deducible from the EOG are called derived orders of the EOG. The derived orders are produced based on three rules by looping and they try to apply each rule to the EOG in sequence until this process reaches a fixpoint where no derived order can be produced anymore. From obtained derived orders, if there exists a cycle, then the EOG is infeasible. Otherwise, it is not sure whether the EOG is feasible or not. If the EOG is feasible, it is truly a counterexample. Whenever counterexample validation is determined to be infeasible, the abstraction is refined by a refinement generation method that obtains a set of constraints and then can be encoded into simple constraints and reduce a large amount of space. To take advantage of the SCAR method, a parallel refinement is proposed in which multiple engines are used to refine the abstraction simultaneously [78]. Their parallel technique performs on the whole abstraction search space instead of dividing the search space into small ones. Each engine does three steps that iteratively select a counterexample from the abstraction, validate the counterexample (CE), and analyze the refinement constraint (RC). Because they perform on the whole reachable search space, to select a counterexample and avoid duplication, they use a random search strategy. Besides, all engines share the learned clause lib that can be updated and checked whether a counterexample is verified or not by engines. All engines also use the same RC lib, whenever an engine generates a refinement constraint. The constraint is added to the RC lib that is simplified to avoid redundant constraints. By sharing RC lib, each engine can obtain multiple refinement constraints in each iteration, making the number of required iterations for each engine reduce. For each iteration, if one engine returns UNSAT, then the property is proven safe and we finish the verification. Otherwise, in the case of SAT, a counterexample is returned to validate the feasibility. If the counterexample is feasible, all engines will terminate and Unsafe is returned. Otherwise, the engine keeps on doing to analyze RC. Their approach is similar to our approach when they use the shared RC lib to avoid redundant constraints, while we use a shared cache to avoid redundant states and state sequences. However, our approach does not deal with the whole search space by each worker and so we do not need to consider the random search strategy and each worker conducts its sub-state spaces independently.

The existing parallel techniques mentioned above, where SAT/SMT solvers play the main role, are different from our approach. The advantage of their approach is that it is fully automatic to verify concurrent programs, while our approach starts from a specification and then implements a concurrent program based on the specification. Our technique can be used to verify not only a concurrent Java program [79], but also check whether the concurrent program conforms to the specification, which can be used to complement the very last step

in correct-by-constructions software development. We do not compare our tool with existing tools based on BMC and SCAR mentioned above because they are dedicated to verifying C and C++ programs. Besides, JBMC is used to model check sequential Java programs, but not concurrent Java programs. JPF is one of the most mature model checkers for Java programs. That is why it is used to evaluate and compare with JBMC [70] in terms of correctness and running performance. Therefore, it is worth comparing our tool with JPF as well as improving the running performance of JPF.

In correct-by-construction system or software development, Ning Ge et al. [26] propose a High-Level Language (HLL) for Event-B that can be used between Event-B models and C code. Event-B models are translated to respective HLL models, where Event-B invariants are proved using a SAT-based model checker, from which C code is automatically generated. Although the authors claim that they propose a technique that makes it possible to conduct conformance proofs between HLL models and the C code generated from the models, they do not describe how they do so in detail. This thesis describes how to check the conformance of programs with specifications in detail that can be regarded as a complement to the very last step in the correct-by-construction technique.

Dalvandi et al. [18] propose a way to generate executable code in Dafny [80] from scheduled Event-B models. Scheduled Event-B is an augmented version of Even-B such that a scheduling language is used to make the control flow in an Event-B model explicit and facilitate the derivation of algorithmic structure in Event-B refinement. Generated executable code can be verified with a static program verifier, such as Z3, because code is written in Dafny. Generated executable code is dedicated to sequential programs, while our tool can test concurrent programs.

Rivera et al. [19] propose a way to generate JML-annotated Java programs from Even-B models. They build a tool called EventB2Java to support the proposed program generation technique. Two case studies are conducted with EventB2Java to demonstrate the usefulness of the proposed technique and the support tool. Both sequential and concurrent programs can be generated with EventB2Java. Note that there is a sentence "... JML [...] is designed to specify arbitrary sequential Java programs ...," though, in the paper [20]. Because generated programs are annotated with JML [81], Java programs generated by EventB2Java can be verified. However, the paper [19] does not describe how to formally verify generated Java programs. To the best of our knowledge, JML focuses on the sequential behavior of Java programs, while extending JML to support concurrency is in progress [82]. Hence, checking, verifying, or testing JML-annotated Java concurrent programs is still not matured.

Tran-Jørgensen et al. [20] propose a way to automatically generate JML-annotated Java programs from VDM models. They address the semantic differences between the contract-based elements of VDM-SL and JML and describe how to use dynamic JML assertion checks to ensure the consistency of VDM's subtypes. It looks like that JML-annotated Java programs generated from VDM models are only sequential.

3.2 Linear Temporal Logic Model Checking

Barnat et al. [10] survey some recent advancements in parallel model checking algorithms for LTL. Graph search algorithms need to be redesigned to make the best use of multi-core and/or multi-processor architectures. Parallel model checkers based on such parallel model checking algorithms have been developed, among which are DiVinE 3.0 [83], Garakabu2 [84, 85], and a multicore extension of SPIN [86]. A parallel version of $L+1$-DCA2L2MC does not need to redesign graph search algorithms and can essentially use any existing model checkers that can handle leads-to properties. This is one big difference from any existing parallel model checkers.

SAT/SMT-based BMC has been extended to model check concurrent programs [37]. Given a concurrent program P together with two parameters u and r that are the loop unwinding bound and the number of round-robin schedules, respectively, an intermediate bounded program P_u is first generated by unwinding all loops and inlining all function calls in P with u as a bound except for those used for creating threads and then P_u is transformed into a sequential program $Q_{u,r}$ that simulates all behaviors of P_u within r round-robin schedules. $Q_{u,r}$ is then transformed into a propositional formula that can be analyzed by a SAT/SMT solver. This way to model check concurrent programs can be parallelized by decomposing the set of execution traces of a concurrent program into symbolic subsets and analyzing the set of execution traces in parallel [71]. The approaches to BMC of concurrent programs seem able to deal with safety properties only, while our tool is able to deal with leads-to properties, a class of liveness properties. Note that $\varphi \leadsto \bot$ is equivalent to $\Box\neg\varphi$ and then our tool can deal with invariant properties (safety properties) as well.

Distributed-memory model checking with SPIN has been proposed [87]. Its purpose is shared by the technique proposed in this thesis. When the size of a system model (or specification) under model checking by SPIN is greater than the physical memory, the model checking performance is degraded or the model checking may become infeasible. To make it possible to conduct model checking experiments for a large-state system model with SPIN, Lerda and Sisto [87] came up with a way to partition the reachable state space of a large-state system model into multiple nodes (computers) connected with networks, where some techniques are used by SPIN, such as partial order reduction and bit state hashing, can be used. Some experimental data demonstrate the usefulness of the proposal. Their distributed-memory SPIN is able to deal with safety properties only, while our tool is able to deal with leads-to properties, a class of liveness properties.

To tackle a large system that cannot be handled by an exhaustive verification mode, SPIN has a bit-state verification mode that may not exhaustively search the entire reachable state space. The larger a system under verification becomes, the higher chances the SPIN bit-state verification mode may overlook flaws lurking in the system. To overcome such situations, Swarm Verification [88] has been proposed. The key ideas of Swam Verification are parallelism and search diversity. For each of multiple different search strategies, one instance of bit-state

verification is conducted. These instances are totally independent and can be conducted in parallel. Different search strategies traverse different portions of the entire reachable state space, making it more likely to achieve higher coverage of the entire reachable state space and find flaws lurking in a large system if any. An implementation of Swarm Verification on GPUs, called Grapple [75], has also been developed. Although the technique proposed in this thesis splits the reachable state space from each initial state into multiple layers, generating multiple sub-state spaces, it exhaustively searches each sub-state space with Maude LTL model checker or its revision that can find all counterexamples at once in parallel. It may be worth adopting the Swarm Verification idea into our technique such that Swarm Verification is conducted for each sub-state space instead of an exhaustive search, which may make it possible to quickly find a flaw lurking in a large system.

Assume-guarantee verification is a technique to mitigate the state space explosion in model checking by decomposing a single analysis of the global state space into a number of local analyses. Each local analysis is in charge of checking whether a single component M guarantees a property g under assumption a denoted $\{a\}M\{g\}$, where a abstracts away the rest of a program. For example, we can verify each thread in a shared-memory program as a component separately by using an assumption that models interleaved steps of the other threads [89]. A compositional proof rule is used to reason about components separately along with their assumptions from which the global property is concluded without having to reason about the composed program directly. There are several proof rules that have been proposed in the work [90, 91, 92, 93] and most of them are based on two principles: *transitivity* and *mutual induction*. The former states that if component M satisfies $\{a\}M\{g\}$ and component N satisfies $\{true\}N\{a\}$, then the composition $N||M$ satisfies $\{true\}N||M\{g\}$. The latter states that if component M satisfies $\{a\}M\{g\}$ and component N satisfies $\{g\}N\{a\}$, meaning that component M guarantees property g under assumption a that is guaranteed by component N under assumption g, then the composition $N||M$ satisfies $\{true\}N||M\{a \wedge g\}$. The proof rules based on the latter are called circular rules that are sound for safety properties but potentially unsound for liveness properties. Because each component can be analyzed separately, then we can take advantage of parallelization to improve the verification time to some extent. However, the most challenge in assume-guarantee verification is to create right assumptions for components from which such a proof rule can be applied correctly [94]. Although some techniques have been devised to generate automatically the assumptions in [95, 96], the challenge still remains. Moreover, the number of components usually correlates with the number of threads/processes in a concurrent/distributed program under verification. Hence, not many components are involved and then the gain that can be obtained by parallelization may not be significant for the assume-guarantee verification technique.

Besides LTL model checking, there are several studies on improving the running performance of CTL model checking by taking advantage of high computing systems [97, 98, 99]. One of them is distributed CTL model checking by using MapReduce with the Hadoop framework [97].

They focus on three kinds of formulas in the forms of $EX\varphi$, $EG\varphi$, and $E[\varphi U \phi]$, where φ and ϕ are CTL formulas, because every CTL formula can be expressed over them. For each kind of the formula, they design a dedicated algorithm based on a fixed-point algorithm where MAP and $REDUCE$ functions for mappers and reducers in Hadoop are specified, respectively. For $EX\varphi$ formulae, a single MapReduce job is used where the predecessor states of states that satisfy φ are processed in parallel, while for $EG\varphi$, and $E[\varphi U \phi]$, an iterative MapReduce algorithm is used where the output of a previous iteration is the input of the next iteration until a fixed point is reached based on the fixed-point algorithm, where the input and the output are the same. MapReduce is basically a master-worker model as what is used in our thesis and our tool can also be deployed in a distributed environment because we use sockets to communicate between the master and workers.

3.3 Cryptographic Protocol Analysis

The way to parallelize the backward narrowing in step (1) in our work is close to parallel breadth-first search (BFS) algorithms. Various parallel BFS algorithms have been intensively studied [100, 101, 102, 103, 104]. Some of these algorithms work efficiently compared to the classical serial BFS algorithm [105, Section 22.2]. PBFS [103] uses a multiset data structure called a bag instead of a queue (FIFO). The bag supports insertion essentially as fast as FIFO and can be split and combined efficiently. In addition, for efficient implementation, PBFS contains a benign race condition in their algorithm and uses a bag reducer that allows updating concurrently to a shared variable or data structure at the same time. A bag is a collection of pennants in which each pennant is a tree of 2^k nodes, where k is a non-negative integer. Each node in this tree contains two pointers referring to its left and right children. The bag is a crucial data structure in PBFS that is implemented efficiently in C++, while we mostly use a set data structure that can be defined in Maude. Both Maude-NPA and Par-Maude-NPA-2 are written in Maude, a specification language, which is impossible to implement PBFS in a high-level language like Maude. However, our way to parallelize Maude-NPA would be what we could do as much as possible by using the currently available parallelization techniques. If Maude-NPA is drastically re-implemented in C++ to improve its efficiency, applying PBFS to the parallel version of Maude-NPA is one piece of our future work. Note that the idea to parallelize BFS is shared between our tool and PBFS and we have demonstrated that the breadth-first search in Maude-NPA can be reasonably parallelized. Besides, we parallelize the transition subsumption in Maude-NPA based on the concept of the divide and conquer approach in [105, Section 4].

In addition to Maude-NPA, there are several cryptographic analysis tools for security protocols, such as Athena [49], ProVerif [50], Avispa [51], Scyther [52], Tamarin [53], DEEPSEC [54], Verifpal [55], and CPSA [56]. Among them, some symbolic tools are described and compared with Maude-NPA as follows. Scyther [52] is an automatic analyzer that supports both an

unbounded number of sessions and a bounded number of sessions, and always terminates. However, it only considers a fixed set of cryptographic primitives consisting of symmetric and asymmetric encryption. Security properties verified in Scyther are mostly trace properties that hold for any execution trace, where secrecy and authentication properties can be expressed. Maude-NPA supports rich cryptographic primitives that are user-definable and it can verify not only trace properties as its natural reachability analysis but also equivalence properties [106], which state that two processes in a security protocol are equivalent when an adversary cannot distinguish the difference between interactions with two processes. Verifying equivalence properties is harder than verifying trace properties because we need to consider the relation between traces instead of a single trace.

Tamarin [53] is a prover that generalizes the backward search used by the Scyther tool and supports an unbounded session model, reasoning modulo equational theories, modeling complex control flow (e.g., loops), and mutable global state. It provides both automatic and interactive modes to construct proofs. However, it often needs some lemmas provided by users to complete its proofs. In Tamarin, a protocol specification is specified by means of multiset rewriting rules, while a property specification is written as a guarded fragment of first-order logic. Each protocol trace corresponds to a multiset rewriting derivation that is the sequences of the labels of the applied rules. Tamarin performs an exhaustive backward search to look for a trace that does not satisfy the property and returns a counterexample as an attack. If no rule can be applied anymore and no counterexample is found, then the protocol satisfies the property. Tamarin can verify trace properties and observational equivalence properties. To make the problem of security protocol verification decidable, Tamarin also uses the finite variant property [107] to reduce reasoning modulo an equational theory with respect to a rewrite theory as in Maude-NPA. Basically, Tamarin can support at the same level as Maude-NPA in cryptographic protocol analysis. However, Maude-NPA does not require lemmas from users and it is fully automatic.

ProVerif [55] is an abstraction-based approach to symbolically analyzing cryptographic protocols. The protocol specifications that are specified in an extension of the pi calculus are translated into Horn clauses and the security properties being proved are translated into derivability queries on the Horn clauses. ProVerif uses a resolution algorithm to check whether a fact is derivable from the clauses. If there is no derivation, the property is proved. Otherwise, the derivation found is reconstructed at the pi calculus level as an attack. However, the attack may be spurious because some abstractions are used in Horn clauses. In the case of a false attack, ProVerif cannot conclude anything. ProVerif can verify secrecy, authentication, and some observational equivalence properties. Besides, cryptographic primitives can be defined by equations or rewrite rules that also need to satisfy the finite variant property to make the analysis terminate as in Maude-NPA. However, it does not support associativity and homomorphic properties as in Maude-NPA.

DEEPSEC [54] focuses on deciding trace equivalence properties in security protocols, which

are specified in a dialect of the applied pi calculus [108]. However, it only supports a bounded number of sessions and cryptographic primitives are specified by a set of subterm convergent rewrite rules, where the right-hand side of each rule must be a subterm of the left-hand side or a ground term. To guide the decision of equivalence of two processes in cryptographic protocols, DEEPSEC constructs a so-called partition tree, where each node consists of a set of symbolic processes and constraints. Initially, the root node only consists of the two symbolic processes and empty constraints. Given a node, sibling nodes can be constructed based on some rules in DEEPSEC. The partition tree is then constructed in a top-down style. While constructing the partition tree, if there is some node that does not contain both two processes originated from the two beginning processes, DEEPSEC returns an attack; otherwise, there are no attacks. Because sibling nodes are independent, the construction of the subtree from each sibling node can be processed in parallel as follows. DEEPSEC maintains a queue of jobs with a fixed size. It first starts with a breadth-first search from the root node to generate all successor nodes and put them into the queue until reaching the size. Each idle worker can fetch a job to handle and checks if the queue is full. If so the worker starts constructing the entire subtree from the node included in the job; otherwise, it keeps on producing jobs to put into the queue. The way to parallelize DEEPSEC is different from ours because we never generate the entire subtree from a node. It also seems that DEEPSEC does concern visited nodes while constructing the partition tree. To the best of our knowledge, Maude-NPA is the first cryptographic security tool parallelized for an unbounded number of sessions. Although there are many parallel model checking algorithms for LTL [10], such as DiVinE 3.0 [83], Garakabu2 [84, 85], a multicore extension of SPIN [86], and Parallel $L+1$-DCA2L2MC [109], where DCA2L2MC stands for a divide & conquer approach to leads-to model checking.

Parallel $L+1$-DCA2L2MC [109] is a new technique to mitigate the state space explosion as well as increase the running performance in model checking for leads-to properties. $L+1$-DCA2L2MC splits the reachable state space from each initial state of a system into multiple layers, generating multiple sub-state spaces, and conducting model checking experiments for each sub-state space. If the size of each sub-state space is much smaller than the one of the original reachable state space, it is feasible to conduct model checking experiments with the approach even though it is infeasible to do so for the original reachable state space due to the state space explosion. Model checking experiment for each sub-state space is basically independent, especially, the model checking experiments for the sub-state spaces in the final layer are completely independent. Parallel $L+1$-DCA2L2MC is then proposed to conduct such model checking experiments in parallel. The way to parallelize $L+1$-DCA2L2MC is close to our work where they also use a master-worker model with a support tool, which is implemented in Maude. Each model checking experiment can be regarded as a job assigned to a worker by the master to conduct the model checking experiment in parallel. In their work, they use Maude sockets instead of meta-interpreters to communicate between the master and workers. As described above, the use of meta-interpreters is faster than the use of Maude sockets.

Chapter 4

Techniques to Parallelize Formal Verification Tools

We parallelize three formal verification tools: (1) parallelization of JPF for testing concurrent programs, (2) parallelization of Maude LTL model checker for checking leads-to properties, and (3) parallelization of Maude-NPA for cryptographic protocol analysis. Some techniques are shared between three parallel versions of these formal verification tools. This chapter then describes the shared techniques and a generic approach to parallelizing tools used for formal methods.

4.1 A Master-Worker Model

A master-worker model is not a brand-new algorithm for parallelization but is widely used in practice. There is an option to use an existing collection of open-source software facilities that make the best use of parallel/distributed environments, such as Apache Hadoop[6] that uses the MapReduce programming model [110] to implement a parallel version of a formal verification tool. However, we did not take the option. This is because we do not use a large number of computers, making it unnecessary to consider computer failure and/or stragglers. Note that MapReduce is basically a master-worker model. For (1), we use RabbitMQ, a robust message queue library, as a message broker, while we develop our message broker for (2) and (3) from scratch because (2) and (3) are developed in Maude and there is no such message broker developed in Maude before. The master uses the message broker to distribute jobs to workers in a well-balanced way. For (1) and (2), we use sockets to communicate between the master and workers while we use Unix domain sockets to obtain better performance for (3).

[6] `https://hadoop.apache.org/`

4.2 Dividing the Reachable State Space into Multiple Layers

To make the best use of the parallelization, we need to come up with a way/technique to divide the reachable state space into multiple sub-state spaces such that tackling the problem for those sub-state spaces is equivalent to the problem for the original reachable state space. For (1), (2), and (3), we divide the reachable state space from each initial state of the system under test/verification into multiple layers, generating multiple sub-state spaces. If each sub-state space is much smaller than the original reachable state space, then it is feasible to tackle each sub-state space even though it is infeasible to tackle the original reachable state space due to the state space explosion problem. This is the key to alleviating the state space explosion problem in our techniques. Regarding layer configuration, the depth of each layer is different for each tool. For (1), we describe a preliminary way to decide the depth of each layer. For (2), we come up with an approach to finding a good layer configuration with an analysis tool that supports the approach. For (3), we parallelize the breadth-first search in Maude-NPA, and then the depth of each layer is merely one.

4.3 Tackling Multiple Sub-State Spaces in Parallel

As described above, the reachable state space is divided into multiple layers, generating multiple sub-state spaces at each layer. Those sub-state spaces at each layer can be tackled independently in parallel, although we may need to do some synchronized tasks after each layer before moving to the next layer. If the number of sub-state spaces at each layer is considerably large, we can take advantage of parallelization by using the master-worker model as mentioned above. In addition, we should consider the size of each sub-state space. If the size is too small, tackling the sub-state space (a job) can be completed quickly by a worker and so the communication cost may be larger than the benefit able to be gained from parallelization. However, if the size is too large, each worker may tackle some shared sub-state spaces and we may not take advantage of the use of cache effectively, which is described in the next section, making the running performance degrade. The size of each sub-state space is decided by the layer configuration. In general, a job assigned to each worker by the master should not be a light job. Tackling multiple sub-state spaces in parallel is the key to improving the running performance of formal verification tools.

4.4 Caching to Avoid State Duplications

The use of cache is essential in software development. In our tools, we use cache to avoid making unnecessary duplications of jobs and reduce the communication cost between the master and

workers. In the master-worker model, we can use a cache at the master as well as at each worker. The master is in charge of distributing and collecting jobs to/from workers, and so the use of cache is to avoid making unnecessary duplications of jobs because workers only communicate with the master. Each worker is in charge of handling each job assigned by the master. The worker often produces new jobs to send back to the master, and so the use of a cache at each worker can avoid sending unnecessary jobs that have been sent to the master by the worker before. To see the effectiveness of the use of a cache at the master as well as at each worker, we investigate it for (2) in Chapter 6. In general, the use of a cache in the master is mandatory while the use of a cache at each worker is optional.

4.5 A Generic Approach to Parallelizing Tools Used for Formal Methods

The advantage of our three parallel versions is that we do not need to alter the core algorithm used in each model checker for (1), (2), and (3). Especially, we would like to improve the running performance for (2), and so we develop a new model checker; otherwise, we do not need to do so. We think that not every tool used for formal methods deserves parallelization. From our experiences, the following two criteria can be used to decide whether we should parallelize a tool used for formal methods:

- We need to come up with a way/technique to divide the original reachable state space from each initial state of the system under test/verification into multiple sub-state spaces such that each sub-state space is preferable to be conducted mostly in parallel and tackling the problem for such sub-state spaces is equivalent to the problem for the original reachable state space. For (1), we divide the reachable state space from each initial state of the system under test into multiple layers by each layer depth. For (2), we use $L+1$-DAC2L2MC technique to divide the reachable state space into $L+1$ layers. For (3), we transform the breadth-first search in Maude-NPA into a parallel breadth-first search. All of them generate multiple sub-state spaces at each layer that basically can be checked in parallel and the correctness is preserved.

- For each sub-state space, the job that a worker needs to handle should not be a light job. Besides, the number of jobs in total assigned to workers should be considerably large as well. Because we need to pay an extra cost for parallelization, such as the communication cost between the master and each worker, and checking duplicated jobs. If the job is a light job and the number of jobs is small, the extra cost may be larger than the benefit able to be gained from parallelization. For (1), we set each layer depth to a moderately large value so that a worker needs to spend considerable time completing the job for each sub-state space in a layer. For (2), the sub-state spaces at the final layer are not

light jobs also. For (3), generating all successor states from a given state and conducting the transition subsumption at each layer is time-consuming because of its complexity implemented in Maude-NPA. For (1) and (2), the number of jobs is often considerably large, while (3) works really effectively when the number of states (jobs) located at each layer is considerably large.

To take advantage of parallelization for tools used for formal methods, we should consider the two criteria above to decide whether we should parallelize the tools. For example, we have attempted to parallelize the `search` command in Maude by using `metaSearch` and `metaMatch`. Although the `search` command uses the breadth-first search to explore the reachable state space of a system in Maude and there may be many states located at each layer, we could not take advantage of parallelization compared to the original `search` command in Maude by parallelizing the breadth-first search. In this case, it looks similar to the parallel version of Maude-NPA when we attempt to parallelize it in the sense of the parallel breadth-first search, meaning that the first criterion holds. However, the second criterion does not hold. In the `search` command, the time taken to complete a job, generating all successor states from a given state, is so fast that the extra cost of parallelization makes the running performance of the parallel version of the `search` command degrade. In addition, we use the reflective facilities in Maude that are slower than the use of the `search` command, which is implemented in C++, in terms of running performance. Therefore, we could not take advantage of parallelization for the `search` command in Maude in which the breadth-first search is parallelized. This case can be regarded as an example that does not deserve parallelization from our point of view. However, if we come up with a better way to divide the reachable state space into multiple sub-state spaces for the Maude search command in the future, the techniques proposed in this thesis could be effectively used to parallelize the Maude search command.

Note that the techniques proposed in this thesis are not for parallelization of any fundamental algorithms, such as those for SAT/SMT, but for parallelization of some tools for formal verification. Thus, they cannot be used to parallelize algorithms used for SAT/SMT, but could be used to parallelize SAT/SMT solvers (tools). It is necessary to decide what should be parallelized or divided. One possible candidate for a SAT/SMT solver is to divide a SAT/SMT problem or a formula into multiple sub-problems/sub-formulas. If we come up with a good way to do so, the techniques proposed could be effectively used to parallelize a SAT/SMT solver. Actually, there are some existing studies that have parallelized a SAT/SMT solver in this approach [71, 72, 78].

Chapter 5

Parallel Specification-based Testing for Concurrent Programs

This chapter proposes a specification-based testing technique for concurrent programs. For a formal specification S and a concurrent program P, state sequences are generated from P and checked to be accepted by S. 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. Thus, we propose a technique to generate state sequences from P and check if such state sequences are accepted by S in a stratified way. A tool is developed to support the proposed technique that can be processed naturally in parallel. Some experiments demonstrate that the proposed technique mitigates the state space explosion and improves the verification time, which cannot be achieved with the straightforward use of JPF. This chapter also describes how to extend the technique to test concurrent programs with JPF without checking if execution sequences generated from Java programs can be accepted by Maude specifications.

5.1 Specification-based Concurrent Program Testing with a Simulation Relation

We have proposed a specification-based testing technique for concurrent programs that uses a simulation relation candidate from a concurrent program to a formal specification [27]. The technique is depicted in Fig. 5.1. 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

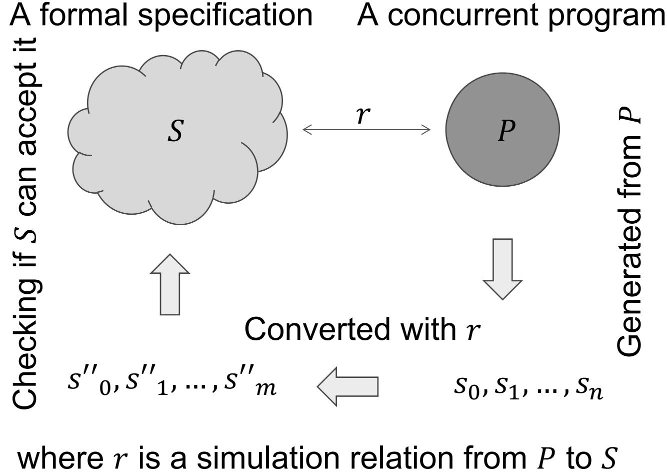


Figure 5.1: Specification-based concurrent program testing with a simulation relation

$s''_1, s''_2, \ldots, s''_m$ 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 [111]. 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. Programmers are able to know the maximum number of atomic code fragments in programs for one atomic state transitions in formal specifications if programs are implemented/generated from formal specifications based on some reasonable techniques. EventB2Java [19] can generate concurrent Java programs from Event-B models in which actions of each event are translated to one atomic fragment and a shared lock is used to guarantee that at most one atomic fragment (or an event) can be executed at the same time by multiple threads in programs. Hence, one possible way to implement programs from formal specifications is to use a shared lock by multiple threads and translate each state transition in specifications to one atomic fragment in programs. If so, the maximum number of such transition steps in specifications is one. However, we can use more than one lock to implement programs. For ABP protocol implementation, we use two locks shared by multiple threads in the program as shown in Algorithm 2. One atomic fragment protected by one lock and another atomic fragment protected by the other lock can be executed in parallel. Hence, at most two atomic code fragments may be executed by multiple

threads in parallel (or simultaneously), although each atomic action (or state transition) in the formal specification is implemented by one atomic code fragment in the program. In general, the number of locks used is the maximum steps in programs for one atomic state transition in formal specifications. We suppose that P is written in Java and Java Pathfinder (JPF) is used to generate state sequences from P [112].

5.2 State Sequence Generation from Concurrent Programs

5.2.1 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 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 concurrent Java programs, such as race conditions and deadlocks. When a flaw is detected, it reports a whole execution leading to the flaw. JPF explores all potential executions of a program under test, while an ordinary Java VM executes the code in only one possible way. Because JPF is 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 previously [27] 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. The present work then proposes a divide & conquer approach to generating state sequences from a concurrent program in a stratified way.

5.2.2 Generating State Sequences by JPF

JPF consists of two main components: (1) a VM and (2) a search component. The VM is a state generator. It generates state representations by interpreting Java 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 a state management that makes it possible to do state matching, state storing, and execution backtracking when exploring 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 on which the VM should work, 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. There are some strategies that can be used to traverse the state space. By default, the search component uses a depth-first search (DFS), although we can configure JPF to use different strategies, such as a breadth-first search.

The most important extension mechanism of JPF is listeners which provide a way to observe, interact with, and extend JPF execution. We can configure JPF with many of our own listener classes provided that our own listener classes need to 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.

A SequenceState class that extends ListenerAdapter class is made to observe and interact with JPF execution. In SequenceState class, we override the two important methods: stateAdvanced and stateBacktracked. Whenever the stateAdvanced method is invoked, we need to retrieve all necessary information about the next state at this step. We use an instance Path of ArrayList class to maintain the path from the beginning state to the current state being visited by the DFS. Each element of Path corresponds to a state in JPF and is encapsulated as an instance of a Configuration class prepared by us. 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 a state;
- **depth** - the current depth of search path;
- **lock** - a Lock object that contains the lock observable component value, which is either true or false;
- **threads** - an ArrayList object of threads, each of which consists of the current location information that is either rs or cs.

Figure 5.2: A way to generate state sequences with JPF

We need to keep up with the change of observable components in each state stored in Path. Observable components are implemented as object data in JPF. To obtain that information, 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 of observable components. Thereby, we can gather the values of observable components, create a new Configuration object, and append it to Path whenever the stateAdvanced method is invoked.

Whenever JPF hits an end state, a state that has already been visited or a depth bound, a path (or a state sequence) is made, all but one are deleted from each of consecutive same states in the path. A cache is used to store all paths that have been explored. If the path does not exist in the cache, we save the path to the cache and then send it to another part of our system to check whether the path conforms to the specification with Maude on the fly subsequently. Otherwise, we just discard the path.

Because the reachable state space could be huge, we manage a parameter in order to prevent JPF from diverging as follows:

- **DEPTH** - the maximum depth from the initial state; once JPF reaches any state whose depth from the initial state is DEPTH, a backtrack message is sent to the search component for backtracking.

DEPTH could be set to unbounded, meaning that we ask JPF to generate as deep state sequences as possible. Every time JPF performs backtracking because of no more successor state, the last state is deleted from Path in the stateBacktracked method to keep up with the change of the current path in the DFS.

The way to generate state sequences from concurrent programs with JPF is depicted in Fig. 5.2. White nodes with a thick border in red indicate that those nodes have been visited by JPF. Blue nodes cause backtracking because the node (or state) does not have any more

successor nodes and such a node has been seen (or visited) before or the depth of the node reaches DEPTH.

5.3 A Divide & Conquer Approach to Generating State Sequences

JPF often encounters the notorious state space explosion even without checking any property violation while exploring the reachable state space of a system under test. When we do not set DEPTH to a moderately small number and ask JPF to exhaustively (or almost exhaustively) explore all (or a huge number of) possible states, JPF may not finish the exploration and may lead to out of memory. To mitigate the situation, we propose a technique to generate state sequences from concurrent programs in a stratified way, which is called a divide & conquer approach to generating state sequences. Given a concurrent program P, our approach splits the reachable state space from each initial state s_{d_0} into multiple layers, for example, L layers (where L is a non-zero natural number) as shown in Fig. 5.3. Let $d(i)$ be the depth of layer i for $i = 0, 1, \ldots, L$. We suppose that there virtually exists layer 0 such that $d(0) = 0$. $d(i)$ is a non-zero natural number if $i = 1, \ldots, L$. Let d_i be $d(0) + \ldots + d(i)$ for $i = 0, \ldots, L$, namely that d_i is the depth of the bottom of layer i (or the depth of the top of layer $i + 1$) from the initial state. States located at the depth d_i are called beginning states of layer $i+1$ (or ending states of layer i). The depth of a state is the depth from the initial state where the state is located. In Fig. 5.3, s_{d_i} denotes a beginning state of layer $i+1$ for $i = 0, \ldots, L$, and $s_{d_i}^{j_i}$ denotes an ending state of layer i for $i = 0, \ldots, L$, although we do not use $s_{d_0}^{j_0}$ (which is an ending state of layer 0, a beginning state of layer 1, and the same as s_{d_0}) in the figure. $s_{d_i}^{j_i}$ is also a beginning state of layer $i + 1$, such as s_{d_l} that equals $s_{d_l}^{j_l}$ in Fig. 5.3.

Intuitively, we first generate state sequences from each initial state, where the length of each sequence is $d(1)$ (see Fig. 5.3). If $d(1)$ is small enough, it is possible to do so. We then generate state sequences from each of the states at depth $d(1)$, where the length of each sequence is $d(2)$. If $d(2)$ 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 number of states at depth $d(1)$ reachable from the initial state. 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 and either the last state of seq_1 or the first state of seq_2 is removed, we are to generate state sequences, where the length of each sequence is $d(1) + d(2)$, which can be done even though $d(1)+d(2)$ is large. Similarly, we could generate state sequences up to depth $d(1) + \ldots + d(L)$ for L layers (see Fig. 5.3).

Let Π_i for $i = 0, \ldots, L$ be the set of all state sequences generated in layer i reachable from initial states. Initially, Π_0 is the set of initial states whose depth is 0. For a state sequence $\pi \in \Pi_i$ for $i = 0, \ldots, L$, let last(π) be the last state in π. For a state s_{d_i} at the bottom of

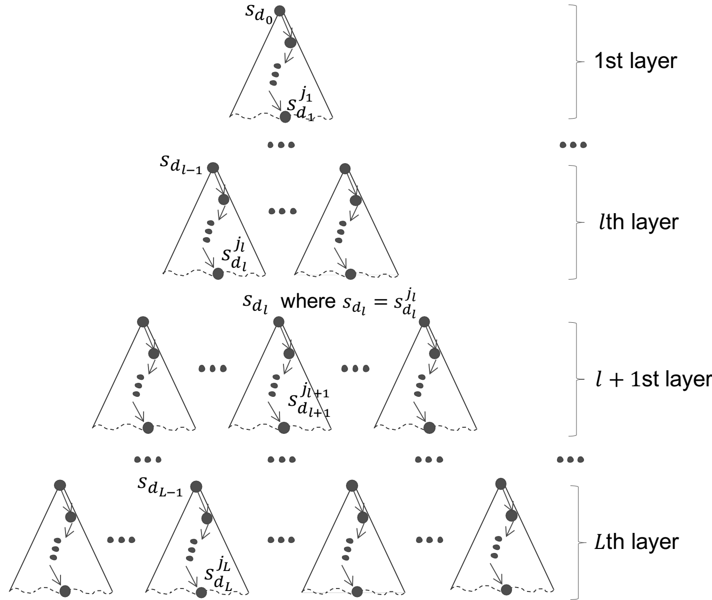


Figure 5.3: A divide & conquer approach to generating state sequences

the ith layer or the beginning of the $i+1$st layer (see in Fig. 5.3), let gen$(s_{d_i}, d(i+1))$ for $i = 0, \ldots, L-1$ be the set of state sequences in the $i+1$st layer reachable from s_{d_i}, where the length of each state sequence is $d(i+1)$. For two state sequences π and π' such that the last state of π is equal to the first state of π', let $combine(\pi, \pi')$ be the combined state sequence of π and π', where either the last state of π or the first state of π' is removed. Algorithm 1 shows a divide & conquer approach to generating state sequences for L layers reachable from the set of initial states Π_0. For each layer $l \in 1 \ldots L$, Π_l is initially set to empty at line 2. Suppose that we have the set of state sequences Π_{l-1} of the $l-1$st layer, which is obvious for layer 1 because Π_0 has initialized. For each state sequence π in the preceding layer, whose length is d_{l-1}, we get the last state of π to generate a set of state sequences reachable from $last(\pi)$, where the length of each state sequence is $d(l)$, and assign to Seq at line 4. If $d(l)$ is small enough, it is possible to do so. For each state sequence π' in Seq, we can combine the state sequence π and π' to get a state sequence of the lth layer, whose length is d_l, because the last state of π is equal to the first state of π', and add it to Π_l at line 6. By generating state sequences at each layer and combining them with state sequences at the preceding layer, we are able to generate state sequences for L layers even though d_L is large. Π_L is returned as the final result of Algorithm 1 at line 7.

When generating state sequences for L or unbounded layers, DEPTH can be regarded as $d(1) + \ldots + d(L)$ or unbounded, respectively. When the entire reachable state space is huge, DEPTH parameter is also shared by many bounding techniques, which systematically explore a part of the entire reachable state space, such as bounded model checking (BMC) [113, 69, 70] and context bounding [114, 115]. In those existing studies, the depth parameter is iteratively increased until a bug or counterexample is found. In our environment, selecting DEPTH, or in

Algorithm 1: A divide & conquer approach to generating state sequences for L layers

input : P – a concurrent program
Π_0 – the set of initial states of P
$d(1)\ \ldots\ d(L)$ – a list of non-zero natural numbers, where L is a non-zero natural number

output: a set of state sequences

```
1 forall l ∈ 1...L do
2     Π_l ← ∅;
3     forall π ∈ Π_{l−1} do
4         Seq ← gen(last(π), d(l));
5         forall π′ ∈ Seq do
6             Π_l ← Π_l ∪ combine(π, π′);
7 return Π_L;
```

other words, the depth of each layer is essential. We can select each layer depth as follows. We start with a small depth, namely 10, as the layer depth and increment it by one small number, namely 5, until JPF is not able to explore the entire sub-state space up to the depth reachable from an initial state in a reasonable amount of time. The depth in which JPF is able to do so at the last should be used as each layer depth. In addition, the number of states located at the first layer for each depth option is also considered. However, finding good depth information for layers is one piece of our future work.

Let us consider the test&set protocol and suppose that we write a concurrent program (denoted $P_{\mathrm{t\&s}}$) in Java based on the specification $S_{\mathrm{t\&s}}$ of the protocol. We suppose that there are three processes participating in the protocol. $S_{\mathrm{t\&s}}$ has one initial state and so does $P_{\mathrm{t\&s}}$. Let each of $d(1)$ and $d(2)$ be 50 and let us use the proposed technique to generate state sequences from $P_{\mathrm{t\&s}}$. 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 $P_{\mathrm{t\&s}}$ are totally different from those used in $S_{\mathrm{t\&s}}$. 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 the two state sequences such that consecutive equal states are removed to withhold one, we get the combined state sequence (denoted seq_3) 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)} |
{(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 $P_{\mathrm{t\&s}}$, where DEPTH is 100.

If each sub-state space is much smaller than the original reachable state space, then it is feasible to generate its sub-state sequences even though it is infeasible to generate state sequences for the original reachable state space due to the state space explosion problem. Therefore, using the divide & conquer approach to generating state sequences, we are able to generate longer or deeper state sequences that are unfeasible by using JPF only without our approach. In addition, generating state sequences for each sub-state space is independent from that for any other sub-state spaces. 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 l, we can generate state sequences for all sub-state spaces in layer $l + 1$ simultaneously. This is an advantage of the divide & conquer approach to generating state sequences from concurrent programs.

5.4 A Divide & Conquer Approach to Testing Concurrent Programs and Its Support Tool

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

For each layer l, we generate state sequences that start from each state located at depth $\mathrm{d}(1) + \ldots + \mathrm{d}(l - 1)$ 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 a stratified way and then could check if each state sequence can be accepted by S as shown in Algorithm 1. But, we do not combine multiple (sub-)state sequences to generate a whole state sequence of P because we do not need to do

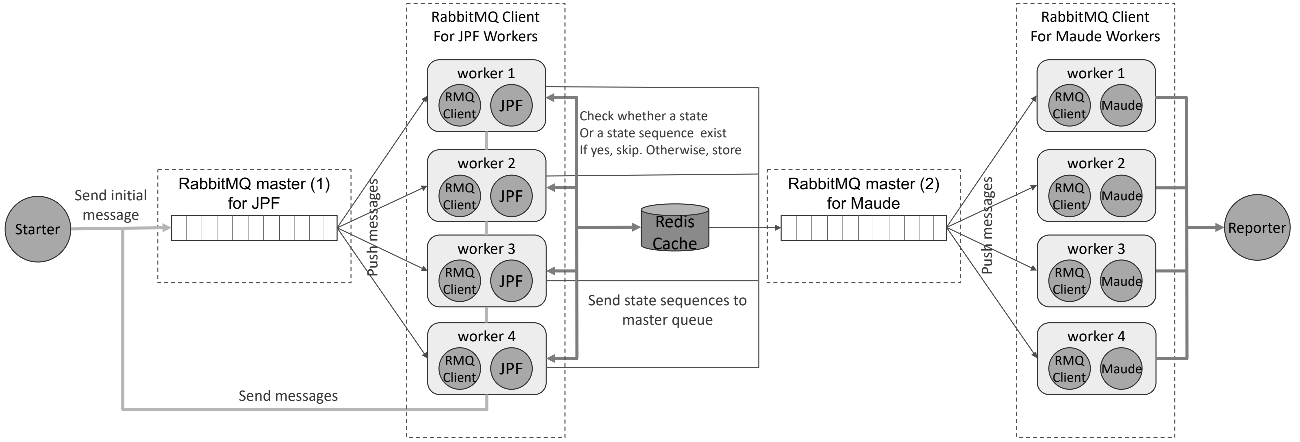


Figure 5.4: The architecture of a tool supporting the proposed technique

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. 5.4. As shown in Fig. 5.4, the architecture is a master-worker model (or pattern), where there are two main groups. The first left half group uses one RabbitMQ master (1) and four JPF workers to generate state sequences while the second right half group also uses one RabbitMQ master (2) and four Maude workers to check whether or not state sequences can be accepted by specifications on the fly. Note that we can use as many workers as possible in each group. Both state sequence generation and conformance checking to specifications are conducted in parallel. We use Redis [116] and RabbitMQ [117] to develop our tool.

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

- RabbitMQ is used as a message broker. The RabbitMQ master maintains a message queue to store and dispatch messages from/to RabbitMQ (RMQ) clients. In the first group, each JPF worker consists of a RabbitMQ client to fetch messages (states) from the RabbitMQ master (1). For a fetched message (a state), a JPF instance that is internally started by the worker starts generating state sequences from the state up to a depth. For each state sequence generated, we obtain the last state in the state sequence and check if the state exists in a cache of states. If not, the state is sent back to the RabbitMQ master (1). We jointly check if the state sequence exists in a cache of state sequences. If

not, the state sequence is sent to the RabbitMQ master (2). In the second group, each Maude worker consists of a RabbitMQ client to fetch messages (state sequences) from the RabbitMQ master (2). For a fetched message (a state sequence), a Maude instance that is internally started by the worker checks if the state sequence can be accepted by a formal specification.

Initially, we run a starter program to send an initial message to the RabbitMQ master (1) for JPF to kick off the tool, where an initial message is regarded as an initial state specified in a specification. The starter program is just specialized in sending an initial message. First of all, we flush all keys and values from the Redis cache to clean up data in memory. Secondly, we make a connection to RabbitMQ master for JPF with a designated configuration. After making sure that it is connected, we prepare an initial message to send to the message queue maintained by RabbitMQ master (1). The data is encapsulated into a *Configuration* object, namely *config*. Before sending the initial message to the RabbitMQ master, we use *SerializationUtils* class supported by the Apache Commons Lang [118] to serialize the *config* object that makes it easy to deserialize to the original object at the receiver side without doing any extra thing.

As soon as the RabbitMQ master has received a message from a worker, the message is stored in a message queue. By default, the RabbitMQ master will pop a message from the message queue and then dispatch it to a worker, in sequence. RabbitMQ has a noticeable parameter that needs to be configured, namely, *prefetch*. It indicates a maximum of unacknowledged messages that each worker may receive at once. If *prefetch* is not configured, the default value is unlimited. From our experience, it takes much more time for JPF workers to generate state sequences than for Maude workers to check if state sequences are accepted by formal specifications. Hence, to improve the stability and efficiency of our environment, *prefetch* value is assigned to 1 and 10 for JPF workers and Maude workers, respectively.

Let us consider the first left half group in the environment architecture. Firstly, JPF workers make a connection to the RabbitMQ master (1). After connected, workers are willing to receive messages from RabbitMQ master (1). Whenever a worker receives a message from RabbitMQ master (1), the worker deserializes the message into its original object, namely *config*, which is an object of *Configuration* class, by invoking the *deserialize* method of *SerializationUtils* class. Given the *config* object, we create an instance of *RunJPF* class. Then the instance invokes the *run* method to initialize a JPF instance with some configuration that is built from the *config* object. We need to let the JPF instance know which message arguments are passed to the system under test and also need to register our listener class to the JPF instance so that we can interact with the JPF instance. Consequently, the worker can internally start a JPF instance to generate state sequences from the given message.

Note that all workers, as well as JPF instances, are running in parallel and using one shared Redis instance. A JPF instance traverses the (sub-)state space reachable from the state derived from a given message. Whenever a JPF instance reaches the designated depth or finds that the

current state being visited has no more successor states, our listener class does the following:

1. Removing all consecutive same states except for one from the state sequence;
2. Converting the state sequence to a string representation, then using the SHA256 algorithm to hash the string representation to a unique signature;
3. Asking the Redis cache whether the state sequence exists or not; If yes, skipping what follows; Otherwise, saving the signature as the key and the string representation as the value into the Redis cache, making a connection to the RabbitMQ master for Maude (2), and then sending the state sequence as a message to the RabbitMQ master (2) for conformance checking to formal specifications in the second group subsequently;
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 or not; If yes, skipping what follows; Otherwise, asking the Redis cache to save the signature as the key and the string representation for the last state as the value into the Redis cache and sending a message that contains the last state's information to RabbitMQ master (1), which then prepares a message that asks a worker to generate state sequences from the state unless the current layer is the final one.

Let us consider the second right half group where the tool has been integrated with Maude so that a Maude instance can check if state sequences are accepted by formal specifications on the fly. The RabbitMQ master (2) is used to gather all state sequences emitted from the workers in the first group. We have known that once JPF instances have generated state sequences, they send the state sequences to the RabbitMQ master (2) where a message queue is maintained to store such state sequences. The RabbitMQ master (2) then gradually dispatches state sequences as messages to Maude workers. Initially, each Maude worker makes a connection to the RabbitMQ master (2) with a designated configuration. After connected, the worker launches internally a Maude instance and feeds to the Maude instance some Maude files that are a specification of a concurrent program being tackled and a meta-programming script to check the correctness of state sequences. Whenever the worker receives a message, the worker deserializes the message into the original object that represents a state sequence. Then it calls to the Maude instance to check whether or not the state sequence is accepted by the specification loaded into the Maude instance before. Given a command line with a designated module name, a state sequence, and a depth, an instruction that can be fed into the Maude instance is constructed. Let `M` be a module name, `Seq` is a state sequence being checked, and `D` is a depth that is the possible number of transition steps (see Sect. 5.1). The instruction looks as follows:

```
Maude> reduce checkConform(M, Seq, D) .
```

Whenever the Maude instance receives the instruction, the Maude instance executes it and checks whether `Seq` is accepted by `M` with depth `D`. A result will be returned in the form of either a success or failure message. As the worker that calls the Maude instance to check the state sequence receives the message result from the Maude instance, it parses the message to know what it is, and then displays the result to the console output. All state sequences and their results can be stored in MySQL [119] to easily monitor and diagnose problems if needed. Note that all workers are running in parallel and use different Maude instances but load the same specification and the meta-programming script.

5.5 Case Studies

We conducted case studies in which Alternating Bit Protocol (ABP), a cloud synchronization protocol (CloudSync) and Needham-Schroeder-Lowe Public-Key authentication protocol (NSLPK) were tackled. We experimented on two versions of CloudSync protocol, including Revised CloudSync and Original CloudSync, which are described in detail in this section. Hence, we conducted four case studies in total. Our experiments were carried out by an Apple iMac Late 2015 that had a Processor 4GHz Intel Core i7 and a Memory of 32GB 1867 MHz DDR3.

5.5.1 Alternating Bit Protocol (ABP)

Introduction

Alternating Bit Protocol (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

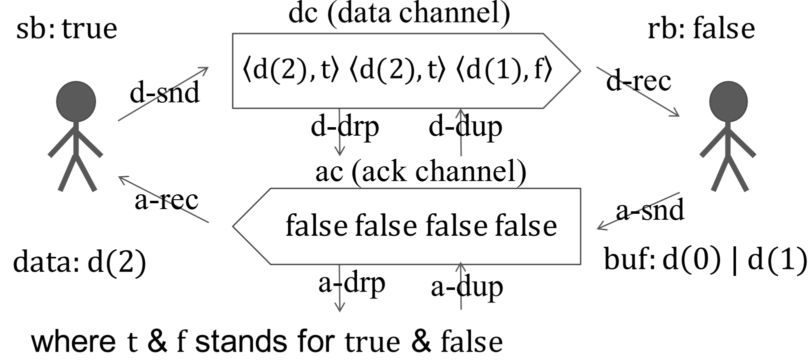


Figure 5.5: A state of ABP

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. 5.5 shows a graphical representation of a state of ABP.

Each state of ABP is formalized as a term {(sb : b_1) (data : d(n)) (rb : b_2) (buf : dl) (dc : q_1) (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. 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}.
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 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.

Experiments

Algorithm 2 shows the pseudo-code for sender, receiver, dropper, and duplicator actions that are executed simultaneously by multiple threads in the program. Actions d-send and a-rec are implemented by the code fragments at line 6 and lines 9 - 13, respectively. Actions a-send and d-rec are implemented by the code fragments at line 20 and lines 23 - 27, respectively. Actions a-drp and d-drp are implemented by the code fragments at lines 32 and 35, respectively. Actions a-dup and d-dup are implemented by the code fragments at lines 40 and 43, respectively. $lock_{dc}$ and $lock_{ac}$ are two locks that are used to protect those atomic code fragments. Note that $lock_{dc}$ may be the same as $lock_{ac}$ when one lock is actually used. We can see that one atomic fragment protected by $lock_{dc}$ and another atomic fragment protected by $lock_{ac}$ can be executed in parallel. Hence, at most two atomic code fragments may be executed by multiple threads in parallel (or simultaneously), although each atomic action (or state transition) in the formal specification is implemented by one atomic code fragment in the program.

We suppose that the sender is to deliver four data to the receiver, the depth of each layer is 100, and DEPTH is unbounded. 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. Table 5.1 shows experimental data in which one lock is used in the program, meaning that the maximum number of transition steps is 1, while table 5.2 shows experimental data in which two locks are used in the program, meaning that the maximum number of transition steps is 2. We can change algorithm 2 to make the program using only one lock easily by replacing two locks $lock_{dc}$ and $lock_{ac}$ with the same lock as mentioned.

When each channel size was 1, one lock experiment took about 5 hours 47 minutes, while two locks experiment took about 8 hours 39 minutes to generate all state sequences with four

Algorithm 2: Sender, Receiver, Dropper, and Duplicator in ABP program

input : ABP – a concurrent program
$lock_{dc}$ – a lock on the data channel
$lock_{ack}$ – a lock on the ack channel

```
1  dc ← empty; ac ← empty;
2  function Sender is
3      data ← 0; sb ← true;
4      while true do
5          request(lock_dc);
6          dc.put(< data, sb >);
7          release(lock_dc);
8          request(lock_ac);
9          if ac.size() > 0 then
10             b ← ac.get();
11             if b ≠ sb then
12                 sb ← ¬sb;
13                 data ← data + 1;
14         release(lock_ac);
15 function Receiver is
16     buf ← 0;
17     rb ← true;
18     while true do
19         request(lock_ac);
20         ac.put(rb);
21         release(lock_ac);
22         request(lock_dc);
23         if dc.size() > 0 then
24             < d, b >← dc.get();
25             if b = rb then
26                 buf.put(d);
27                 rb ← ¬rb;
28         release(lock_dc);
```

workers and check them with Maude on the fly. The number of the state sequences generated is 47,505 and 64,854, respectively. Note that the number of state sequences is the total number of sub-state sequences at each layer without combining sub-state sequences. Maude detected

Algorithm 2: Sender, Receiver, Dropper, and Duplicator in ABP program (continuously)

```
29 function Dropper is
30     while true do
31         request(lock_ac);
32         ac.get();
33         release(lock_ac);
34         request(lock_dc);
35         dc.get();
36         release(lock_dc);
37 function Duplicator is
38     while true do
39         request(lock_ac);
40         ac.duptop();
41         release(lock_ac);
42         request(lock_dc);
43         dc.duptop();
44         release(lock_dc);
```

that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively. If that is the case, a tool component [27] 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 was 2, one lock experiment took about 6 days, while two locks experiment took about 7 days 13 hours 21 minutes to generate all state sequences with four workers and check them with Maude on the fly. The number of state sequences generated is 4,606,719 and 6,611,839, respectively. As is the case in which each channel is 1, Maude detected that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively, due to the flaw intentionally inserted in P_{ABP}.

Table 5.1: Experimental data for ABP program in which one lock is used

Channel size	Worker	Time (d:h:m)	#seqs
1	Worker 1	0:5:33	47,505
	Worker 2	0:5:31	
	Worker 3	0:5:47	
	Worker 4	0:5:47	
2	Worker 1	5:23:27	4,606,719
	Worker 2	6:0:32	
	Worker 3	5:23:55	
	Worker 4	5:23:26	
3	Worker 1	33:17:26	37,403,548
	Worker 2	33:12:11	
	Worker 3	33:15:24	
	Worker 4	33:15:06	

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.

Table 5.2: Experimental data for ABP program in which two locks are used

Channel size	Worker	Time (d:h:m)	#seqs
1	Worker 1	0:8:14	64,854
	Worker 2	0:8:31	
	Worker 3	0:8:14	
	Worker 4	0:8:39	
2	Worker 1	7:13:21	6,611,839
	Worker 2	7:12:30	
	Worker 3	7:11:58	
	Worker 4	7:12:15	
3	Worker 1	38:16:12	54,429,058
	Worker 2	38:16:41	
	Worker 3	38:14:11	
	Worker 4	38:21:34	

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.

Table 5.3: Experimental data for ABP program with various numbers of workers

Channel size	#locks	#seqs	#workers	Time (d:h:m)
1	1	47,505	1	0:16:40
			4	0:5:4
			8	0:2:50
			12	0:2:7
	2	64,854	1	1:0:24
			4	0:7:37
			8	0:4:27
			12	0:3:13

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.

When each channel size was 3, one lock experiment took about 33 days 17 hours 26 minutes, while two locks experiment took about 38 days 21 hours 34 minutes to generate all state sequences with four workers and check them with Maude on the fly. The number of the state sequences generated is 37,403,548 and 54,429,058, respectively. As is the case in which each channel is 1 and 2, Maude detected that some state sequences have adjacent states s and s' such that s cannot reach s' by S_{ABP} in both experiments with one and two state transitions, respectively, due to the flaw intentionally inserted in P_{ABP}.

The experimental data in tables 5.1–5.2 show that the more locks are used, the more time it takes to do testing programs. It is reasonable because the more locks used will introduce more synchronized points in programs from which more state sequences are generated. In addition, our experiments also demonstrate that programmers are able to know the maximum steps in programs for one atomic state transition in formal specifications based on the number of locks used in programs.

If we did not use the proposed approach and simply used JPF to generate state sequences with the same computer, we encountered an out-of-memory error even when each channel size was 1 [112]. It was reported [120] that when each channel was 3 and the number of data delivered was 3 (but not 4), a straightforward use of JPF did not complete a model checking experiment for ABP into which no flaw was intentionally inserted, leading to an out-of-memory error after it took about four days with almost the same computer used in the experiments reported in this thesis. Therefore, the proposed technique can alleviate the out-of-memory situation due to the state space explosion.

We would like to demonstrate further the effectiveness of our parallel specification-based testing for testing concurrent programs by conducting more experiments for ABP case study with our divide & conquer approach to testing concurrent programs with different numbers

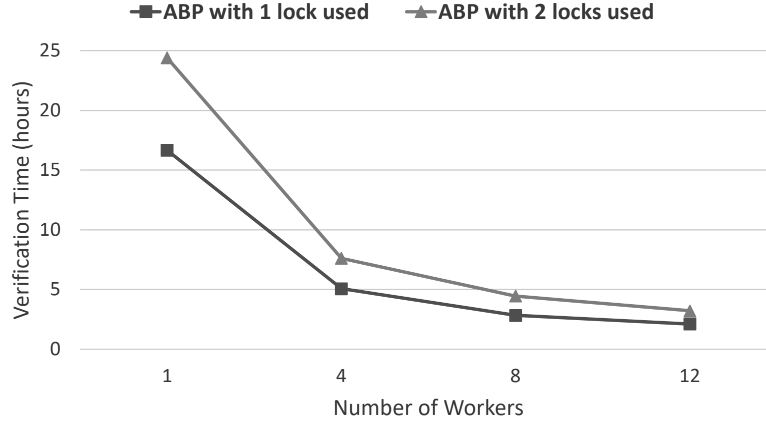


Figure 5.6: Verification time for ABP programs with various numbers of workers

of workers shown in Table. 5.3. Even when the number of workers is 1, our tool successfully completes the experiment, meaning that it alleviates the state space explosion, while the straightforward use of JPF did not as above-mentioned. We use a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory to conduct the experiments because we need to use many workers. For ABP whose number of locks used is 1, it takes 16 hours 40 minutes, 5 hours 4 minutes, 2 hours 50 minutes, and 2 hours 7 minutes to complete generating state sequences and checking them with Maude when the number of workers is 1, 4, 8, and 12, respectively. Meanwhile, for ABP whose number of locks used is 2, it takes 1 day 24 minutes, 7 hours 37 minutes, 4 hours 27 minutes, and 3 hours 13 minutes to complete generating state sequences and checking them with Maude when the number of workers is 1, 4, 8, and 12, respectively. We plot the experimental data in Table. 5.3 on the graph shown in Fig. 5.6. We can see that the verification time improves quickly for ABP programs when we increase the number of workers from 1 to 4, which demonstrates that parallelization is effective for our proposed technique. The improvement for ABP with one and two locks used is about 69.9% and 68.8%, respectively. When we increase the number of workers from 4 to 8 and 8 to 12, the verification time improves as well, although the improvement is slower. This is because the more workers used, the busier the master and the Redis cache need to handle and communicate with workers. Therefore, depending on the power of the machine used to conduct experiments, we may choose a reasonable number of workers when using our tool. In conclusion, we have demonstrated the power of our parallel specification-based testing for concurrent programs.

5.5.2 Revised CloudSync

Introduction

Revised CloudSync is a simplified cloud synchronization protocol in which many PCs would like to exchange messages with a *Cloud* in order. For simplicity, we use natural numbers as

messages. A *PC* may connect to the *Cloud* if and only if both the *PC* and *Cloud* are in an idle state. After connected, the *PC* can fetch the value from the *Cloud* and then update either the value of *Cloud* or the value of the *PC* depending on which value is larger. The following shows how the Revised CloudSync protocol works in detail.

The *Cloud* maintains two pieces of information: *statusc* and *valc* that are the status and value of the *Cloud*, respectively. *statusc* is set to one of *idlec* and *busy* that are labels and *valc* is set to a natural number. *statusc* and *valc* are initially set to *idlec* and a natural number n, respectively. Meanwhile, each *PC* maintains three pieces of information: *statusp*, *valp*, and *tmp* that are the status, value, and temporary value of the *PC*, respectively. *statusp* is set to one of *idlep*, *gotval*, and *updated* that are also labels, and *valp* and *tmp* are set to natural numbers. *statusp*, *val*, and *tmp* are initially set to *idlep*, a natural number, such as *new*, l, and m, and the natural number 0. Note that n, *new*, l, and m are arbitrary natural numbers used. The protocol uses three transition rules.

The first transition rule is *gotval* depicted in Fig. 5.7. A *PC* wants to connect to the *Cloud* if and only if the *statusc* of the *Cloud* is *idlec* and the *statusp* of the *PC* is *idlep*. If that is the case, the *PC* fetches the current *valc* of the *Cloud* and updates the *tmp* of the *PC* to the value fetched; the *statusp* of the *PC* is also updated to *gotval*, while the *statusc* of the *Cloud* is changed to *busy*.

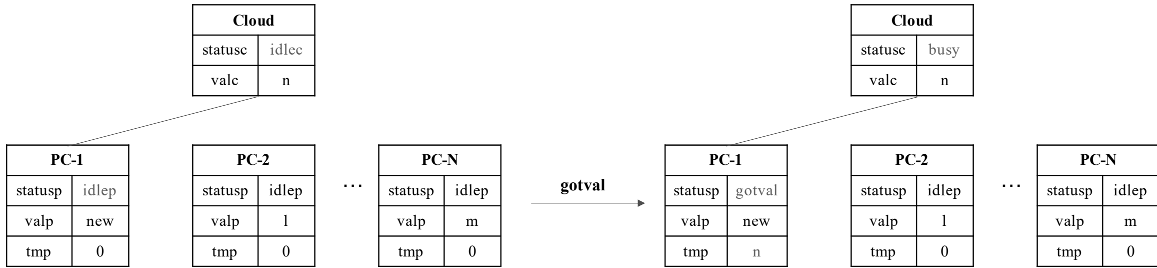


Figure 5.7: gotval transition

The second transition rule is *updated*. If the *tmp* of the *PC* involved is equal or greater than the *valp* of the *PC*, *updated* conducts *update*1 depicted in Fig. 5.8. Otherwise, it conducts *update*2 depicted in Fig. 5.9. Both *update*1 and *update*2 change the *statusp* of the *PC* to *updated*. *update*1 changes the *valp* of the *PC* to n, which is the same as the *tmp* of the *PC* and the *valc* of the *Cloud*, and leaves the other values unchanged, while *update*2 changes both the *valc* of the *Cloud* and the *tmp* of the *PC* to *new* and leaves the other values unchanged. Basically, the *updated* rule guarantees that the *valp* of the *PC* and the *valc* of the *Cloud* maintain the same largest number between two of them after the rule has been applied.

The last transition rule is *gotoidle* depicted in Fig. 5.10. After the *updated* rule has been just carried out by a *PC* and the *Cloud*, the *statusc* of the *Cloud* is *busy*, the *statusp* of the *PC* is *updated* and the *valc* of the *Cloud* and the *valp* and *tmp* of the *PC* have a same value, say *new*. If so, the *gotoidle* rule updates the *statusc* of the *Cloud* back to *idlec*, the *statusp*

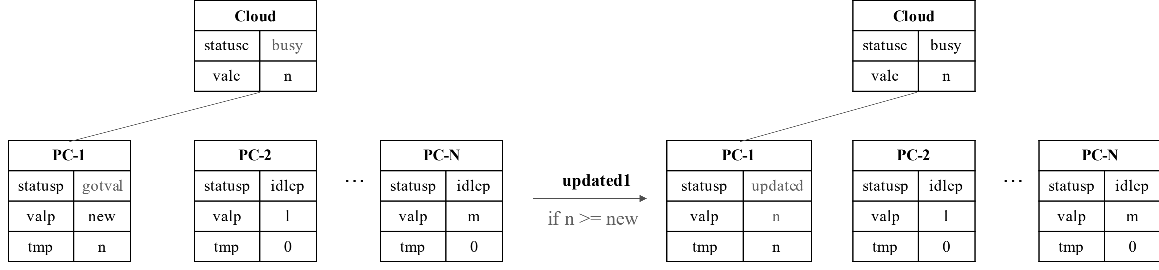


Figure 5.8: updated1 transition

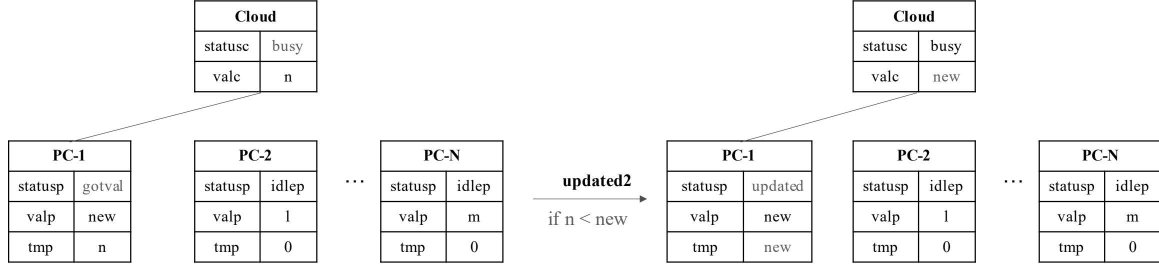


Figure 5.9: updated2 transition

of the *PC* back to *idlep* and the *tmp* of the *PC* back to 0. The last transition rule makes the *Cloud* as well as the *PC* free. From now on, the *Cloud* can freely connect to any *PC* for exchanging messages subsequently.

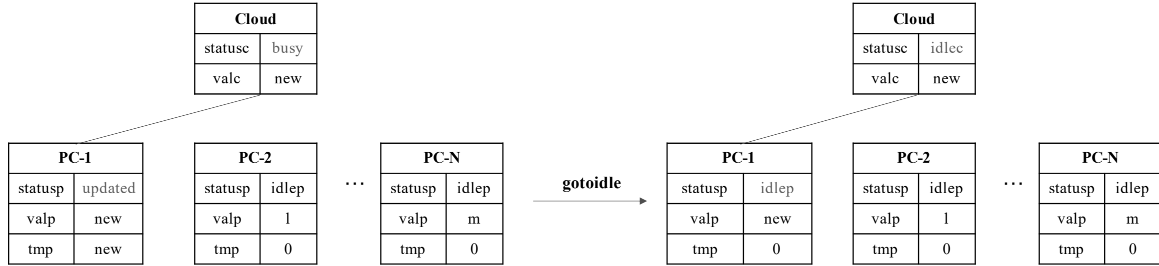


Figure 5.10: gotoidle transition

The three transition rules are specified in rewrite rules as follows:

```
rl [getval] : {(cloud: < idlec,CVal >) (pc[P]: < idlep,PVal,OldCVal >) OCs}
  => {(cloud: < busy,CVal >) (pc[P]: < gotval,PVal,CVal >) OCs} .
crl [update1] : {(cloud: < busy,CVal >) (pc[P]: < gotval,PVal,GotCVal >) OCs}
  => {(cloud: < busy,CVal >) (pc[P]: < updated,GotCVal,GotCVal >) OCs}
  if GotCVal >= PVal .
crl [update2] : {(cloud: < busy,CVal >) (pc[P]: < gotval,PVal,GotCVal >) OCs}
  => {(cloud: < busy,PVal >) (pc[P]: < updated,PVal,PVal >) OCs}
  if GotCVal < PVal .
rl [gotoidle] : {(cloud: < busy,CVal >) (pc[P]: < updated,PVal,OldCVal >) OCs}
  => {(cloud: < idlec,CVal >) (pc[P]: < idlep,PVal,0 >) OCs} .
```

Table 5.4: Experimental data for Revised CloudSync

Depth	Worker	Time (d:h:m)	#seqs
400	One Worker	1:23:22	4,449
400	Worker 1	0:1:36	3,118
	Worker 2	0:1:36	
	Worker 3	0:1:36	
	Worker 4	0:1:33	

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.
- One Worker – we did not use our environment but just used JPF and Maude in a straightforward way.

Words starting with a capital letter, such as P, PVal, GotCVal, CVal, OldCVal, RandVal, and OCs are Maude variables. P and OCs are variables of Pid sort and observable component soups, respectively. PVal, GotCVal, CVal, OldCVal, and RandVal are variables of Nat sort. `idlec` and `busy` are the constants of LabelC sort that denote the possible status values of the *Cloud*. `idlep`, `gotval`, and `updated` are the constants of LabelP sort that denote the possible status values of a *PC*. How to implement CloudSync in Java is described in a document publicly available at Footnote 2, which resides under the *documents* folder.

Experiments

In the Revised CloudSync protocol experiment, three PCs and one Cloud are involved. Because the reachable state space is huge, we need to specify a depth bound to make sure that the experiments terminate. We conducted two experiments: (1) only one worker was used, DEPTH was 400 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 200 (namely that the global DEPTH was 400). The experimental data are shown in Table 5.4.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF workers were checked with Maude on the fly when the maximum number of transition steps was 1 and no bug was detected. For experiment (1), it took 1 day 23 hours and 22 minutes to generate all state sequences and check them with Maude. The number of the state sequences generated is 4,449. For experiment (2), it took 1 hour 36 minutes to generate all state sequences and check them with Maude. The number of the state sequences generated is 3,118. The experimental results show that (2) outperforms (1) and is 29 times faster than (1).

5.5.3 NSLPK

Introduction

Needham-Schroeder Public-Key authentication protocol (NSPK) can be described as three message exchanges:

$$\begin{aligned}
&\text{Challenge: A} \rightarrow \text{B} : \{N_a, A\}_{K_b}\\
&\text{Response: B} \rightarrow \text{A} : \{N_a, N_b\}_{K_a}\\
&\text{Confirmation: A} \rightarrow \text{B} : \{N_b\}_{K_b}
\end{aligned}$$

where A and B are principals called an initiator and a responder, respectively, K_p is the public key owned by a principal p, N_p is a nonce generated by p, and m_{K_p} is the ciphertext obtained by encrypting a message m with K_p. Note that m_{K_p} can only be decrypted by a principal who owns the private key that corresponds to K_p. Lowe found an attack to NSPK and corrected it [45]. The corrected version is called NSLPK that can be described as follows:

$$\begin{aligned}
&\text{Challenge: A} \rightarrow \text{B} : \{N_a, A\}_{K_b}\\
&\text{Response: B} \rightarrow \text{A} : \{N_a, N_b, B\}_{K_a}\\
&\text{Confirmation: A} \rightarrow \text{B} : \{N_b\}_{K_b}
\end{aligned}$$

The difference between NSPK and NSLPK is that the sender principal ID B is used to construct the *Response* message. The ciphertext is obtained by encrypting N_a, N_b and B with the A's public key K_a. Let us describe the formal specification of NSLPK in Maude. We use the following operators as the constructors of observable components:

```
op nw:_ : Soup{Msg} -> OCom [ctor] .
op rands:_ : Soup{Rand} -> OCom [ctor] .
op nonces:_ : Soup{Nonce} -> OCom [ctor] .
op prins:_ : Soup{Prin} -> OCom [ctor] .
```

where `Soup{Msg}`, `Soup{Rand}`, `Soup{Nonce}`, and `Soup{Prin}` are the sorts for soups of messages, random numbers, nonces, and principals, respectively. The `nw` observable component stores all messages sent by principals. The `rands` observable component stores the random numbers available. The `nonces` observable component stores the nonces gleaned by the intruder. The `prins` observable component stores the principals participating in the protocol. The `nw` observable component formalizes the network. We suppose that the network is initially empty and then the `nw` observable component is initially the empty soup denoted *emp*. We also suppose that there are two random numbers initially available, three principals (two trustable ones and one intruder), the `rands` observable component is initially $r1$ $r2$, and the `prins` observable component is initially p q *intrdr*, where p and q denote the two trustable principals and *intrdr* denotes the intruder. Because nothing has been initially gleaned by the intruder, the `nonces` observable component is initially *emp*. The initial state denoted `init` is as follows:

```
op init : -> Config .
eq init = {(nw: emp) (rands: (r1 r2)) (nonces: emp) (prins: (p q intrdr))} .
```

The message exchanges exactly obeying the protocol are specified in rewrite rules as follows:

```
rl [Challenge] : {(nw: NW) (nonces: Ns) (rands: (R Rs)) (prins: (P Q Ps))}
  => {(nw: (m1(P,P,Q,c1(Q,n(P,Q,R),P)) NW))
  (nonces: (if Q == intrdr then (n(P,Q,R) Ns) else Ns fi)) (rands: Rs)
  (prins: (P Q Ps))} .
rl [Response] : {(nw: (m1(P',P,Q,c1(Q,N,P)) NW)) (rands: (R Rs)) (nonces: Ns) OCs}
  => {(nw: (m2(Q,Q,P,c2(P,N,n(Q,P,R),Q)) m1(P',P,Q,c1(Q,N,P)) NW)) (rands: Rs)
  (nonces: (if P == intrdr then (N n(Q,P,R) Ns) else Ns fi)) OCs} .
rl [Confirmation] : {(nw: (m2(Q',Q,P,c2(P,N,N',Q)) m1(P,P,Q,c1(Q,N,P)) NW))
  (nonces: Ns) OCs}
  => {(nw: (m3(P,P,Q,c3(Q,N')) m2(Q',Q,P,c2(P,N,N',Q)) m1(P,P,Q,c1(Q,N,P)) NW))
  (nonces: (if Q == intrdr then (N' Ns) else Ns fi)) OCs} .
```

Messages are formalized in the form m$i(P', P, Q, C)$ for $i = 1, 2, 3$, where P' is the actual sender, P is the seeming sender, Q is the receiver, and C is a ciphertext. P' cannot be seen by principals. If P' is different from P, then P' must be *intrdr* and the message has been forged by the intruder. For example, rewrite rule `Response` says that if there exists a *Challenge* message $m1(P', P, Q, c1(Q, N, P))$ in the network, where $c(Q, N, P)$ denotes the ciphertext obtained by encrypting N and P with the Q's public key, and a random number R is available, then Q replies to the message by putting $m2(Q, Q, P, c2(P, N, n(Q, P, R), Q))$ into the network and the nonce $n(Q, P, R)$ made by Q is gleaned by the intruder if P is the intruder. Note that messages are never deleted from the network. Faking messages by the intruder based on the nonces gleaned and the messages in the network are specified in rewrite rules as follows:

```
rl [fake11] : {(nw: NW) (nonces: (N Ns)) (prins: (P Q Ps)) OCs}
  => {(nw: (m1(intrdr,P,Q,c1(Q,N,P)) NW))  (nonces: (N Ns)) (prins: (P Q Ps)) OCs} .
rl [fake12] : {(nw: (m1(P',P'',Q'',C1) NW)) (prins: (P Q Ps)) OCs}
  => {(nw: (m1(intrdr,P,Q,C1) m1(P',P'',Q'',C1) NW)) (prins: (P Q Ps)) OCs} .
rl [fake21] : {(nw: NW) (nonces: (N N' Ns)) (prins: (P Q Ps)) OCs}
  => {(nw: (m2(intrdr,Q,P,c2(P,N,N',Q)) NW)) (nonces: (N N' Ns)) (prins: (P Q Ps))
  OCs} .
rl [fake22] : {(nw: (m2(Q',Q'',P'',C2) NW)) (prins: (P Q Ps)) OCs}
  => {(nw: (m2(intrdr,Q,P,C2) m2(Q',Q'',P'',C2) NW)) (prins: (P Q Ps)) OCs} .
rl [fake31] : {(nw: NW) (nonces: (N Ns)) (prins: (P Q Ps)) OCs}
  => {(nw: (m3(intrdr,P,Q,c3(Q,N)) NW)) (nonces: (N Ns)) (prins: (P Q Ps)) OCs} .
rl [fake32] : {(nw: (m3(P',P'',Q'',C3) NW)) (prins: (P Q Ps)) OCs}
  => {(nw: (m3(intrdr,P,Q,C3) m3(P',P'',Q'',C3) NW)) (prins: (P Q Ps)) OCs} .
```

For example, rewrite rule `fake21` says that if there are two nonces N and N' gleaned by the intruder, the intruder fakes $m2(intrdr, Q, P, c2(P, N, N', Q))$; rewrite rule `fake22` says that if there is $m2(Q', Q'', P'', C2)$ in the network, the intruder fakes $m2(intrdr, Q, P, C2)$, where P and Q are principals chosen randomly. How to implement NSLPK in Java is described in a document publicly available at Footnote 2, which resides under the *documents* folder.

Experiments

In the experiments, we suppose that there are two non-intruder principals, one intruder, and two random numbers. Because the state space is huge, a bounded depth is used to generate state sequences to make sure that the experiments terminate. We conduct two experiments: (1) only one worker was used, DEPTH was 200 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 100 (namely that the global DEPTH was 200). The experimental data are shown in Table 5.5.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF were checked with Maude on the fly when the maximum number of transition steps was 1 and no bug was detected. For experiment (1), it took over 8 days to generate all state sequences and check them with Maude. The number of the state sequences generated is 1,117,537. For experiment (2), it took about 14 hours to generate all state sequences and check them with Maude. The number of the state sequences generated is 109,933. The experimental results show that (2) outperforms (1) and is about 14 times faster than (1).

We conduct one more experiment for NSLPK with the same configuration as the experiment (2) above, however, a bug is intentionally inserted into the program in which the sender information is not included in the *Response* messages from principals. Our tool can quickly detect the bug just in some seconds and show a warning message as follows:

```
warning ({nw: emp rand: (r1 r2) nonces: emp prins: (p q intrdr)} |
{nw: (m1(p,p,q,c1(q,n(p,q,r1),p))) rand: (r2) nonces: emp prins: (p q intrdr)} |
{nw: (m1(p,p,q,c1(q,n(p,q,r1),p)) m2(q,q,p,c2(p,n(p,q,r1),n(q,p,r2))))
 rand: emp nonces: emp prins: (p q intrdr)} | nil)
```

This is because the message $m2$, a *Response* message, in the network *nw* does not consist of the sender information in the ciphertext $c2$. Hence, Maude cannot parse the message and show the warning message. This demonstrates that our tool can detect the flaw.

5.5.4 Original CloudSync

Introduction

Original CloudSync is the original version of CloudSync protocol that is the same as Revised CloudSync except for some points, which then are described in this sub-section. For conve-

Table 5.5: Experimental data for NSLPK

Depth	Worker	Time (d:h:m)	#seqs
200	One Worker	8:18:09	1,117,537
200	Worker 1	0:14:23	109,933
	Worker 2	0:14:21	
	Worker 3	0:14:35	
	Worker 4	0:14:32	

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.
- One Worker – we did not use our environment but just used JPF and Maude in a straightforward way.

nience, Original CloudSync is called as CloudSync. As described above, Revised CloudSync uses three transition rules that are *gotval*, *updated*, and *gotoidle*. CloudSync uses the same three transition rules described above and one more transition rule: *modval* depicted in Fig. 5.11.

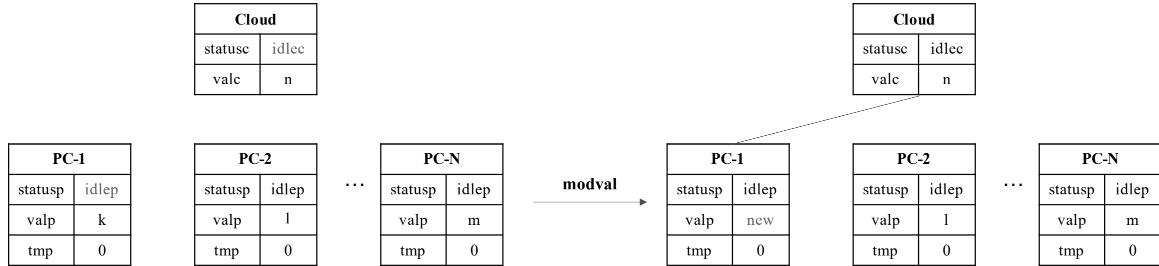


Figure 5.11: modval transition

The *modval* transition rule does not care about the initial *valp* of PCs. Before exchanging messages between the *Cloud* and a *PC*, the *statusc* of the *Cloud* is *idlec* and the *statusp* of the *PC* is *idlep*. The *modval* rule updates the *valp* of the *PC* to a random natural number, say *new*, meaning that the *PC* is willing to be connected to the *Cloud*. For simplicity, the *new* value is the current value of *valp* plus one in our specification. The *modval* transition rule is specified in the following rewrite rule:

```
rl [modvalue] : {(pc[P]: <idlep,PVal,OldCVal>) OCs}
  => {(pc[P]: < idlep,s(PVal),OldCVal >) OCs} .
```

where `s(PVal)` denotes PVal + 1. How to implement Original CloudSync in Java is described in a document publicly available at Footnote 2, which resides under the *documents* folder.

Table 5.6: Experimental data for Original CloudSync

<table>
<tr><th>Depth</th><th>Worker</th><th>Time (d:h:m)</th><th>#seqs</th></tr>
<tr><td>400</td><td>One Worker</td><td>1:18:49</td><td>16,185</td></tr>
<tr><td rowspan="4">400</td><td>Worker 1</td><td>8:19:11</td><td rowspan="4">433,611</td></tr>
<tr><td>Worker 2</td><td>8:18:24</td></tr>
<tr><td>Worker 3</td><td>8:17:41</td></tr>
<tr><td>Worker 4</td><td>8:17:34</td></tr>
</table>

- Time – time taken to generate state sequences with JPF.
- #seqs – the total of state sequences generated.
- One Worker – we did not use our environment but just used JPF and Maude in a straightforward way.

Experiments

In the CloudSync case study, three PCs and one Cloud are involved. A depth bound is used to make sure that the experiments terminate while generating state sequences due to the huge state space. We conduct two experiments: (1) only one worker was used, DEPTH was 400 and the reachable state was not divided, and (2) four workers were used, the reachable state space was divided into two layers and each layer depth was 200 (namely that the global DEPTH was 400). The experimental data are shown in Table 5.6.

In these experiments, we did not intend to insert any bugs into the program. All state sequences generated by JPF were checked with Maude on the fly when the maximum number of transition steps was 1 and no bug was detected. our approach. For experiment (1), it took 1 day 18 hours and 49 minutes to generate all state sequences and check such state sequences with Maude. The number of the state sequences was 16,185. For experiment (2), it took more than 8 days 17 hours to generate all state sequences and check the state sequences with Maude. The number of the state sequences was 433,611. The experimental results show that (1) outperforms (2), meaning that the straightforward use of JPF works more effectively than our tool in this case study.

In CloudSync, whenever *modval* is used, the *valp* of a *PC* being to be connected with the *Cloud* is increased by one before the *PC* and the *Cloud* are connected. Therefore, the reachable state space of CloundSync becomes bigger than that of Revised CloudSync because the *modval* rule produces many different states. It implies that there may be many states located at the bottom of each layer. Furthermore, each *PC* has an equal chance to connect with the *Cloud* and only one *PC* can connect with the *Cloud* at one time. Therefore, the increment of *valp* makes the values of *PCs* turnover and so there may be some cross transitions between states in different sub-state spaces in a layer or even some backward transitions from some states in

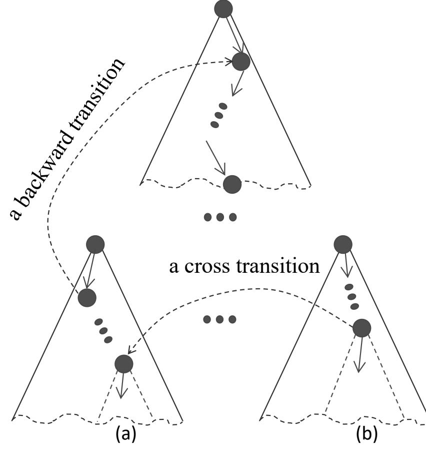


Figure 5.12: Backward and cross transitions

sub-state spaces in a layer to some states in sub-state spaces in previous layers (see Fig. 5.12). We suppose that there are two states located in two different sub-state spaces in a layer such that there is a cross transition between them as shown in Fig. 5.12. The two sets ((a) and (b) shown in Fig. 5.12) of states located at the bottom of the layer reachable from the two states are different because the depths of the states are different. We only store states located at each layer in a big cache to remove duplicate states, but do not store all states in the reachable state space due to the state space explosion problem. Therefore, there are many states being collected at the layer although there are many states shared by many sub-state spaces that start from those collected states. Because of this situation, it makes our environment inefficient when the workers need to explore many states shared by many sub-state spaces.

5.5.5 Threats to Bug Detection

As described, we divide the reachable state space from each initial state to multiple layers, generating multiple sub-state spaces. For each layer l, given a state s_0, JPF instance needs to explore the sub-state space reachable from s_0 up to the layer depth $d(l)$ to generate its sub-state sequences. We suppose that there is a state sequence represented in JPF in the layer l that starts from s_0 as shown in Fig. 5.13. As described above, whenever JPF visits a state, we need to extract the values of observable components from the program under test by looking inside the heap of JPF, and then construct a state representative in the form of state expressions used in a specification. Because the execution units in a program are much finer than those in a specification, and therefore there are some consecutive states in a state sequence represented in JPF that have the same observable component values and so we construct the same state represented in a specification. For example, from s_0 to s_i, they have the same observable component values, and so $s_0^{'}$ is constructed (see Fig. 5.13). Similarly, from s_m to s_{m+j}, they have the same observable component values and so $s_n^{'}$ is constructed. We finally obtain the state sequence $s_0^{'} \dots s_n^{'}$ such that two consecutive states are different. The state sequence is

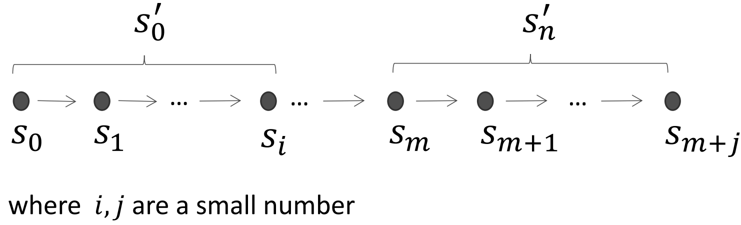


Figure 5.13: A state sequence in a layer l

much shorter than the state sequence $s_0 \ldots s_{m+j}$ represented in JPF and can be checked with Maude. If the last state $s_n^{'}$ does not exist in a cache of states, it is sent to RabbitMQ master (1) to distribute to a worker subsequently. Given $s_n^{'}$, we may generate $s_{m+j}, \ldots, s_{m+j+k}$ or $s_m, \ldots, s_{m+j}, \ldots, s_{m+j+l}$ or something else represented in JPF, where $l < k$, and then this may affect the size of state sequences to be checked by our tool, where the size may be less than DEPTH. Therefore, if a bug locates nearly at the boundary of the bounded depth, there may be a case in which our tool may overlook the bug. One possible way to mitigate this problem is that we can increase the depth of the final layer so that the bug can be covered and detected.

5.6 Extending Our Technique to Test Concurrent Programs with JPF

Our divide & conquer approach to testing concurrent programs is used to test concurrent programs with formal specifications. However, we can extend the technique to test concurrent Java programs without checking if execution sequences generated from Java programs can be accepted by Maude specifications. We then propose a divide & conquer approach to testing concurrent programs with JPF [79] that splits the entire reachable state space of a system under test from each initial state into multiple layers, generating multiple sub-state spaces. For each layer, we generate all states located at the bottom of the layer reachable from each state located at the beginning of the layer. If there is one initial state in the system, there is only one sub-state space in the first layer. Meanwhile, there are as many sub-state spaces in the second layer as the number of states located at the bottom of the first layer. Checking each sub-state space in a layer with JPF is basically independent, making it possible to check each sub-state space in parallel. Some experiments are conducted to demonstrate that the proposed technique can mitigate the state space explosion in model checking that cannot be achieved with the straightforward use of JPF. In this work, we can quickly detect a state of a Java implementation of the NSPK authentication protocol [46] where the nonce secrecy property is broken. However, we were not able to detect a state in which the one-to-many agreement (authentication) property is broken. This is because a state in which the latter property is broken is located at a much deeper position than a state in which the former property is

broken.

To aim at making it possible to detect a state in which the one-to-many agreement property is broken, we propose a parallel stratified random testing for concurrent programs [121] in which random state selection at each layer is used. We suppose that a system (or a protocol) is formally specified in Maude and a concurrent program is written in Java based on the formal specification. To test such a concurrent program, we divide the reachable state space of the concurrent program into L layers such that each layer depth is D_l for $l = 1, \ldots, L$. If there is one initial state in the concurrent program, there is one sub-state space in layer 1. If there are m states located at depth $D_1 + \ldots + D_l$, there are m sub-state spaces in layer $l + 1$ if we do not use any random state selection. We use a percentage δ_l for $l = 1, \ldots, L - 1$ to randomly select some states located at $D_1 + \ldots + D_l$. If n_l states are generated at depth $D_1 + \ldots + D_l$, approximately $0.01 \times \delta_l \times n_l$ states are randomly selected among the n_l states and then we have approximately $0.01 \times \delta_l \times n_l$ sub-state spaces in layer $l + 1$. For a Java implementation of the NSPK authentication protocol, even though we used two layers and each layer depth was 100, it was not complete in three weeks to exhaustively test the concurrent program. On the other hand, when we used three layers, each layer depth was 100 and the percentages for layer 1 & layer 2 were 0.05% (or 0.1%) & 0.05% (or 0.1%), respectively, it was complete in 19h to randomly test the concurrent program. Although we still did not find a state in which the one-to-many agreement property is broken, we made some progress toward making it possible to detect such a violated state.

5.7 Limitations

As shown in the Original CloudSync case study, the running performance of our tool cannot outperform the straightforward use of JPF because there may have some long backward transitions and cross transitions between sub-state spaces. Indeed, the main drawback of our technique/tool is that it cannot handle long backward transitions in programs under test because if a program has such long backward transitions, some sub-state spaces at the very last layer may not be small enough compared with the original reachable state space. Moreover, if there are some cross transitions between sub-state spaces, our tool needs to verify many states again, making the running performance of our tool degrade. Therefore, we should avoid long backward transitions in programs to make the best use of our tool/technique. The cross transitions may arise from a symmetry of threads/processes, making some states likely to be shared by some sub-state spaces in our tool/technique. One possible way to mitigate it is to use the symmetric reduction technique [122] to remove replicated structures produced by such symmetry in programs under test.

5.8 Summary

We have proposed a specification-based testing technique for concurrent programs in a stratified way. The proposed technique could be processed naturally in parallel, which has been utilized by the tool supporting the technique. The experiments reported in this chapter demonstrate that the proposed technique can mitigate the state space explosion problem and largely improve the timing performance for testing for all cases except for one, which cannot be achieved with the straightforward use of only one JPF instance. Besides, we have described how to extend the technique to test concurrent Java programs without checking if execution sequences generated from Java programs can be accepted by Maude specifications. We have made some progress toward making it possible to detect a violated state located at a deep depth. The tool supporting the technique is dedicated to Java programs. However, our technique can be applied to other programming languages provided that we have a model checker for the language concerned and can interact with the model checker as we have done with JPF.

The experiments reported in this chapter demonstrate that concurrent programs in which there are no long lasso loops can be effectively tackled with the proposed technique, while those in which there exist long lasso loops cannot. We reported on totally four case studies. The three programs can be tackled well with our tool, while one cannot. Accordingly, a non-small number of concurrent programs would be likely to belong to the first group if not all. However, we need to conduct some more case studies in which some other concurrent programs will be tackled with our tool in order to make sure that the proposed technique and our tool supporting it can mitigate the state space explosion reasonably well.

Regarding bugs that can be found by our technique/tool, only invariant properties [62] that hold in the whole reachable state spaces of programs are taken into account. Because it is enough to check each reachable state, then splitting the reachable state spaces into multiple layers and generating multiple sub-state spaces never lose any reachable states. Therefore, checking states on the sub-state spaces is equivalent to checking states on the original reachable state space that guarantees the correctness of our approach. Currently, we only consider finite state sequences generated from programs, so we cannot detect any liveness property flaws. As one piece of our future work, we will use some semantics of temporal logics defined over finite state sequences [123, 124] and extend the technique/tool so that some liveness properties can be tested.

Chapter 6

A Parallel Version of a Support Tool for the Divide & Conquer Approach to Leads-to Model Checking

The $L+1$-layer divide & conquer approach to leads-to model checking ($L+1$-DCA2L2MC) [39] is a new technique to mitigate the state space explosion in model checking. As shown by the name, $L+1$-DCA2L2MC is dedicated to leads-to properties. This chapter describes a parallel version of $L+1$-DCA2L2MC and a tool that supports it. In a temporal logic called UNITY designed by Chandy and Misra, the leads-to temporal connective plays an important role and many case studies have been conducted in UNITY, demonstrating that many systems requirements can be expressed as leads-to properties. Besides, Dwyer et al. [41] showed some statistics on the usage distribution of the various patterns in property specifications in which the leads-to property (or the response pattern) had the highest proportion. Therefore, it is worth focusing on leads-to properties. This chapter also reports on some experiments that demonstrate that the tool can increase the running performance of model checking. Counterexample generation is one of the main tasks in the tool that can be optimized to improve the running performance of the tool to some extent. This chapter then proposes a technique to generate all counterexamples at once that is based on the Tarjan algorithm, implemented in C++, and integrated into Maude, a language and system based on rewriting logic, so that users can use it easily. Some experiments are conducted to demonstrate the power of the technique that can improve the running performance of the tool. Furthermore, layer configuration selection affects the running performance of the tool. Therefore, this chapter then proposes an approach to finding a good layer configuration for the tool with an analysis tool that supports the approach. Some experiments are conducted to demonstrate the usefulness of the analysis tool as well as the approach for layer configuration selection.

6.1 Introduction

$L+1$-DCA2L2MC splits the reachable state space from each initial state into $L+1$ layers, where L is a positive natural number, generating multiple sub-state spaces. Model checking experiments are then carried out for each sub-state space instead of the original reachable state space. If each sub-state space is much smaller than the original reachable state space, then it is feasible to conduct a leads-to model checking even though it is infeasible to directly conduct it for the original reachable state space due to the state space explosion problem. We have also developed a support tool written in Maude for a sequential version of $L+1$-DCA2L2MC [125]. Because model checking experiments for multiple sub-state spaces are basically independent, it is possible to parallelize $L+1$-DCA2L2MC from which we can mitigate the state space explosion problem (space challenge) and improve the running performance of model checking (time challenge) as well.

This chapter demonstrates that $L+1$-DCA2L2MC can be naturally parallelized by describing a parallel version of $L+1$-DCA2L2MC and a tool that supports it. The tool has been built in Maude [13] as an implementing language. Maude is one direct successor language of OBJ3 [126], an algebraic specification language, and based on rewriting logic [127] as its theoretical foundation. Maude is equipped with reflective programming (meta-programming) facilities, making it possible to develop tools, such as Real-Time Maude [128] in which real-time systems can be formally specified and verified. The main reason why we have selected Maude for the tool development is that many tools have been developed in Maude, we use Maude as a formal specification language and an LTL model checker, and Maude LTL model checker is comparable to Spin in terms of model checking running performance [15]. The architecture of the tool is a master-worker model where one master and multiple workers are involved. The tool uses shared caches maintained by the master and local caches maintained by each worker so as to avoid making unnecessary duplications of jobs and reduce the communication cost between the master and workers. The tool uses two queues of jobs and worker identifiers so as to distribute (or assign) jobs to workers in a well-balanced way. It is an option to use an existing collection of open-source software facilities that make the best use of parallel/distributed environments, such as Apache Hadoop that uses the MapReduce programming model [110] to implement a parallel version of the support tool for $L+1$-DCA2L2MC. However, we did not take the option. This is because we do not use a large number of computers, making it unnecessary to consider computer failure and/or stragglers, Maude is equipped with facilities that can be used to build parallel systems, it is necessary to convert data formats from Maude to Apache Hadoop and vice versa, etc. Note that MapReduce is basically a master-worker model. The chapter also reports on some case studies that demonstrate that the parallel $L+1$-DCA2L2MC tool increases the running performance of model checking for all examples used except one that is a simple mutual exclusion protocol, namely test&set. We compare our tool with the straightforward use of Maude LTL model checker when we use enough amount of memory for the model

checker. Note that the parallel $L+1$-DCA2L2MC tool outperforms its sequential version for all examples used and both versions of our tools can conduct model checking experiments even though Maude LTL model checker cannot due to the state space explosion. The support tool is publicly available at Footnote 4.

Counterexample generation is one of the main tasks in our tool that can be optimized to improve the running performance of our tool. In our previous work [125], we need to generate counterexamples one by one at each layer, which makes our tool slower because we need to ask Maude to conduct model checking experiments as many times as the number of counterexamples. Maude LTL model checker uses a nested depth-first search algorithm to search for an accepting cycle in a product automaton [15], which terminates and returns a counterexample as soon as it is found. We would like to generate all counterexamples at once to improve the running performance of our tool. The Tarjan algorithm [129] is well-known as an effective algorithm to find all strongly connected components (SCCs) in a graph while each SCC can be regarded as a counterexample. Therefore, we propose a technique to generate all counterexamples at once based on the Tarjan algorithm. We develop a new model checker to support the technique by using facilities existing in Maude LTL model checker, such as building the property automaton of the negation of a correctness property. The new model checker is implemented in C++ and integrated into Maude so that users can use it easily. Some experiments are conducted to demonstrate the power of the technique that can improve the running performance of our tool. The implementation is publicly available Footnote 3.

As described $L+1$-DCA2L2MC splits the reachable state space from each initial state into $L+1$ layers, where L is a positive natural number, generating multiple sub-state spaces (see Fig. 6.1). For each sub-state space reachable from a state in a non-final layer, we need to generate states reachable from the state up to the layer depth, a positive natural number, and located at the bottom of the layer for the next layer, while for each sub-state in the final layer whose depth is unbounded, a model checking experiment is carried out. The number of layers and each layer depth for non-final layers affect the running performance of our tool. Therefore, we propose an approach to finding good layer configurations for the tool as follows. We start with a two-layer configuration d_1, where d_1 is the depth of layer 1, such that d_1 is small, which can avoid the state space explosion while generating states and keep the product automaton reasonably small so as to take advantage of our new model checker above. We generate states in layer 1 and measure the time taken to do so. We would like to know how much time it takes to conduct model checking experiments for the final layer (layer 2 this time). Even in layer 1 whose depth is small, there may be many states. Thus, it is not reasonable to tackle all states. Hence, we randomly select some states to conduct model checking for the final layer and roughly estimate the time taken to conduct model checking experiments for the final layer. We should increase the coverage of states for better estimation. If we tackle a large number of states in sequence, it may take an unreasonable amount of time. Therefore, we can take advantage of parallelization by running multiple instances of a model checker. Each instance

is in charge of a small number of randomly selected states to estimate the verification time for the final layer. If multiple instances are used and each instance selects some states randomly and independently, it is possible to increase the coverage of states in a reasonable amount of time. Indeed, the more states are selected and the more instances are used, the more confident we are in the estimated verification time. If the estimated verification time for the final layer is too large, we add one more layer with a small depth d_2, the depth of layer 2. The next layer configuration is d_1 d_2, a three-layer configuration. From states generated in layer 1, we generate states in layer 2 and measure the time taken to do so. We estimate the verification time for the final layer (layer 3 this time). If the estimated verification is not too large, the current layer configuration d_1 d_2 is a good layer configuration candidate. If the time taken to generate states in the first two layers is not small enough, we stop the layer configuration selection and select d_1 d_2 as a good layer configuration. Otherwise, we can keep on finding a better layer configuration by adding one more layer with a small depth d_3, when the next layer configuration is d_1 d_2 d_3, a four-layer configuration. We generate states in layer 3 and measure the time taken to do so. We estimate the verification time for the final layer (layer 4 this time). We compare the (estimated) total verification time (the time taken to generate states plus the estimated verification time for the final layer) for d_1 d_2 and the total verification time for d_1 d_2 d_3. If the former is smaller than the latter, we stop layer configuration selection and select d_1 d_2 as a good layer configuration. Otherwise, we keep on doing what has been described. We develop an analysis tool to support the approach that uses a server as the central place to handle commands that are sent by a commander. The tool is developed in Maude and uses sockets to communicate between the server and the commander. Especially, for the server, we use two new features in Maude: timer and meta-interpreter. Some experiments are conducted to demonstrate the usefulness of our analysis tool as well as our approach. The analysis tool is publicly available at Footnote 4, which resides under the *analysis* folder.

6.2 The $L+1$-Layer Divide & Conquer Approach to Leads-to Model Checking

Our research group has proposed the $L+1$-layer divide & conquer approach to leads-to model checking (or $L+1$-DCA2L2MC) [39] that aims to mitigate the state space explosion in model checking. In this section, let us briefly summarize $L+1$-DCA2L2MC. Given a Kripke structure K, $L+1$-DCA2L2MC splits the reachable state space from each initial state $s_{d_0} \in I$ into $L+1$ layers (where L is a non-zero natural number) as shown in Fig. 6.1. Let $d(i)$ be the depth of layer i for $i = 0, 1, \ldots, L, L+1$. We suppose that there virtually exists layer 0 such that $d(0) = 0$. $d(i)$ is a non-zero natural number if $i = 1, \ldots, L$. $d(L+1) = \infty$. Let d_i be $d(0) + \ldots + d(i)$ for $i = 0, \ldots, L$, namely that d_i is the depth of the bottom of layer i (or the depth of the top of layer $i+1$) from the initial state. d_i is also referred to as the depth of layer $i+1$ (from the

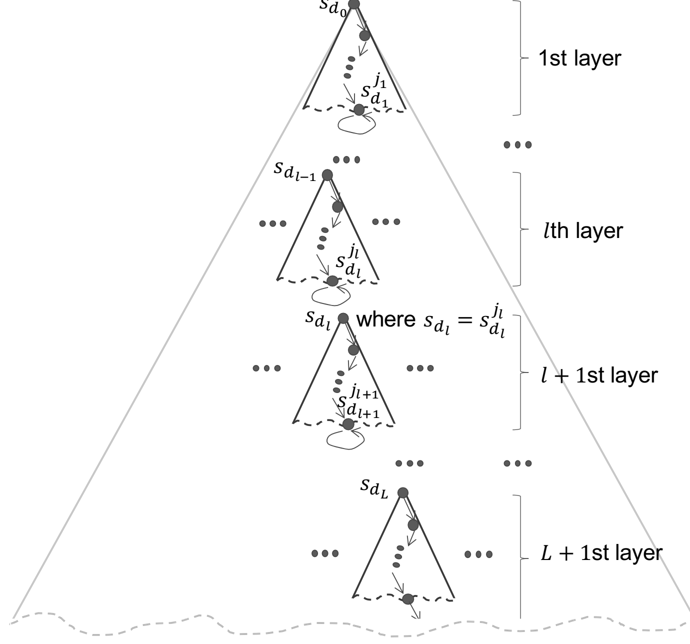


Figure 6.1: Split of the reachable state space into $L + 1$ layers

initial state). States located at the depth d_i are called beginning states of layer $i+1$ (or ending states of layer i) and also states (located) at layer $i+1$. Beginning states of layer $i+1$ are also called states (located) at layer $i+1$. The depth of a state is the depth from the initial state where the state is located. The depth of a beginning state of layer $i+1$ (or an ending state of layer i) is d_i. Let us suppose that the depth of a state is d_i, which implies that the state is a beginning state of layer $i+1$, and then the next depth of the state is d_{i+1}. In Fig. 6.1, s_{d_i} denotes a beginning state of layer $i+1$ for $i = 0, \ldots, L$, and $s_{d_i}^{j_i}$ denotes an ending state of layer i for $i = 0, \ldots, L$, although we do not use $s_{d_0}^{j_0}$ (which is an ending state of layer 0, a beginning state of layer 1, and the same as s_{d_0}) in the figure. $s_{d_i}^{j_i}$ is also a beginning state of layer $i+1$, such as s_{d_l} that equals $s_{d_l}^{j_l}$ in Fig. 6.1. For each ending state of each non-final layer, such as $s_{d_1}^{j_1}$, $s_{d_l}^{j_l}$, and $s_{d_{l+1}}^{j_{l+1}}$, a self-transition $s \to s$ is added so that we can have paths in the layer because we need to have them to conduct model checking experiments. It is unnecessary to add any self-transitions to the final layer because the depth $d(L+1)$ of the final layer is ∞.

Let Π_i for $i = 1, \ldots, L$ be the set of all paths constructed as described in layer i. Let Π_{L+1} be the set of all paths in the final layer $L+1$. For a path $\pi \in \Pi_i$ for $i = 1, \ldots, L$, let last(π) be the state in π that repeats forever that is an ending state of layer i. For an ending state s of layer i (or a beginning state s of layer $i+1$), let take(Π_{i+1}, s) for $i = 1, \ldots, L$ be the set of all paths that start from s. For state propositions φ_1, φ_2, $K, s_{d_0} \models \varphi_1 \leadsto \varphi_2$ can be checked in a stratified way as shown in Algorithm 3. When $K, s_{d_0} \not\models \varphi_1 \leadsto \varphi_2$, Algorithm 3 just returns Failure. A small modification, however, makes it possible to construct and return a counterexample [125]. For $\pi \in \Pi_1$, if $K, \pi \not\models \varphi_1 \leadsto \varphi_2$, then last($\pi$) is called a counterexample state (of layer 1). Let $Cx\Pi_2$ be the set of all $\pi \in \Pi_2$ such that $K, \pi' \not\models \varphi_1 \leadsto \varphi_2$ for some

Algorithm 3: $L + 1$-DCA2L2MC

input : K – a Kripke structure
$s_{d_0} \in I$ – an initial state of K
φ_1, φ_2 – state propositions
$d(1) \ \ldots \ d(L)$ – a list of non-zero natural numbers, where L is a non-zero natural number

output: Success ($K, s_{d_0} \models \varphi_1 \rightsquigarrow \varphi_2$) or Failure ($K, s_{d_0} \not\models \varphi_1 \rightsquigarrow \varphi_2$)

```
1  CxΠ_1 ← ∅
2  forall l ∈ {1, ..., L} do
3      CxΠ_{l+1} ← ∅
4      forall π ∈ Π_l do
5          if K, π ⊭ φ1 ⇝ φ2 then
6              CxΠ_{l+1} ← CxΠ_{l+1} ∪ take(Π_{l+1}, last(π))
7      forall π ∈ CxΠ_l do
8          if K, π ⊭ ◊φ2 then
9              CxΠ_{l+1} ← CxΠ_{l+1} ∪ take(Π_{l+1}, last(π))
10 forall π ∈ Π_{L+1} do
11     if K, π ⊭ φ1 ⇝ φ2 then
12         return Failure
13 forall π ∈ CxΠ_{L+1} do
14     if K, π ⊭ ◊φ2 then
15         return Failure
16 return Success
```

$\pi' \in \Pi_1$ and $\pi(0) = \mathrm{last}(\pi')$. For $\pi \in \Pi_i$ for $i = 2, \ldots, L$, if $K, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$, then $\mathrm{last}(\pi)$ is called a counterexample state (of layer i); for $\pi \in Cx\Pi_i$ for $i = 2, \ldots, L$, if $K, \pi \not\models \Diamond\, \varphi_2$, then $\mathrm{last}(\pi)$ is also called a counterexample state (of layer i), where $Cx\Pi_{i+1}$ is the union of (a) and (b): (a) the set of all $\pi \in \Pi_{i+1}$ such that $K, \pi' \not\models \varphi_1 \rightsquigarrow \varphi_2$ and $\pi(0) = \mathrm{last}(\pi')$ for some $\pi' \in \Pi_i$ and (b) the set of all $\pi \in \Pi_{i+1}$ such that $K, \pi' \not\models \Diamond\, \varphi_2$ and $\pi(0) = \mathrm{last}(\pi')$ for some $\pi' \in Cx\Pi_i$.

For each layer $l \in \{1, \ldots, L+1\}$ we check $K, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$ and $K, \pi \not\models \Diamond\varphi_2$ for each $\pi \in \Pi_l$ and $\pi \in Cx\Pi_l$, respectively. Note that $Cx\Pi_l \subseteq \Pi_l$ because Π_l is the set of all ending states of layer l. States in Π_l are called all-states (in layer l), while those in $Cx\Pi_l$ are called cx-states (in layer l). At the beginning of each layer $l \in \{1, \ldots, L\}$, $Cx\Pi_{l+1}$ is initially set to $\emptyset$ at line 3. The code fragment at lines 4–6 checks $K, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$ for each path $\pi \in \Pi_l$ in the lth layer. If that is the case, $Cx\Pi_{l+1}$ is updated with all paths reachable from the state $last(\pi)$ in the $l+1$st layer. The code fragment at lines 7–9 checks $K, \pi \not\models \Diamond\varphi_2$ for each path $\pi \in Cx\Pi_l$ of the lth

layer such that $last(\pi)$ is a counterexample state. If that is the case, $Cx\Pi_{l+1}$ is updated with all paths reachable from the state $last(\pi)$ in the $l+1$st layer. The code fragment at lines 10–15 checks $K, \not\models \varphi_1 \leadsto \varphi_2$ and $K, \pi \not\models \Diamond\varphi_2$ for each path $\pi \in \Pi_{L+1}$ and $\pi \in Cx\Pi_{L+1}$ in the $L+1$st layer where $d(L+1)$ is ∞, respectively. If that is the case, `Failure` is returned. Otherwise, `Success` is returned.

We mean conducting a model checking experiment for the sub-state space from the beginning state of layer l by "model checking a state at layer l", where a state is a beginning state of layer l, or just "model checking a state" if layer l can be understood from the context or it is unnecessary to think a specific layer l. Model checking a state at an intermediate layer is to generate all counterexample and non-counterexample states located at the next depth reachable from the state, while model checking a state at the final layer is to judge if there is a counterexample or not.

6.3 Parallel $L+1$-DCA2L2MC and Its Tool Support

The support tool is implemented in Core Maude and a part of Full Maude. Full Maude is an extension of (Core) Maude written in Maude itself that provides support for new types of systems, such as object-based systems. We use object-based programming powered by Full Maude that can model an object-based system, where objects can communicate with each other via message passing. In addition, Maude also supports communicating with external objects by using sockets so that objects inside an object-based system can interact with different objects inside another object-based system. We adopt such functionalities to make a parallel version of $L+1$-DCA2L2MC based on a master-worker model, which is elaborated in this section.

$L+1$-DCA2L2MC can check $K \models \varphi_1 \leadsto \varphi_2$ in a stratified way [39]. The reachable state space from each initial state is split into multiple layers ($L+1$ layers) and multiple smaller sub-state spaces are generated. For each smaller sub-state space, we check if $\varphi_1 \leadsto \varphi_2$ and/or $\Diamond\varphi_2$ holds. Such multiple model checking problems for sub-state spaces are almost independent. Those for sub-state spaces located in one layer, such as the final layer, are totally independent. This is one of the most important advantages of the technique, making it possible to conduct a leads-to model checking experiment in parallel. We use a master-worker model to make a parallel version of $L+1$-DCA2L2MC, which needs to consider the two issues: well-balanced job distribution and low communication overhead between a master and workers. In our tool, a master maintains shared caches for all workers, while each worker also maintains local caches. The use of the shared caches substantially prevents jobs that have been processed from being assigned to workers, while the use of the local caches prevents jobs that have been processed from being made by workers. The use of both shared and local caches reduces the communication cost between the master and workers. The very initial job is made by the master, while all the other jobs are made by workers and sent to the master unless the jobs have been processed by

the workers themselves, which can be checked by using the local caches. Jobs are assigned to workers by the master unless the jobs have been tackled, which can be checked by using the shared caches.

For the sub-state space that starts from a counterexample state, it is necessary to check $\Diamond\varphi_2$ as well as $\varphi_1 \leadsto \varphi_2$, while for the sub-state space that starts from a non-counterexample state, it is enough to check $\varphi_1 \leadsto \varphi_2$. Based on what has been just described, jobs are classified into two types: `cx` and `all`. Given a beginning state s_{d_l} of layer $l+1$, a log list of the state is a list $< s_{d_{l-1}} : d_l > ... < s_{d_1} : d_2 >< s_{d_0} : d_1 >$ of state & natural number pairs, where s_{d_i} is a beginning state of layer $i+1$ and d_{i+1} is the next depth of the state s_{d_i} for $i = 0, \dots, l-1$; the list denotes how to get to s_{d_l} from the initial state s_{d_0}. The log list of each initial state is nil.

There are three kinds of messages exchanged by the master and workers: job, getJob, and stop. A job message is in the form of a 5-tuple that consists of a beginning state of a layer, the type of the job, the depth of the state, the next depth of the state, and a log list of the state. The type of the job is either `all` or `cx`. Let $l+1$ be the layer concerned. For each state in $Cx\Pi_l$, one job whose type is `cx` is made. For each state in Π_l, one job whose type is `all` is made. Note that for each state in $Cx\Pi_l$, two jobs whose types are `cx` and `all`, respectively, are made because $Cx\Pi_l \subseteq \Pi_l$. A job message is sent by the master to a worker, distributing (or assigning) a job to the worker. Meanwhile, a job message is sent by a worker to the master, delivering a job made by the worker to the master. A getJob message is sent by a worker to the master, asking the master to assign a job to the worker. A stop message is sent by a worker to the master, notifying the master that the worker has found a counterexample in the final layer, and so the tool then terminates. Both getJob and stop messages do not have any parameters.

The master is in charge of collecting jobs from workers, checking whether each of the jobs has been processed, and distributing (or assigning) unprocessed jobs to workers. Besides, the master can stop the tool when a worker has found a counterexample in the final layer or there is no unprocessed job left. Meanwhile, each worker is responsible for processing a job assigned to it by the master. If a state encapsulated in a job is a beginning state of a non-final layer, a worker generates all beginning states of the next layer reachable from the state and all counterexample states among them by checking $\varphi_1 \leadsto \varphi_2$ in case of an all-state. In case of a cx-state, a worker only generates all counterexample states by checking $\Diamond\varphi_2$. The worker may then construct new jobs and send them to the master as job messages. If such a state is a beginning state of the final layer, a worker checks $\varphi_1 \leadsto \varphi_2$ or $\Diamond\varphi_2$ based on the type of the state that is either `all` or `cx`. If the worker finds a counterexample, the worker sends a stop message to the master. At last, when a worker has completed a job, the worker requests a new job by sending a getJob message to the master. The master uses a queue data structure to distribute jobs to workers so that job distribution can be well-balanced, which means that all workers are mostly processing jobs all the time except the beginning and ending of a model checking experiment. For both master and workers to check whether jobs have been already processed, they use map data structures as caches, where the depth of a state, a natural number, is used as a key and a set

Algorithm 4: Job Scheduling by a Master

input : K – a Kripke structure
$s_{d_0} \in I$ – an initial state of K
φ_1, φ_2 – state propositions
$d_1 ... d_L$ – a list of non-zero natural numbers,
where L is a non-zero natural number, $d_0 = 0$, $d_{L+1} = \infty$
N – a number of workers

output: Success (K, $s_{d_0} \models \varphi_1 \rightsquigarrow \varphi_2$) or Failure (K, $s_{d_0} \not\models \varphi_1 \rightsquigarrow \varphi_2$)

```
1  AllStates ← empty; CxStates ← empty;
2  jobs ← empty; workers ← empty;
3  JOB ← (s_{d0}, all, d0, d1, nil);
4  if s_{d0} ∉ AllStates[d0] then
5  |   enqueue(jobs, JOB);
6  |   AllStates[d0] ← AllStates[d0] ∪ s_{d0};
7  while True do
8  |   for k ← 1 to N do
9  |   |   if DATA ← recv(worker_k) then
10 |   |   |   if DATA = getJob then
11 |   |   |   |   enqueue(workers, worker_k)
12 |   |   |   else if DATA = stop then
13 |   |   |   |   closeConnection();
14 |   |   |   |   return Failure;
15 |   |   |   else
16 |   |   |   |   (s_{dl}, type, d_l, d_{l+1}, log) ← DATA;
17 |   |   |   |   if type = all and s_{dl} ∉ AllStates[d_l] then
18 |   |   |   |   |   enqueue(jobs, DATA);
19 |   |   |   |   |   AllStates[d_l] ← AllStates[d_l] ∪ s_{dl};
20 |   |   |   |   if type = cx and s_{dl} ∉ CxStates[d_l] then
21 |   |   |   |   |   enqueue(jobs, DATA);
22 |   |   |   |   |   CxStates[d_l] ← CxStates[d_l] ∪ s_{dl};
23 |   while not isEmpty(workers) and not isEmpty(jobs) do
24 |   |   worker ← dequeue(workers);
25 |   |   job ← dequeue(jobs);
26 |   |   send(worker, job);
27 |   if isEmpty(jobs) and size(workers) = N then
28 |   |   closeConnection();
29 |   |   return Success;
```

of states located at the depth is used as a value.

Algorithm 4 shows the pseudo-code for job scheduling conducted by the master. `jobs` and `workers` are queue data structures. `jobs` contains jobs that are distributed to workers, while `workers` contains worker identifiers that are requesting jobs. `AllStates` and `CxStates` are map data structures used by the master as shared caches for all workers and are used to check whether `all` and `cx` types of jobs have been processed, respectively. Initially, `AllStates` and `CxStates` are set to the empty map at line 1, while `jobs` and `workers` are set to the empty queue at line 2. The master starts the job scheduling by building an initial job, namely `JOB` at line 3, a 5-tuple of the initial state s_{d_0}, the type `all`, the depth d_0, the next depth d_1, and the log list nil for the first layer exploration. Given the key d_0 and the state s_{d_0}, it is checked if `AllStates` has a set of states whose key is d_0 such that the set contains s_{d_0}. It is not the case because `AllStates` is initially empty. `JOB` is enqueued to `jobs` and the state s_{d_0}, together with the key d_0, is added to the shared cache `AllStates`. For each $worker_k$, whenever the master receives `DATA` from $worker_k$, which is one of the three kinds of messages described above, it checks if `DATA` is getJob, meaning that the worker is requesting a job. If so, $worker_k$ is enqueued to `workers` so that a job can be assigned to $worker_k$. If `DATA` is stop, meaning that $worker_k$ has found a counterexample in the final layer, the master closes all connections from all workers and `Failure` is returned as the result of Algorithm 4. When `DATA` is neither getJob nor stop, meaning that a job has been made and sent from $worker_k$, the master deconstructs `DATA` into a 5-tuple at line 16. The code fragment at lines 17 - 22 checks if the state in the 5-tuple does not exist in either `AllStates` or `CxStates` based on the type of the job. If no, the job received from $worker_k$ is enqueued to `jobs` and the state, together with its key, the depth of the state, is added to the corresponding shared cache. Otherwise, the job is discarded because it has been already processed. The code fragment at lines 23 - 26 checks if `workers` and `jobs` are not empty. If that is the case, the master dequeues `workers` and `jobs` to obtain a job and a worker identifier and assigns the job to the worker by sending a job message to the worker. Termination detection can be done by checking if `jobs` is empty and the size of `workers` is equal to the number of workers at line 27, meaning that there are neither unprocessed jobs left nor jobs being processed by workers. If so, the master stops Algorithm 4, meaning that the tool terminates, and returns `Success` as the result of Algorithm 4.

Algorithm 5 shows the pseudo-code for job handling conducted by workers. Each worker maintains `AllStates` and `CxStates` as its local caches, which are map data structures and initially set to empty. Note that the shared caches `AllStates` and `CxStates` maintained by the master are almost the same as the union of all local caches `AllStates` and the one of all local caches `CxStates`, respectively. A worker may make a job that has been processed by the worker. Local caches are used to prevent such jobs from being sent to the master by workers. Multiple different workers may make the same job. Local caches cannot prevent such a job from being sent to the master by multiple workers. The master uses the shared caches to prevent such a job from being distributed (or sent) to workers. This is why we use both shared and local caches. Workers start the job handling by sending a getJob message to the master to request a

Algorithm 5: Job Handling by Workers

input : K – a Kripke structure, φ_1, φ_2 – state propositions
$d_1 ... d_L$ – a list of non-zero natural numbers, where L is a non-zero natural number
$d_0 = 0$, $d_{L+1} = \infty$

output: a counterexample if exists

1 $AllStates \leftarrow empty$; $CxStates \leftarrow empty$;
2 send(server, getJob);
3 **while** *isOpen()* **do**
4   **if** $DATA \leftarrow recv(server)$ **then**
5     $(s_{d_l}, type, d_l, d_{l+1}, log) \leftarrow DATA$;
6   **if** $type = all$ **then**
7   **if** $d_{l+1} \neq \infty$ **then**
8   **forall** $\pi \in take(\Pi_{l+1}, s_{d_l})$ **do**
9           $s_{d_{l+1}} \leftarrow last(\pi)$;
10   **if** $s_{d_{l+1}} \notin AllStates[d_{l+1}]$ **then**
11             $JOB \leftarrow (s_{d_{l+1}}, all, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log)$;
12   send(server, JOB);
13             $AllStates[d_{l+1}] \leftarrow AllStates[d_{l+1}] \cup s_{d_{l+1}}$;
14   **if** $K, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$ *and* $s_{d_{l+1}} \notin CxStates[d_{l+1}]$ **then**
15             $JOB \leftarrow (s_{d_{l+1}}, cx, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log)$;
16   send(server, JOB);
17             $CxStates[d_{l+1}] \leftarrow CxStates[d_{l+1}] \cup s_{d_{l+1}}$;
18   **else**
19   **forall** $\pi \in take(\Pi_{l+1}, s_{d_l})$ **do**
20   **if** $K, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$ **then**
21   send(server, stop);
22   **return** *buildCx()*;
23   **if** $type = cx$ **then**
24   **if** $d_{l+1} \neq \infty$ **then**
25   **forall** $\pi \in take(\Pi_{l+1}, s_{d_l})$ **do**
26           $s_{d_{l+1}} \leftarrow last(\pi)$;
27   **if** $K, \pi \not\models \Diamond\varphi_2$ *and* $s_{d_{l+1}} \notin CxStates[d_{l+1}]$ **then**
28             $JOB \leftarrow (s_{d_{l+1}}, cx, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log)$;
29   send(server, JOB);
30             $CxStates[d_{l+1}] \leftarrow CxStates[d_{l+1}] \cup s_{d_{l+1}}$;
31   **else**
32   **forall** $\pi \in take(\Pi_{l+1}, s_{d_l})$ **do**
33   **if** $K, \pi \not\models \Diamond\varphi_2$ **then**
34   send(server, stop);
35   **return** *buildCx()*;
36   send(server, getJob);

job at line 2. While the connection is open, whenever a worker receives `DATA` from the master, which must be a job, the worker deconstructs it into a 5-tuple at line 5. The code fragment at lines 6 - 22 handles the state in the 5-tuple such that the type of the job is `all`. If the state is a beginning state of a non-final layer, the worker firstly generates all states located at the next depth of the state and reachable from the state. For each generated state, together with its key, the next depth, if it does not exist in the local cache `AllStates`, the worker constructs a new job and sends it to the master; the generated state, together with its key, is added to the local cache at line 13. Secondly, the worker generates all counterexample states located at the next depth of the state and reachable from the state such that $\varphi_1 \leadsto \varphi_2$ does not hold for the sub-state space that starts from the state in the layer concerned. For each generated counterexample state, together with its key, if it does not exist in the local cache `CxStates`, the worker also constructs a new job, sends it to the master, and updates the local cache. If the next depth is unbounded (∞), meaning that the state is a beginning state of the final layer, the worker checks if there exists a path in the sub-state space that starts from the state in the final layer such that $\varphi_1 \leadsto \varphi_2$ does not hold at line 20. If that is the case, then the worker sends a stop message to the master for termination and returns a counterexample as the result of Algorithm 5.

Similarly, the code fragment at lines 23 - 35 checks for the state in the 5-tuple such that the type of the job is `cx`. However, the worker only generates all counterexample states such that $\Diamond\varphi_2$ does not hold in case of a beginning state of a non-final layer. For the final layer, the worker checks $\Diamond\varphi_2$ instead of $\varphi_1 \leadsto \varphi_2$.

The tool returns `Success` if $K, s_{d_0} \models \varphi_1 \leadsto \varphi_2$ holds. When $K, s_{d_0} \not\models \varphi_1 \leadsto \varphi_2$, the tool returns `Failure` together with a counterexample.

6.4 Outline of Parallel $L+1$-DCA2L2MC

We use a test&set mutual exclusion protocol (TAS) as an example to outline the parallel $L+1$-DCA2L2MC. TAS uses a Boolean global variable *locked* shared with all processes. *locked* is initially false. TAS uses the atomic instruction test&set that takes a Boolean variable x and atomically conducts the following: x is set to true and the old value stored in x is returned. TAS can be described in Algol-like pseudo-code as follows:

"Start Section"
ss : ...
ws : **repeat while** test&set(*locked*);
"Critical Section"
cs : *locked* := false;
"Final Section"
fs : ...

We suppose that each process wants to enter the critical section once and is located at one of the four labels ss (Start Section), ws (Waiting Section), cs (Critical Section), and fs (Final Section). Each process p is initially located at ss. We are not interested in what p does in Start

Section and then what p does at ss is to move to ws from ss. p waits at ws until test&set(*locked*) returns false. If test&set(*locked*) returns false, p goes to cs, entering Critical Section. We are not interested in what p actually does in Critical Section. p located at cs sets *locked* to false and moves to fs. We are not interested in what p does in Final Section and just suppose that p stays there.

When there are n processes participating in TAS, each state in S_{TAS} is expressed as follows:

{(locked: b) (pc[p_1] : l_1) ... (pc[p_n] : l_n) (cnt: x)}

The `locked` observable component stores the value b stored in *locked*, the pc[p_i] observable component stores the label l_i at which process p_i is located, and the `cnt` observable component stores the number x of processes that have not reached fs yet. Initially, b is `false`, each l_i is `ss` and x is n.

In this section, we suppose that there are two processes `p1` and `p2` participating in TAS. The initial state (referred to as `init`) is as follows:

```
{(locked: false) (pc[p1]: ss) (pc[p2]: ss) (cnt: 2)}
```

I_{TAS} consists of one state denoted `init`.

T_{TAS} is specified in rewrite rules as follows:

```
rl [start] : {(pc[I]: ss) OCs} => {(pc[I]: ws) OCs} .
rl [wait] : {(locked: false) (pc[I]: ws) OCs} => {(locked: true) (pc[I]: cs) OCs} .
rl [exit] : {(locked: B) (pc[I]: cs) (cnt: N) OCs}
  => {(locked: false) (pc[I]: fs) (cnt: dec(N)) OCs} .
rl [fin] : {(cnt: 0) OCs} => {(cnt: 0) OCs} .
```

The four rewrite rules are given the names `start`, `wait`, `exit`, and `fin`, respectively. `I` is a Maude variable of process ids, `B` is a Maude variable of Boolean values, `N` is a Maude variable of natural numbers, and `OCs` is a Maude variable of observable component soups. If `dec` takes a non-zero natural number $x + 1$, it returns x; if `dec` takes 0, it returns 0. Given a state expressed as `{(locked: true) (pc[p1]: cs) (pc[p2]: ss) (cnt: 2)}`, rewrite rule `exit` can change it to the following: `{(locked: false) (pc[p1]: fs) (pc[p2]: ss) (cnt: 1)}`. Fig. 6.2 shows the reachable state space made from S_{TAS}, I_{TAS}, and T_{TAS}. There are 15 states in the reachable state space.

We take two atomic propositions denoted `inWs1` and `inCs1` into account in this chapter. So, P_{TAS} consists of `inWs1` and `inCs1`. L_{TAS} is defined by the following equations:

```
eq {(pc[p1]: ws) OCs} |= inWs1 = true .
eq {(pc[p1]: cs) OCs} |= inCs1 = true .
eq {OCs} |= PROP = false [owise] .
```

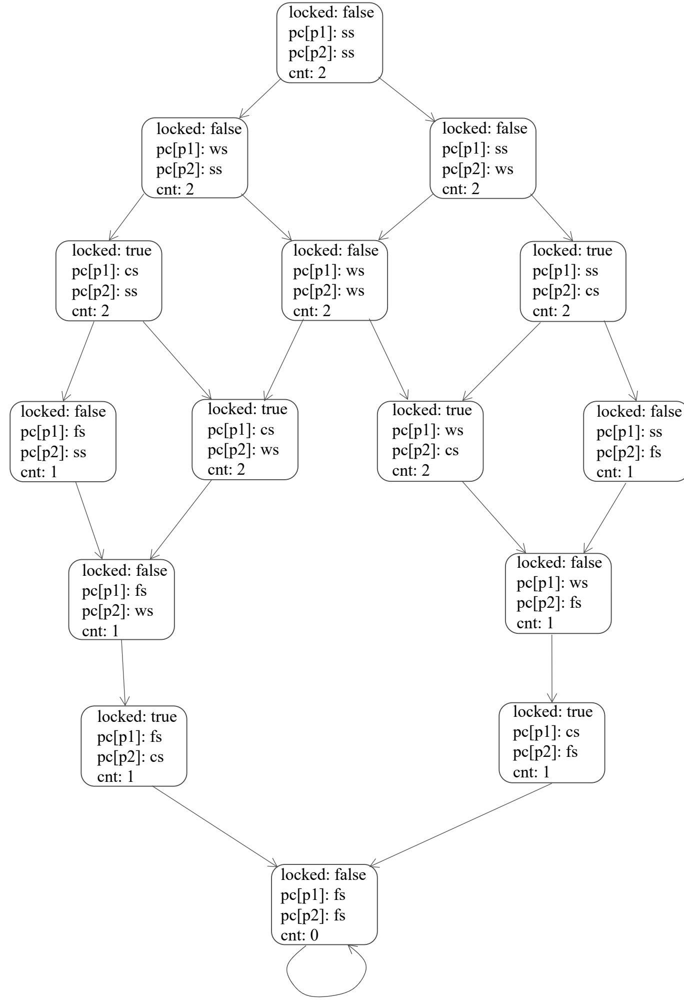


Figure 6.2: The reachable state space of TAS

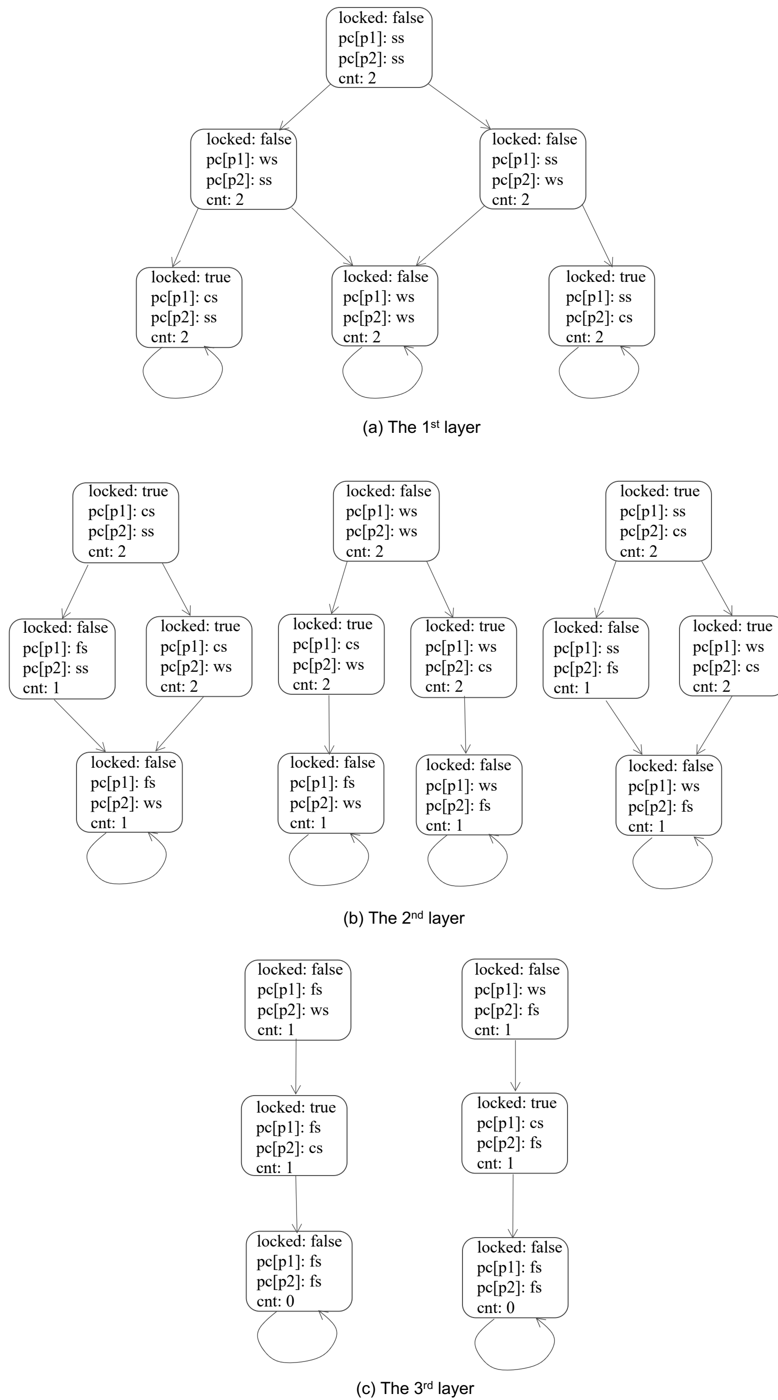
where `OCs` is a Maude variable of observable component soups and `PROP` is a Maude variable of atomic propositions. The first equation says that for all states s, $L_{\text{TAS}}(s)$ has `inWs1` if s has `(pc[p1]: ws)`; the second equation says that for all states s, $L_{\text{TAS}}(s)$ has `inCs1` if s has `(pc[p1]: cs)`; the third equation says that for all states s, $L_{\text{TAS}}(s)$ has neither `inWs1` nor `inCs1` otherwise.

We can check $K_{\text{TAS}} \models$ `inWs1` $\rightsquigarrow$ `inCs1`, namely that TAS enjoys the lockout freedom property with Maude LTL model checker by reducing the following command:

```
Maude> modelCheck(init,inWs1 |-> inCs1)
```

where `_|->_` is the Maude operator that expresses $\rightsquigarrow$. Maude LTL model checker concludes that TAS enjoys the lockout freedom property when there are two processes.

Although we do not need to use our tool to tackle the model checking experiment, we use it to outline the parallel $L+1$-DCA2L2MC. We split the reachable state space shown in Fig. 6.2 into three layers such that the first layer depth is 2 and the second layer depth is 2. Note that we do not need to specify the final layer depth (the third layer depth for the example used).



(a) The 1st layer

(b) The 2nd layer

(c) The 3rd layer

Figure 6.3: Three layers of TAS reachable state space

Fig. 6.3 (a) shows the first layer, Fig. 6.3 (b) shows the second layer, and Fig. 6.3 (c) shows the third layer. We have six sub-state spaces made from the whole reachable state space. The first layer has one sub-state space, the second layer has three sub-state spaces, and the third layer has two sub-state spaces. Among the six sub-state spaces, one sub-state space consists of six states, one sub-state space consists of five states, two sub-state spaces consist of four states, and two sub-state spaces consist of three states, while the whole reachable state space consists of 15 states. Even though it is impossible to conduct a model checking experiment for 15 states,

if it is possible to do so for six states, then $L+1$-DCA2L2MC makes it possible to conduct the model checking experiment. This is the core idea of $L+1$-DCA2L2MC.

In Algorithms 4 and 5, `AllStates`, `CxStates`, `workers`, and `jobs` are initially set to empty. Note that each state is added a `depth` observable component as `(depth: d)`, where `d` is the depth of the state, a natural number. However, the `depth` observable component is eliminated from each beginning state of the final layer right before conducting model checking for its sub-state spaces [39]. Such revisions of formal systems specifications can be automatically made by the tool and the tool can be used as an ordinary model checker in that all human users need to do is basically to feed a formal systems specification and a formal property specification. Let us use only one worker to outline the parallel $L+1$-DCA2L2MC, although in our tool the number of workers is not limited. The master starts the job scheduling by building an initial job, the 5-tuple `(init0, all, 0, 2, empty)`, referred to as `job0`, where `init0` is the initial state `init` to which the `depth` observable component is added. `job0` is enqueued to `jobs` and `init0`, together with the key 0, is added to the shared cache `AllStates` because the type of the initial job is `all`. Some values maintained by the master look as follows:

```
AllStates = 0 |-> {init0}
CxStates  = empty
workers   = empty
jobs      = job0
```

The worker starts the job handling by sending a getJob message to the master to request a job. When the master receives the getJob message, the worker identifier is enqueued to `workers`. After that, `jobs` and `workers` are not empty. The master dequeues `jobs` and `workers` to get `job0` and the worker identifier so that the master sends the job to the worker. When the worker receives `job0`, it starts handling `init0` for the first layer. What to do is to generate all states and counterexample states located at depth 2 for the leads-to property concerned. There are three states located at depth 2 as follows:

```
{(locked: true) (cnt: 2) (depth: 2) (pc[p1]: cs) (pc[p2]: ss)}
{(locked: false) (cnt: 2) (depth: 2) (pc[p1]: ws) (pc[p2]: ws)}
{(locked: true) (cnt: 2) (depth: 2) (pc[p1]: ss) (pc[p2]: cs)}
```

The three states are referred to as `init1`, `init2`, and `init3`, respectively. Among them, there is one counterexample state `init2`. The worker constructs the job `(init1, all, 2, 4, <init0 : 2>)`, referred to as `job1`, sends it to the master, and updates the local cache `AllStates` from empty to `2 -> {init1}` because `init1`, together with the key 2, does not exist in the local cache. When the master receives the job message, namely `job1`, it checks if `init1`, together with the key 2, does not exist in the shared cache `AllStates`. If that is the case, `job1` is enqueued to `jobs` and the shared cache is updated with a new entry as `2 |-> {init1}`. Similar things are done when the worker generates `init2` and `init3`. Then the values maintained by the master are updated as follows:

```
AllStates = 0 |-> {init0}, 2 |-> {init1, init2, init3}
CxStates  = empty
workers   = empty
jobs      = job1 | job2 | job3
```

where job2 and job3 are (init2, all, 2, 4, <init0 : 2>) and (init3, all, 2, 4, <init0 : 2>), respectively. Meanwhile, some values maintained by the worker are updated as follows:

```
AllStates = 2 |-> {init1, init2, init3}
CxStates  = empty
```

Once the worker finishes handling the jobs for all all-states, it handles the jobs of all cx-states. init2 is one counterexample state, together with the key 2, which does not exist in the local cache CxStates. The worker constructs the job (init2, cx, 2, 4, <init0 : 2>), referred to as job2', sends it to the master, and updates the local cache from empty to 2 |-> {init2}. When the master receives the job message, namely job2', it checks if init2, together with the key 2, does not exist in the shared cache CxStates because the type of the job is cx. If that is the case, job2' is enqueued to jobs and the shared cache is updated from empty to 2 |-> {init2}. The worker completes job0 and sends a getJob message to the master to request a job.

job1 is the top element in jobs, which is sent to the worker. When the worker receives job1, it starts handling init1 for the second layer, which is similar to what the worker has done for init0. There is only one state located at depth 4 reachable from init1 as follows:

```
{(locked: false) (cnt: 1) (depth: 4) (pc[p1]: fs) (pc[p2]: ws)}
```

The state is referred to as init4, which together with the key 4, does not exist in the local cache AllStates. The worker constructs the job (init4, all, 4, unbounded, <init1 : 4> <init0 : 2>), referred to as job4, sends it to the master, and updates the local cache with a new entry as 4 -> {init4}. When the master receives job4, it checks if init4, together with the key 4, does not exist in the shared cache AllStates. If that is the case, job4 is enqueued to jobs and the shared cache is updated with a new entry as 4 |-> {init4}. There is no counterexample state. The worker completes job1 and sends a getJob message to the master to request a job.

job2 is the top element of jobs, which is sent to the worker. When the worker receives job2, it starts handling init2 for the second layer. There are two states located at depth 4 reachable from init2, which are init4 and the other state as follows:

```
{(locked: false) (cnt: 1) (depth: 4) (pc[p1]: ws) (pc[p2]: fs)}
```

The state is referred to as init5, which is also one counterexample state. Once the worker completes job2, some values maintained by the master are updated as follows:

```
AllStates = 0 |-> {init0}, 2 |-> {init1, init2, init3}, 4 |-> {init4, init5}
CxStates  = 2 |-> {init2}, 4 |-> {init5}
jobs      = job3 | job2' | job4 | job5 | job5'
```

where job5 and job5' are (init5, all, 4, unbounded, log) and (init5, cx, 4, unbounded, log), where log is <init2 : 4> <init0 : 2>. The values maintained by the worker are also updated as follows:

```
AllStates = 2 |-> {init1, init2, init3}, 4 |-> {init4, init5}
CxStates  = 2 |-> {init2}, 4 |-> {init5}
```

The worker then sends a getJob message to the master to request a job. job3 is the top element of jobs, which is sent to the worker. When the worker receives job3, it starts handling init3 for the second layer. There is one state located at depth 4 reachable from init3, which is init5, and it is also one counterexample state. init5, together with the key 4, has existed in both local caches AllStates and CxStates, and so the worker does nothing and sends a getJob message to the master to request a job.

job2' is the top element of jobs, which is sent to the worker. When the worker receives job2', it starts handling init2 by generating all counterexample states for the eventual property concerned because the type of the job is cx and the state is a beginning state of a non-final layer. There is one counterexample state located at depth 4 reachable from init2, which is init5. However, init5, together with the key 4, has existed in the local cache CxStates, and so the worker does nothing and sends a getJob message to the master to request a job.

jobs currently has three jobs left for the final layer: job4, job5, and job5'. job4 is the top element of jobs, which is sent to the worker. When the worker receives job4, it starts handling init4 for the third layer (the final layer) by conducting model checking with the leads-to property for the sub-state space that starts from init4 because the type of the job is all. There is no counterexample. The worker completes job4 and sends a getJob message to the master to request a job. The master sends job5 to the worker. When the worker receives job5, it starts handling init5 for the third layer as what the worker has done for init4. There is also no counterexample. The worker completes job5 and sends a getJob message to the master to request a job.

Lastly, job5' is the last job left in jobs, which is sent to the worker. When the worker receives job5', the worker starts handling init5 for the final layer by model checking with the eventual property for the sub-state space that starts from init5 because the type of the job is cx. There is no counterexample. The worker completes job5' and sends a getJob message to the master to request a job. However, there is no unprocessed job left. The tool detects it for termination and returns Success at the end. Hence, we conclude that TAS enjoys the lockout freedom property when there are two processes.

Table 6.1: Experimental data with 2GB memory restriction

Protocol	#Processes	Maude LTL Model Checker	Layers	$L+1$-DCA2L2MC
Qlock	8 processes	25s	2 2	35s
	9 processes	NA		6m 57s
Anderson	7 processes	11s	2 2	16s
	8 processes	NA		2m 23s
MCS	4 processes	1s	4 4 4 4	16s
	5 processes	NA		23m 35s
TAS	11 processes	54s	3 3	1h 40m 30s
	12 processes	NA		15h 36m 11s

6.5 Experiments

We have mainly used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory to conduct experiments. Multiple experiments were performed simultaneously on the same computer so as to save time because there was only one MacPro computer available. We use Maude LTL model checker, a sequential version [125], and our parallel version of L+1-DCA2L2MC in our case studies. We use four mutual exclusion protocols: Qlock, Anderson, MCS, and TAS. Qlock is an abstract version of the Dijkstra binary semaphore. Anderson is an array-based mutual exclusion protocol invented by Anderson [130]. MCS is a list-based queuing mutual exclusion protocol (invented by Mellor-Crummey & Scott) whose variants have been used in Java virtual machines [131]. We suppose that each process enters the critical section once. For case studies, the property to be checked is the same as we used for TAS in Sect. 6.4: `inWs1` $\leadsto$ `inCs1`. Mutual exclusion protocols are still one of the core mechanisms/components used in concurrent/parallel/distributed systems [132]. Although the first mutual exclusion protocol was invented in the 60's by Dijkstra [133], research on them is still active, for example [134, 135]. As written, one of the four mutual exclusion protocols is not simply a laboratory protocol but its variants have been used in real practice. When there are a few processes participating in each of the four mutual exclusion protocols, it is possible to do a model checking experiment on an ordinary computer that carries a few, 2 GB memory, but when the number of the processes increases, it becomes infeasible to do so with such a computer because of the state space explosion. For example, when there are 8, 7, 4, and 11 processes participating in Qlock, Anderson, MCS, and TAS, respectively, both Maude LTL model checker and the sequential version of $L+1$-DCA2L2MC can complete the model checking experiments for the four protocols with a computer that carries 2 GB memory (see Table 6.1). When there are 9, 8, 5, and 12 processes in Qlock, Anderson, MCS, and TAS, respectively, however, Maude LTL model checker cannot do so with the computer, while the sequential version can do so with the computer (see Table 6.1). This is why we used the four mutual exclusion protocols for our

Table 6.2: The use of the shared and local caches

Qlock with 10 processes (8 workers - layers 2 2)			
shared caches	**local caches**	**time (1)**	**time (2)**
yes	yes	1h 14m	2h 9m
yes	no	1h 26m	2h 22m
no	yes	2h 52m	3h 42m
no	no	3h 3m	3h 45m

Anderson with 9 processes (8 workers - layers 2 2)			
shared caches	**local caches**	**time (1)**	**time (2)**
yes	yes	21m	39m
yes	no	22m	39m
no	yes	35m	1h 1m
no	no	39m	1h 1m

MCS with 6 processes (8 workers - layers 8 8)			
shared caches	**local caches**	**time (1)**	**time (2)**
yes	yes	6d 16h 46m	N/A
yes	no	5d 6h 34m	N/A
no	yes	35d 8h 7m	N/A
no	no	82d 4h 40m	N/A

TAS with 13 processes (8 workers - layers 3 3)			
shared caches	**local caches**	**time (1)**	**time (2)**
yes	yes	3d 21h 25m	N/A
yes	no	3d 8h 36m	N/A
no	yes	20d 3h 54m	N/A
no	no	28d 16h 15m	N/A

experiments.

Our tool uses the shared caches maintained by the master and the local caches maintained by each worker to prevent processed jobs from being made and assigned and reduce the communication cost between the master and workers. We have conducted some experiments to confirm the usefulness of the caches, although some experiments have not been finished due to the time limitation. The experimental data are shown in Table 6.2. Column time (1) shows the time taken when experiments were conducted on the MacPro. Column time (2) shows the time taken when experiments were conducted in a distributed environment, where the master ran on a basic AWS server that carries 1 core and 1 GB memory, while all workers ran on an iMac computer that carries 4 cores and 32 GB memory. The tool uses sockets to communicate between the master and workers. Hence, it can be deployed in a distributed environment as well as on a shared-memory machine. This is one advantage of the tool so that we can flexibly choose a shared-memory machine or a distributed environment. For MCS and TAS experiments, N/A showed in column time (2) says that the basic AWS server is so weak that we could not conduct experiments because of lack of memory, namely that the state space explosion. The experimental data say that the use of the shared and local caches is effective for Qlock and Anderson experiments, while only the use of the shared cache is more effective for MCS and TAS experiments. The effectiveness of the shared caches is larger than that of the local caches. This is likely to be because of the following: (a) duplicated states are likely to appear at shallower places from each initial state more often than at deeper places and (b) jobs including duplicated states may be made by different workers because of the dynamic nature of job distribution. Because of (b), the tendency of time (1) seems different from that of time (2). The experimental data, however, show that the use of shared caches is mandatory, while the use of local caches together is optional, which needs to be investigated furthermore with different case studies as well as configurations as one piece of our future work. To this end, we use both shared and local caches for Qlock and Anderson experiments, while we disable local caches and

Table 6.3: Experimental data for Maude LTL model checker, $L + 1$-DCA2L2MC, and Parallel $L + 1$-DCA2L2MC

Protocol	**Maude LTL Model Checker**	**Layers**	$L + 1$**-DCA2L2MC**	**Parallel** $L + 1$**-DCA2L2MC (8 workers)**
Qlock (10 processes)	37d 1h 23m	2 2	6h 31m	1h 14m
Anderson (9 processes)	12d 16h 42m	2 2	1h 44m	21m
MCS (6 processes)	25d 15h 46m	8 8	23d 19h 7m	5d 6h 34m
TAS (13 processes)	20h 18m	3 3	23d 5h 15m	3d 8h 36m

only use shared caches for MCS and TAS experiments.

We model checked `inWs1` $\leadsto$ `inCs1` for Qlock in which there are 10 process participants, for Anderson in which there are 9 process participants, for MCS in which there are 6 process participants, and for TAS in which there are 13 process participants with (1) Maude LTL model checker, (2) a sequential version [125], and (3) our parallel version of $L + 1$-DCA2L2MC. The experimental results are shown in Table 6.3. d_1 d_2 ... d_L in column *Layers* means that we use $L + 1$ layers for (2) and (3), and the depth of the lth layer is d_l. Maude LTL model checker is the default model checker powered by Maude, which did not use the tools (2) and (3) at all. For Qlock, Anderson, and MCS, the sequential version outperforms Maude LTL Model Checker. Our parallel version of $L + 1$-DCA2L2MC could speed up about 4.5 times on average to the sequential version when 8 workers are used. The speedup is understandable because there are extra costs when using our parallel version, such as socket communication between the master and workers. Although we can flexibly increase the number of workers if we would like to use more computing resources given by the computer, we only used 8 workers for the case studies reported in this work just due to the time available for the experiments. Note that for Qlock, Anderson, MCS, and TAS, Maude LTL model checker could not finish the model checking with an ordinary computer, for example, the one that carries 2 GB memory because of the state space explosion problem, while the sequential version could complete the model checking even with such a computer (see Table 6.1). For TAS, Maude LTL model checker outperforms (2) and (3). TAS is the simplest mutual exclusion protocol among the four protocols used. The difference between TAS and the other three protocols is that all processes waiting for the critical section have an equal chance to enter there next in TAS, while there exists at most one such process that can enter the critical section next in the other three protocols. In other words, TAS has a symmetry from each of the processes that wait for entering the critical section, while the other three protocols do not. Many states are then likely to be shared by many sub-state spaces in the final layer in TAS, which cannot make the best use of the shared and local caches effective. This would be probably why both (2) and (3) do not outperform (1), although (3) speeds up about 6 times to (2). (3) does not deal with all examples well but can handle three

Table 6.4: Experimental data for parallelizing the model checking in the final layer only

Protocol	**Layers**	**Parallel $L+1$-DCA2L2MC**	**Parallel $L+1$-DCA2L2MC (Only Final Layer)**	**Percentage Improvement (%)**
Qlock (10 processes)	2 2	56m 55s	42m 36s	25.15
Anderson (9 processes)	2 2	17m 30s	12m 58s	25.9
MCS (6 processes)	8 8	5d 3h 26m	22h 30m	81.77
TAS (13 processes)	3 3	2d 11h 31m	2d 5h 21m	10.36

non-trivial mutual exclusion protocols, one of which is a practical one meaning that its variants have been used in Java virtual machines. In conclusion, our parallel version can speed up the sequential version for all four protocols and outperform Maude LTL model checker for the three protocols that do not have a symmetry from each of the processes that wait for entering the critical section. It is a good common practice [136] to get rid of symmetries when designing concurrent/parallel/distributed protocols as Qlock, Anderson, and MCS that can be handled well by both sequential and parallel versions of $L+1$-DCA2L2MC.

We realize that the use of parallelization for the final layer only can gain better performance for our tool. Therefore, we support an option in our tool so that users can choose to parallelize only the model checking for the final layer while generating states up to the final layer is proceeded in sequence. The architecture of our tool is kept the same as before. Table 6.4 shows the experimental results between the parallel version in which all layers are proceeded in parallel (the third column) and another parallel version in which only the model checking in the final layer is proceeded in parallel (the fourth column). Note that both parallel versions use the new model checker in Sect. 6.6 for all counterexample generation and run on the MacPro machine above. We can see that parallelization for the final layer only can improve the running performance of our tool. Especially, it can improve 81.77% for MCS protocol.

6.6 All Counterexample Generation

Our previous work [125] (in Sect. IV with `addEqs(...)`) shows how to generate all counterexamples one by one. For each counterexample, we obtain the last state in the counterexample that repeats forever at the end (self-loop) and add an equation to the specification being checked to ignore the counterexample so that the next counterexample can be found. This way to generate all counterexamples makes our tool slower because we need to ask Maude to handle as many times as the number of counterexamples. Maude LTL model checker uses an on-the-fly LTL model checking algorithm that consists of two major steps [137]: the first for constructing a Büchi automaton, which is called the property automaton in this chapter, that represents the

negation of a correctness property and the second for lazily building the synchronous product of the system automaton and the property automaton for searching an accepting cycle reachable from an initial state.

Regarding how to construct the property automaton, there are two possible ways as follows: (1) a given property is first negated and then the negated property is converted into a Büchi automaton, and (2) a given property is first converted into a Büchi automaton and then the automaton is complemented. (1) is commonly used as a standard way [138, 139, 140, 141], while (2) is not used because negating the given property in (1) is more effective than complementing the automaton in (2). To complement a Büchi automaton, it gives rise to an exponential number of states, where the exponent is nonlinear [142], while to negate a given property, it uses a set of rules to simplify the negation of the property that is a cheap, simple, and effective way [140]. Hence, (2) requires much more memory and may take more time than (1).

Büchi automata are closed under intersections [143]. Therefore, verifying the system satisfies the correctness property becomes the emptiness problem in the product automaton. Maude LTL model checker uses a nested depth-first search algorithm to search for an accepting cycle in the product automaton [15], which terminates and returns a counterexample as soon as it is found. To generate many counterexamples at once, we need to use different algorithms so that all accepting cycles in the product automaton can be found. One approach is to detect all strongly connected components (SCCs) of the product automaton such that each SCC contains at least one accepting state. An SCC is a maximal set C of states such that for all $s_1, s_2 \in C$ there is a path from s_1 to s_2. The Tarjan algorithm [129] is well-known as an effective algorithm to find all strongly connected components in a graph and to make on-the-fly LTL verification more efficient [144]. Therefore, we develop a new model checker to generate all counterexamples based on the Tarjan algorithm by using facilities existing in Maude LTL model checker, such as building the property automaton of the negation of the correctness property. The new model checker is implemented in C++ and integrated into Maude that is publicly available in Footnote 3.

The basic idea of the Tarjan algorithm is that it uses a depth-first search (DFS) to visit each node in the graph exactly once. The algorithm maintains a stack of nodes while visiting nodes by the DFS. A node has been visited by the DFS if and only if a (recursive) call to the DFS for the node has been completed. There is an invariant property for the use of the stack: a node remains on the stack after it has been visited by the DFS if and only if there exists a path in the graph from it to some nodes earlier on the stack. In other words, it means that in the DFS a node would be only removed from the stack after all its connected paths have been traversed. Each node consists of two numbers and a Boolean value: the first number is called *index* that can be regarded as the id of the node, which is a unique number in the order in which nodes are visited by the DFS, the second number is called *lowLink* that is the lowest node id in the same SCC reachable from the node, and the Boolean value is called *onStack* that denotes whether the node is on the stack. When we first visit a node v, we assign an increment

unique id to the *index*, make (temporally) the *lowLink* become the *index*, push the node id (*index*) to the stack, and set the *onStack* to true, meaning that the state is on the stack. We then call the DFS recursively on w for each edge (v, w) if w has not been visited yet. The DFS computes the *lowLink* for the node w. Once the DFS completes for w, we update *lowLink* of v as follows: $v.lowLink \leftarrow min(v.lowLink, w.lowLink)$. If w has been visited and it is still on the stack, we update *lowLink* of v as follows: $v.lowLink \leftarrow min(v.lowLink, w.index)$. When completing the DFS for node v, we check if $v.lowLink$ is equal to $v.index$. If so, v is a base node, which has the same value for the *index* and *lowLink*, and so we pop nodes off the stack until v is popped off. Each such group of popped nodes is an SCC.

Given a system automaton S and a property automaton P, the synchronous product of S and P is built on the fly while searching SCCs at the same time. Once the product automaton is completely built, all counterexamples are also constructed and returned based on SCCs found, meaning that we use only one DFS traverse to build the product automaton and generate all counterexamples, demonstrating our practical approach. Note that every state in S is an accepting state, and therefore an accepting state in the product automaton must be an accepting state in P. For each state of the production automaton we keep the following information:

- *parentId* is the parent state id of the current state. The root state has $NONE$ whose value is -1 as its parent state id.
- *systemStateId* is the system state id in the system automaton.
- *propertyStateId* is the property state id in the property automaton.
- *stateId* is the product state id in the product automaton, which is the DFS traversal number used in the Tarjan algorithm for SCC analysis.
- *nextStates* contains all successor state ids.
- *acceptingState* is a Boolean value that denotes whether the state is an accepting state.
- *lowLink* denotes the lowest link of the state in the same SCC as in the Tarjan algorithm for SCC analysis.
- *onStack* denotes whether the state is on the stack as in the Tarjan algorithm for SCC analysis.

Algorithm 6 shows the pseudo-code for building the product automaton of system and property automata and generating all counterexamples at once. We suppose that *systemAutomaton* and *propertyAutomaton* are given as input. Each automaton can be regarded as a graph in which each node has its own unique id. We first initialize *productAutomaton*, *path*, *stateStack*, and *counterexamples* as the empty automaton, the empty list, the empty stack, and the empty

list, respectively. *productAutomaton* contains all states of the product of the system and property automata that is built while running the algorithm. *path* is used to trace the current path from the beginning state to the current state by the DFS. *stateStack* is the stack used in the Tarjan algorithm described above. *counterexamples* contains all counterexamples found by the algorithm. To generate all counterexamples, we call the *findCounterExamples* function at line 6. We first insert a dummy state that is used as the root state (referred to as variable *root*) into the product automaton at line 7 by calling the *productAutomaton.insertState* function with three parameters: the system state id, the property state id, and the parent state id of the current product state being inserted. $NONE$ is used for the three parameters, meaning a dummy id. The function will return the id of the product state just inserted in an incremental way, which starts from zero. We then get all initial state ids from the property automaton at line 8 because it may contain many initial states, not only one initial state as in the system automaton.[7] For each initial state id from the property automaton, we create a new product state at line 10 in which its system state id is $NONE$, its property state id is the initial state id, and its parent state is the root state id. We then call the *sccAnalysis* function at line 11 with three parameters: the product state id just inserted, the system state id that is $NONE$, and the initial property state id concerned. At the end of the *findCounterExamples* function, *counterexamples* are returned as all counterexamples found.

The code fragment at lines 14–33 shows how the *sccAnalysis* function works. The *sccAnalysis* function is to build the product of the system and property automata together with SCC analysis. The first parameter is the state product id, which is passed exactly once to the function marked as visited. We first get the corresponding state of the state id from *productAutomaton*, set the *lowLink* of the state to the state id, push the state id to the *stateStack*, and set the *onStack* of the state to true by calling the *pushState* function at line 15. The *pushState* function is defined in the code fragment at lines 34–48. We then push the state id to the *path* to keep track of the current path from the beginning state to the current state at line 16. Given the *propertyStateId* as a source state id, we get all possible transitions from *propertyAutomaton* at line 17. Each transition consists of two elements: the formula that makes the transition happen and the next property state id that is the target state id of the transition. Given the *systemStateId*, we get all successor state ids from *systemAutomaton* at line 19. For each transition and each successor state id, we check if the successor state satisfies the formula at line 22, where the successor state and the formula are derived from the successor state id and the transition, respectively. If that is the case, this is truly a product state that we need to

[7]Maude LTL model checker basically handles one initial state of a system automaton and so do the sequential and parallel versions of $L+1$-DCA2L2MC. Thus, we suppose that a system automaton has one initial state. It is possible to write a program in Maude so that multiple initial states of a system automaton can be handled by Maude LTL model checker and then it is also possible to revise the sequential and parallel versions of $L+1$-DCA2L2MC to do so. The algorithm for generating all counterexamples at once can also be slightly revised to handle multiple initial states of a system automaton.

Algorithm 6: Building the product automaton of the system and property automata and generating all counterexamples at once

```
   input : systemAutomaton – a system automaton
           propertyAutomaton – a property automaton
   output: all counterexamples if exists or empty list
1  productAutomaton ← empty automaton;
2  path ← empty list;
3  stateStack ← empty stack;
4  counterexamples ← empty list;
5
6  function findCounterexamples() is
7      root ← productAutomaton.insertState(NONE, NONE, NONE);
8      initialStates ← propertyAutomaton.getInitialStates();
9      forall i ∈ initialStates do
10         stateId ← productAutomaton.insertState(NONE, i, root);
11         sccAnalysis(stateId, NONE, i);
12     return counterexamples;
13
14 function sccAnalysis(stateId, systemStateId, propertyStateId) is
15     pushState(stateId);
16     path.push_back(stateId);
17     propertyTransitions ← propertyAutomaton.getTransitions(propertyStateId);
18     forall trans ∈ propertyTransitions do
19         nextSystemStates ← systemAutomaton.getNextSystemStates(systemStateId);
20         forall nextSystemStateId ∈ nextSystemStates do
21             formula ← trans.getFormula();
22             if statisfiesPropositionalFormula(nextSystemStateId, formula) then
23                 nextPropertyStateId ← trans.getStateId();
24                 nextStateId ←
                    productAutomaton.insertState(nextSystemStateId, nextPropertyStateId, stateId);

25                 if isNewState(nextStateId) then
26                     if propertyAutomaton.isAccepting(nextPropertyStateId) then
27                         productAutomaton.getState(nextStateId).acceptingState ← true;
28                     sccAnalysis(nextStateId, nextSystemStateId, nextPropertyStateId);
29                     updateLowLink(stateId, nextStateId, true);
30                 else
31                     updateLowLink(stateId, nextStateId, false);
32     generateSCC(stateId, path);
33     path.pop_back();
```

Algorithm 6: Building the product automaton of the system and property automata and generating all counterexamples at once (continuously)

```
34 function pushState(v) is
35     s ← productAutomaton.getState(v);
36     s.lowLink ← v;
37     stateStack.push(v);
38     s.onStack = true;
39
40 function updateLowLink(v, w, isNewState) is
41     s ← productAutomaton.getState(v);
42     t ← productAutomaton.getState(w);
43     if isNewState then
44         s.lowLink ← min(s.lowLink, t.lowLink)
45     else if t.onStack then
46         s.lowLink ← min(s.lowLink, t.stateId);
47
48 function generateSCC(v, path) is
49     s ← productAutomaton.getState(v);
50     if s.stateId == s.lowLink then
51         circle ← empty list;
52         hasAcceptingState ← false;
53         do
54             w ← stateStack.pop();
55             productAutomaton.getState(w).onStack ← false;
56             circle.push_front(w);
57             if productAutomaton.getState(w).acceptingState then
58                 hasAcceptingState ← true;
59         while w = / = v;
60         if hasAcceptingState then
61             isAcceptedSCC ← false;
62             if circle.size() == 1 then
63                 forall i ∈ productAutomaton.getState(v).nextStates do
64                     if v == i then
65                         isAcceptedSCC ← true;
66             else
67                 isAcceptedSCC ← true;
68             if isAcceptedSCC then
69                 counterexamples.push_back(newCounterExample(path, circle));
```

build and insert into *productAutomaton*. We then get the *nextPropertyStateId* derived from the transition at line 23 and build a product state by calling the *productAutomata.insertState* function at line 24 with three parameters: the system state id is *nextSystemStateId*, the property state id is *nextPropertyStateId*, and the parent state id is the *stateId*. The *nextStateId* is assigned the result of the function, which is the product state id just inserted. We then check if the product state id just inserted is a new state at line 25. If that is the case, then we check if the next property state, whose id is *nextPropertyStateId*, is accepting. If so, we set the *acceptingState* of the state, whose id is *nextStateId*, to true at line 27, meaning that this product state is an accepting state. We then recursively call the *sccAnalysis* function at line 28 with three parameters: *nextStateId*, *nextSystemStateId*, and *nextPropertyStateId*, meaning that we are considering the next product state in the DFS. Once the *sccAnalysis* finishes, the *lowLink* of the product state, whose id is *nextStateId*, has been calculated. We update the *lowLink* of the product state, whose state id is *stateId*, by calling the *updateLowLink* function at line 29 with three parameters: *stateId*, *nextStateId*, and true (which means that the product state is new). If the product state just inserted is not a new state, we simply call the *updateLowLink* function at line 31 with three parameters: *stateId*, *nextStateId*, and false (which means that the product state is not new).

The *updateLowLink* is defined in the code fragment at lines 40–46 with three parameters v, w, and *isNewState*. The product states s and t correspond to the product state ids v and w derived from the *productAutomaton* at lines 41 and 42, respectively, meaning that we want to update the *lowLink* of the state s based on the state t. If t is a new state, meaning that *isNewState* is true, $s.lowLink \leftarrow min(s.lowLink, t.lowLink)$. Otherwise, if t is on the stack, $s.lowlink \leftarrow min(s.lowLink, t.stateId)$. How to update the *lowLink* of a state is exactly the same as what is described in the Tarjan algorithm above. After building all product states and calculating their *lowLink* at the state, whose id is *stateId*, we call the *generateSCC* function at line 32 to search for an SCC with two parameters: the current state id *stateId* and the current path *path*. The code fragment at lines 48–69 defines the *generateSCC* function with two parameters: v and *path*. We first get the state s corresponding to the state id v derived from *productAutomaton* at line 49. We check if the *stateId* of the state is equal to the *lowLink* of the state at line 50. If that is the case, we then start building a new SCC. We pop state ids off the stack *stateStack* and store them into *circle* until v is popped off in the code fragment at lines 51–59. Whenever a state id is removed from the stack, we set the *onStack* of the corresponding state to false. While popping state ids off the stack, if there is an accepting state, we then set *hasAcceptingState* to true. We check if *hasAcceptingState* is true at line 60, meaning that the *circle* contains at least one accepting state. We need to consider two cases if the size of the *circle* is one or greater than one. In the former case, we need to check if there is a self-loop at the state in the code fragment at lines 62–65. If that is the case, the *circle* is an accepting SCC. In the latter case, it is of course an accepting SCC. At the end, if the *circle* is an accepting SCC (or equivalently *isAcceptedSCC* is true), we build a new counterexample from the *path* and *circle* at lines 68–69. For each *circle* in the counterexample, we consider

Table 6.5: Experimental data for generating all counterexamples at once in Parallel $L+1$-DCA2L2MC

Protocol	**Layers**	**Parallel $L+1$-DCA2L2MC**	**Parallel $L+1$-DCA2L2MC (New)**	**Percentage Improvement (%)**
Qlock (10 processes)	2 2	1h 14m	56m	24%
Anderson (9 processes)	2 2	21m	17m	19%
MCS (6 processes)	8 8	5d 6h 34m	5d 3h 26m	2.5 %
TAS (13 processes)	3 3	3d 8h 36m	2d 11h 31m	26 %

Table 6.6: Experimental data for only generating states up to the final layer in $L+1$-DCA2L2MC with all counterexample generation

Protocol	**Layers**	**L+1-DCA2L2MC**	**L+1-DCA2L2MC (New)**	**Percentage Improvement (%)**
Qlock (10 processes)	2 2	50.35s	2.26s	95.5%
Anderson (9 processes)	2 2	20.99s	2.26s	89.2%
MCS (6 processes)	8 8	17h 11m	65m	93.7 %
TAS (13 processes)	3 3	25m 11s	30.64s	97.95 %

only one *path* that leads to the *circle*, although there may be multiple paths leading to the *circle* in the product automaton. Our parallel tool only requires obtaining all accepting circles in the product automaton, and therefore our algorithm is tailored for that purpose. Once the *generateSCC* function completes, we update the *path* by removing the current state id from the list at line 33.

We have described how to build the product of system and property automata and generate all counterexamples at once. We have developed it as a new model checker and integrated it into Maude. For users to use our new model checker, we make a `modelCheckAll` function that has the same parameters as the built-in `modelCheck` function in Maude. For TAS case study, we can run the following command to generate all counterexamples at once:

```
Maude> modelCheckAll(init,inWs1 |-> inCs1)
```

We conduct some experiments to demonstrate the power of generating all counterexamples at once. The MacPro computer above is used to conduct the experiments. The experimental data in Table 6.5 show the verification time with the old parallel $L+1$-DCA2LMC (the third column), which uses the default model checker, and the new parallel $L+1$-DCA2L2MC (the fourth column), which uses the new model checker. The fifth column shows the percentage of improvement of the new version to the old version. We can improve the verification time

of Qlock, Anderson, and TAS protocols by around 20%, while the improvement is only 2.5% for MCS protocol. We would like to demonstrate how all counterexample generation improves state generation in our tools effectively. Therefore, we conduct some more experiments for the sequential version of $L+1$-DCA2LMC with and without the use of all counterexample generation to generate states up to the final layer, but we do not conduct model checking experiments for sub-state spaces in the final layer because we only consider state generation in these experiments. The experimental data are shown in Table 6.6. The third and fourth columns denote the time taken for generating states up to the same final layer of the old and new versions, respectively. We can see that the percentage of improvement is over 89% for all case studies. We can reduce the time taken to generate states up to the final layer of MCS and TAS protocols from 17h 11m and 25m 11s to 65m and 30.64s, respectively. This result is good for us to speed up how to find out a good layer configuration for $L+1$-DCA2LMC technique because generating states at each layer is crucial, which will be described in the next section.

In summary, we have described a technique to generate all counterexamples at once and developed a new model checker based on the Tarjan algorithm by using facilities existing in Maude LTL model checker. The new model checker is implemented in C++ and integrated into Maude from which users can use the new model checker by the `modelCheckAll` function. Our experimental data have shown the power of our technique for all counterexample generation from which the running performance of our tool can be improved to some extent.

6.7 Layer Configuration Selection

We can learn from the experimental data in Sect. 6.5 that layer configuration affects the running performance of our tool. In this section, we describe an approach to finding good layer configurations with an analysis tool. We have some observations when conducting experiments with our tool as follows:

- We can divide the reachable state space into as many layers as possible with a different depth for each layer. However, we should use a small depth in a specific range (e.g., 1–4) for each layer because if so we are more likely to avoid the state space explosion while generating states and keep the product automaton reasonably small so as to take advantage of our new model checker to generate all counterexamples at once. In addition, a large layer depth can be constructed by combining multiple small layer depths. We would like to make each sub-state space small enough as well. Thus, we may rely on the numbers of all-states and cx-states reachable from an initial state at a given depth to decide the depth of each layer that should be used. In the specific range, we use the largest layer depth such that the numbers of all-states and cx-states are smaller than or equal to specific numbers n_1 and n_2, respectively. In our case studies, we use 500 and 250 for n_1 and n_2, respectively. For example, for MCS with 6 process participants, there are

126 all-states and 0 state located at the depth 4 reachable from an initial state, and so we use at most 4 as the largest depth of each layer. States tend to drastically increase when positions from each initial state get deeper. We may need to use a smaller number as the depth of each layer when the layer is located at a deeper position. For example, we use 2 as the depth of each deep layer for MCS subsequently.

- We should divide the reachable state space into a reasonable number of layers because if the depth of states located at the final layer is shallow, there are some sub-state spaces in the final layer whose size is still big, which may lead to the state space explosion and degrade the verification time. Meanwhile, if the depth is too deep, the number of the beginning states of the final layer is very large because the number of states dramatically increases after each layer while the time taken to generate states up to the final layer also dramatically increases when many states need to be considered for state generation after each layer. In addition, we realize that the use of parallelization for model checking states located at the final layer only obtains a better performance with our tool, which was described in detail in Sect. 6.5. Therefore, if the time taken to generate states is very large, we cannot take advantage of the parallelization, while if the time taken to model checking all-states and cx-states at the final layer is very large, we can take advantage of the parallelization to reduce the verification time. Therefore, we may stop using one more layer when the time taken to generate the beginning states of the final layer becomes very large.

Based on the observations, our approach to finding a good layer configuration is as follows. We start with a two-layer configuration d_1 such that d_1 is small in a specific range (e.g., 1–4). Note that we analyze the numbers of all-states and cx-states reachable from an initial state at each layer depth in the range to decide d_1. We generate all cx-states and all-states in layer 1 and measure the time taken to do so. We would like to know how much time it takes to conduct model checking experiments for the final layer (layer 2 this time). Even in layer 1 whose depth is small enough, however, there may be many cx-states and all-states. Thus, it is not reasonable to tackle all of many cx-states and all-states. Hence, we randomly select some cx-states and some all-states to conduct model checking for the final layer and roughly estimate the time taken to conduct model checking experiments for the final layer. We should increase the coverage of cx-states and all-states for better estimation. If we tackle a large number of cx-states and all-states in sequence, it may take an unreasonable amount of time. We then take advantage of parallelization by running multiple instances of a model checker. Each instance is in charge of a small number of cx-states and all-states. If multiple instances are used and each instance selects some cx-states and all-states randomly and independently, it is possible to increase the coverage of cx-states and all-states in a reasonable amount of time. Depending on the power of a machine used, we can decide the number of instances that can run in parallel. The more states are selected and the more instances are used, the more confident we are in the

estimated verification time.

If the estimated verification time for the final layer (layer 2 this time) is too large, we add one more layer with a small depth d_2. The next layer configuration is d_1 d_2. Let us suppose that we keep the cx-states and all-states in layer 1. We then generate all cx-states and all-states in layer 2 and measure the time taken to do so. We estimate the verification time for the final layer (layer 3 this time). If the estimated verification time is not too large, the current layer configuration d_1 d_2 is a good layer configuration candidate. If the time taken to generate cx-states and all-states in the first two layers is not small enough, we stop the layer configuration selection and select d_1 d_2 as a good layer configuration. Otherwise, we can keep on finding a better layer configuration by adding one more layer with a small depth d_3, when the next layer configuration is d_1 d_2 d_3. We generate all cx-states and all-states in layer 3 and measure the time taken to do so. We estimate the verification time for the final layer (layer 4 this time). We compare the (estimated) total verification time (the time taken to generate cx-states and all-states plus the estimated verification time for the final layer) for d_1 d_2 and the total verification time for d_1 d_2 d_3. If the former is smaller than the latter, we stop the layer configuration selection and select d_1 d_2 as a good layer configuration. Otherwise, we add one more layer with a small depth and keep on doing what has been described.

We have developed an analysis tool to support the approach to layer configuration selection described above to find a good layer configuration. We use a server as the central place to handle commands that are sent by a commander. The tool is developed in Maude and uses sockets to communicate between the server and the commander. For the server, we use two new features in Maude: timers and meta-interpreters. Meta-interpreters allow us to run a separate process to handle any jobs, and so we use them to run many model checking instances in parallel. Timers allow us to set up a timeout and listen to the timeout event from which we can analyze the current progress of each meta-interpreter instance on the fly. Maude does not support a convenient way to get the current time in their built-in functions. Therefore, we develop the `getTimeSinceEpochAt` function to get the time elapsed in nanoseconds since the Unix Epoch, which is implemented in C++ and integrated into Maude, which is also publicly available in Footnote 3. The implementation of our analysis tool is publicly available in Footnote 4, which resides under the *analysis* folder.

Our analysis tool supports the following commands that are sent by the commander to the server:

- `seed_` requests to change the seed used to randomly select states at each layer. The default seed is set to 37. State selection is subject to the pseudorandom number generator, whose initial value is the seed.

- `threshold_` requests to change the threshold, the time in which we would like to complete our verification. It is set to unbounded as default. When it is not unbounded, each time model checking a state completes or a timeout event happens, we check the estimated

verification time at that moment. If it exceeds the threshold, we stop all running instances. Otherwise, the instances keep on doing until they accomplish.

- `timeout_` requests to change the timeout time for our analysis tool. It is set to 1 minute by default.
- `maxStates(_,_)` requests to change the numbers of all-states and cx-states reachable from an initial state located at a layer depth, namely n_1 and n_2, respectively, that are used to decide the depth of each layer. n_1 and n_2 are set to 500 and 250 as default, respectively.
- `analyzeDepth` requests the server to analyze the numbers of all-states and cx-states reachable from an initial state at each layer depth in a specific range (e.g., 1–4) and recommend a depth of each layer such that the depth is the largest layer depth in the specific range and the numbers of all-states and cx-states reachable from the initial state at the depth are smaller than n_1 and n_2, respectively. If there is no such depth in the range, 1 is used as the depth.
- `layerCheck_` requests the server to use one more layer with the layer depth information given by the parameter in the command. The very initial use of the command, say `layerCheck` d_1, makes the layer configuration d_1, which means that two layers are used and the first layer depth is d_1.
- `select(_,_)` requests to select some all-states and cx-states at each layer instead of considering all states. We use this command if we would like to quickly reach some states located at the final layer and conduct some model checking experiments in the final layer to know how much time needs to be approximately spent to model check a state on average at the final layer.
- `show` requests to show the current status of the server, such as the depth to the current final layer, the layer configuration with its time taken to generate states accordingly, the seed, and the threshold.
- `analyze(_,_,_)` requests to run some trial model checking experiments in the current final layer. The first parameter is the number of instances we want to run, and the second and third parameters are the number of all-states and cx-states that are randomly selected from the all-states and cx-states located at the current final layer, respectively.
- `kill` requests to kill all running instances in case we want to do so, for example, if the time taken to model check a state at the current final layer is so long that we cannot wait.
- `close` requests to close the connection between the server and the commander. The commander will stop, while the server still works so that another commander can communicate.

- `stop` requests to stop both the server and the commander.

The _ symbol is used as a placeholder for a parameter in each command above. Let us demonstrate how to use the analysis tool to find a good layer configuration for Qlock in which 10 processes participate. Note that we use the default seed, the default threshold, the default numbers of all-states and cx-states (n_1 and n_2), and the default timeout in our analysis. We first need to start our server with the following command:

```
Maude> in analyzer.maude
```

Note that the server needs to pre-load many modules and configurations as in the server in our parallel tool and more to be able to handle the commands sent by the commander. We then start the commander to connect to the server by the following command:

```
Maude> in commander.maude
```

We start with two layers for Qlock such that the first layer depth is small in the range 1–4. To decide the first layer depth candidate (or the depth of each layer), we send the `analyzeDepth` command from the commander to the server. Once the server completes the depth analysis, it outputs the following:

```
[AnalyzeDepth]
  [Layer depth 1] #all-states = 10, #cx-states = 1, time = "17210000ns
    OR 0d:0h:0m:0s"
  [Layer depth 2] #all-states = 100, #cx-states = 18, time = "32654000ns
    OR 0d:0h:0m:0s"
  [Layer depth 3] #all-states = 820, #cx-states = 225, time = "108791000ns
    OR 0d:0h:0m:0s"
  [Layer depth 4] #all-states = 5850, #cx-states = 2169, time = "783230000ns
    OR 0d:0h:0m:0s"
 -----------------------
 Layer depth recommendation: 2
```

Based on the result from the server, it recommends using 2 as the first layer depth because the numbers of all-states and cx-states are 100 and 18 at the bottom of the first layer, respectively. We then send the `layerCheck 2` command from the commander to the server. Once the server completes generating states up to depth 2, it outputs the following:

```
[HandleLayerCheck]
  Depth = 2
  Depth List = 2
  #all-states = 100
  #cx-states = 18
  Time to generate states at this layer = "26018000ns OR 0d:0h:0m:0s"
  Time to generate states in total = "26018000ns OR 0d:0h:0m:0s"
```

The numbers of all-states and cx-states are 100 and 18, respectively. The time to generate states at this layer is so small, namely 26018000ns, that it shows 0 seconds. We next send the `analyze(10, 5, 2)` command from the commander to the server. The server then starts to prepare 10 meta-interpreter instances in which the original module, which specifies the behavior of Qlock protocol, is loaded. We randomly select 5 all-states and 2 cx-states from 100 all-states and 18 cx-states for each meta-interpreter instance, respectively. In total, we check about 50 all-states and 20 cx-states by random. Note that each meta-interpreter instance conducts model checking experiments for the sub-state spaces from its randomly selected 7 states in the final layer, namely layer 2, independently, and a same state may be handled by multiple instances. We then request the meta-interpreter instances to conduct such model checking experiments. Because the current depth is shallow, namely 2, we find it uncompleted to conduct the model checking experiments for almost all the sub-state spaces in several minutes. We can see the current progress of each instance whenever a timeout event is raised. The following information is the progress of the meta-interpreter instance whose id is 2.

```
[interpreter(2)] status = (working).StatusRun
  All-States = 0/5, timeAllStates = (nil).NatList
  Cx-States = 2/2, timeCxStates = 2025625000 4820393000
  Being checking a state, isCxState = false, taking time = "713338363000ns
    OR 0d:0h:11m:53s"
  -----------------------
  Average time to check an all-state = "713338363000ns OR 0d:0h:11m:53s"
  Average time to check a cx-state = "3423009000ns OR 0d:0h:0m:3s"
  Total time to check all-states = "71333836300000ns OR 0d:19h:48m:53s"
  Total time to check cx-states = "61614162000ns OR 0d:0h:1m:1s"
  -----------------------
  Total time to check all states = "71395450462000ns OR 0d:19h:49m:55s"
  Total time to generate states = "26018000ns OR 0d:0h:0m:0s"
  Total time = "71395476480000ns OR 0d:19h:49m:55s"
```

For each meta-interpreter instance, we show the number of completed states per the number of selected states, the time taken (in nanoseconds) for completion of model checking the all-states and cx-states selected at the moment in a form of natural number lists, the current state being checked, and its elapsed time so far. From the results, we estimate the average time taken to model check each of one all-state and one cx-state by summing the natural numbers in the corresponding natural number list and the elapsed time of the current state being checked if any and dividing the sum by the number of the corresponding states. Based on the numbers of all-states and cx-states at the current final layer, we calculate the total time taken to model check all-states (1) and cx-states (2) by simply multiplying the average time taken for one all-state and one cx-state with the numbers of all-states and cx-states, respectively. The total time taken to model check all states in the final layer is the sum of (1) and (2). We have the

time taken to generate all states including those at the final layer and the time taken to model check all states at the final layer, we sum them up to estimate the verification time in total. We decide to quit the current attempt because the time elapsed so far to model check the state being tacked by the meta-interpreter instance whose id is 2 is about 11 minutes and the time taken to complete it would be (much) longer. To stop the running instances, we send the `kill` command from the commander to the server. The server stops all running instances and waits for the next command. We then use one more layer with depth 2 by sending the `layerCheck 2` command from the commander to the server. Once the server finishes, it outputs the following:

```
[HandleLayerCheck]
  Depth = 4
  Depth List = 2 2
  #all-states = 5850
  #cx-states = 2169
  Time to generate states at this layer = "910727000ns OR 0d:0h:0m:0s"
  Time to generate states in total = "936745000ns OR 0d:0h:0m:0s"
```

We can see that the time taken to generate states is still very small, while the number of the states located at layer 2 increases dramatically because there are 100 non-counterexample states and 18 counterexample states located at layer 1 instead of only one (non-counterexample) initial state located at layer 0. We next send the `analyze(10, 10, 10)` command from the commander to the server. The server then prepares 10 meta-interpreter instances and randomly selects 10 all-states and 10 cx-states to be model checked by each meta-interpreter instance. This time we increase the number of selected states because we want to cover more states in this layer. In this attempt, each meta-interpreter instance has completed its 20 model checking experiments in a short time. There may be multiple possible ways to use the results obtained from multiple meta-interpreter instances. One possible way to do so is to calculate the average estimated time taken to model check all-states and cx-states. Another possible way to do so is to consider the worst (largest) estimated time among multiple results. We take the latter option in this work because this option treats the estimated time carefully. The output of one meta-interpreter instance whose estimated verification time in total is the largest is as follows:

```
[interpreter(9)] status = completed
  All-States = 10/10, timeAllStates = 618985000 670995000 630488000 598721000
    649327000 597325000 492742000 5592072000 6228200000 2681010000
  Cx-States = 10/10, timeCxStates = 298284000 201136000 232159000 242414000
    239203000 193797000 232721000 277390000 108865000 8953974000
  -----------------------
  Average time to check an all-state = "1875986500ns OR 0d:0h:0m:1s"
  Average time to check a cx-state = "1097994300ns OR 0d:0h:0m:1s"
  Total time to check all-states = "10974521025000ns OR 0d:3h:2m:54s"
```

```
  Total time to check cx-states = "2381549636700ns OR 0d:0h:39m:41s"
  -----------------------
  Total time to check all states = "13356070661700ns OR 0d:3h:42m:36s"
  Total time to generate states = "936745000ns OR 0d:0h:0m:0s"
  Total time = "13357007406700ns OR 0d:3h:42m:37s"
```

We can see that the time taken to model check each state is very small and each model checking experiment has been completed quickly. Therefore, we can take this layer configuration into account as one candidate. Because the time taken to model check each state is about 1 second, we may not need to use one more layer. However, we suppose that there may be a better layer configuration. We then use one more layer with depth 2 by sending the `layerCheck 2` command from the commander to the server. Once the server finishes, it outputs the following:

```
[HandleLayerCheck]
  Depth = 6
  Depth List = 2 2 2
  #all-states = 187245
  #cx-states = 104400
  Time to generate states at this layer = "1053136606000ns OR 0d:0h:17m:33s"
  Time to generate states in total = "1054073351000ns OR 0d:0h:17m:34s"
```

We can see that the time taken to generate states and the number of states at this layer increase dramatically. We next send the `analyze(10, 10, 10)` command from the commander to the server. Each model checking experiment has been completed quickly and the output of one meta-interpreter instance whose estimated verification time is the largest is as follows:

```
[interpreter(9)] status = completed
  All-States = 10/10, timeAllStates = 1870298000 18879451000 817523000 820570000
    823072000 823560000 823654000 812557000 814786000 813653000
  Cx-States = 10/10, timeCxStates = 813367000 815817000 811394000 812360000
    823490000 818020000 812860000 817541000 826695000 2886698000
  -----------------------
  Average time to check an all-state = "2729912400ns OR 0d:0h:0m:2s"
  Average time to check a cx-state = "1023824200ns OR 0d:0h:0m:1s"
  Total time to check all-states = "511162447338000ns OR 5d:21h:59m:22s"
  Total time to check cx-states = "106887246480000ns OR 1d:5h:41m:27s"
  -----------------------
  Total time to check all states = "618049693818000ns OR 7d:3h:40m:49s"
  Total time to generate states = "1054073351000ns OR 0d:0h:17m:34s"
  Total time = "619103767169000ns OR 7d:3h:58m:23s"
```

The estimated verification time in total is not better than the previous layer configuration. Therefore, we stop here and do not need to use any more layers. This layer configuration

Table 6.7: Parallel $L+1$-DCA2L2MC with different layer configurations

Protocol	**Layers**	**Parallel $L+1$-DCA2L2MC (8 workers)**
Qlock (10 processes)	2 2	42m 36s
	2 2 2	2d 11h 31m
Anderson (9 processes)	2 2	12m 58s
	2 2 2	1h 36m 13s
MCS (6 processes)	4 4 4 4 2	22h 13m 45s
	4 4 4 4 2 2	2d 12h 47m
TAS (13 processes)	3 3 3	1d 12h 33m
	3 3 3 3	1d 13h 53m

Table 6.8: Analysis time to find good layer configurations

Protocol	**Layers**	**Time to generate states**	**Time to model check states**	**Analysis time**
Qlock (10 processes)	2 2	50s	13m	13m 50s
Anderson (9 processes)	2 2	21s	8m	8m 21s
MCS (6 processes)	4 4 4 4 2	8h 1m 15s	1h 20m	9h 21m 15s
TAS (13 processes)	3 3 3	38m 26s	30m	1h 8m 26s

should not be used. So far we have only one layer configuration candidate that is 2 2. However, we conduct experiments for Qlock protocol with two layer configurations 2 2 and 2 2 2 to demonstrate the usefulness of our approach. We use the MacPro machine above to conduct the experiments. The experimental data are shown in Table 6.7. We can see that the layer configuration 2 2 is better than 2 2 2 for Qlock protocol, which demonstrates the usefulness of our approach to finding a good layer configuration for our tool. We conduct the same layer configuration analyses for Anderson, MCS, and TAS protocols. Their good layer configurations found are 2 2, 4 4 4 4 2, and 3 3 3, respectively, while the layer configurations that make their verification time degrade are 2 2 2, 4 4 4 4 2 2, and 3 3 3 3, respectively. We conduct both two layer configurations for each case study to demonstrate the usefulness of our approach. The experimental data are shown in Table 6.7. We can see that the layer configurations 2 2, 4 4 4 4 2, and 3 3 3 are better than 2 2 2, 4 4 4 4 2, and 3 3 3 3 in verification time for Anderson, MCS, and TAS protocols, respectively. Using our approach, we can find the better layer configurations for Qlock and Anderson protocols that are the same as the layer configurations used before in Table 6.3, while we can find better layer configurations for MCS and TAS protocols than the layer configurations used before in Table 6.3. Especially, we can reduce about 16 hours in verification time from 2 days 5 hours 21 minutes to 1 day 12 hours 31 minutes with the new layer configuration for TAS protocol.

Table 6.8 shows the time we spent selecting good layer configurations for the protocols used

with our approach. The last three columns denote the time taken to generate states, the time taken to model check states selected by meta-interpreter instances, and the analysis time in total, respectively. The analysis time in total mainly depends on the time taken to generate states at each layer. Because at each current final layer concerned, we only randomly select some states for each meta-interpreter instance to conduct model checking experiments that do not take much time. For Qlock and Anderson protocols, the time to generate states is small, and therefore the total analysis time is also small. For MCS and TAS protocols, the time to generate states is large, and therefore the total analysis time is also large. However, including the analysis time, the total time taken to conduct model checking experiments with our tools for Qlock, Anderson, and MCS protocols is still much better than Maude LTL model checker. For TAS protocol, even with the good layer configuration found, our tool cannot outperform Maude LTL model checker as described above. For TAS and MCS protocols, we need to spend more time to model check states selected at layers 4 4 4 4 and 3 3, respectively, because these layer configurations also can be regarded as our candidates. However, we can find better layer configurations for them subsequently.

In summary, we have proposed an approach to finding good layer configurations with an analysis tool that starts with two layers such that the first layer depth is small in a specific range (e.g., 1–4), and then gradually stacks up layer by layer with a small depth. The small depth can be recommended by our analysis tool based on the numbers of all-states and cx-states reachable from an initial state at each layer depth in the specific range. At each current final layer, we randomly select some all-states and cx-states to conduct model checking with multiple meta-interpreter instances that run in parallel. For each meta-interpreter instance, we estimate the verification time in total and the average time taken to model check a cx-state and an all-state at the current final layer from which we can decide whether we should use one more layer or not. Based on the output, we may select a good layer configuration. Note that we estimate roughly the verification time based on the number of randomly selected states and the number of meta-interpreter instances used. If the machine used in the analysis is powerful, we can increase the number of instances so that even with the small number of states selected for each instance, the coverage is increased. In conclusion, we should consider the number of states selected, the number of meta-interpreter instances used, and the depth used for each layer, to find a good layer configuration. Besides, we should use a different seed number to select different states for each analysis. Some experiments were conducted to demonstrate the usefulness of our analysis tool as well as our approach for layer configuration selection.

6.8 Scalability Testing

Scalability testing is used to measure how well an application performs with various problem sizes and numbers of processors in terms of strong scaling and weak scaling.

Table 6.9: Strong scaling data for our tool with TAS

Protocol	**Maude LTL Model Checker**	**Layers**	$L+1$-**DCA2L2MC**	**#workers**	**Parallel** $L+1$-**DCA2L2MC**
TAS (13 processes)	20h 18m	3 3 3	6d 13h 44m	8	3d 8h 36m
				16	15h 30m
				16	12h 2m
				32	10h 11m

- **Strong scaling**: the number of processors is increased, while the problem size does not change, making the workload for each processor reduce. An application scales perfectly strongly if the time to complete the problem is reduced in proportion to the number of processors increased.
- **Weak scaling**: both the number of processors and the problem size are increased such that the workload for each processor does not change. An application scales perfectly weakly if the time to complete the problem does not change when both the number of processors and the problem size are increased.

In practice, both strong scaling and weak scaling cannot be achieved perfectly because the communication overhead for algorithms in parallelization is often proportional to the number of processors used. In this section, we measure strong scaling for our tool with TAS case study only. Meanwhile, we ignore weak scaling at this moment because of the followings: it is non-trivial to calculate the problem size of TAS properly from which we can increase the problem size as well as the corresponding number of workers (processors) in our tool; even if we can calculate the problem size properly, the MacPro does not have enough the number of cores to make the workload for each processor unchanged.

For TAS case study, when eight workers were used, our tool could not outperform Maude LTL model checker. Therefore, we would like to use the good layer configuration for TAS found by our analysis tool in the previous section and to increase the number of workers to check whether our tool can outperform Maude LTL model checker. Besides, we measure strong scaling for our tool with TAS. The experimental data is shown in Table 6.9. Because the MacPro has only 28 cores, we use only up to 32 workers. When the number of workers is 16, 24, and 32, our tool can outperform Maude LTL model checker. The strong scaling speedup achieved by increasing the number of workers is plotted in Fig. 6.4. The ideal speedup is the straight line with rectangle marks, while the real speedup is the other line with square marks. The speedup is calculated based on the ratio of the time taken for the parallel version and the sequential version. Note that we use our new model checker integrated into our tool and conduct model

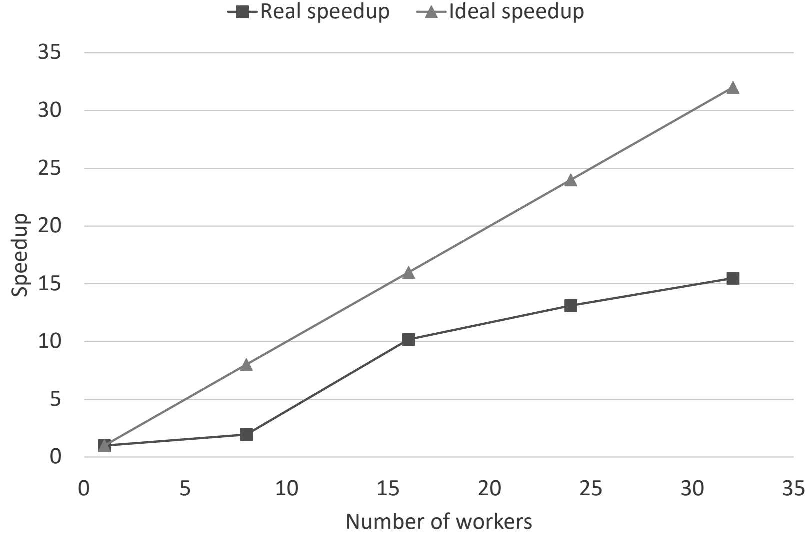


Figure 6.4: Strong scaling speedup for TAS

checking at the final layer in parallel only in these experiments. The goal of strong scaling is to find a sweet spot from which the time to complete the problem degrades subsequently. Because of the limitation of our MacPro, we could not make a test with a large number of workers. However, the speedup by increasing the number of workers tends to go up more or less continuously and reach the sweet spot at the end. After that, if we keep on increasing the number of workers, the speedup will go down because the communication overhead is high while the workload for each worker becomes small. Model checking at the final layer in our technique/tool is completely independent and the number of states and counterexample states at the final layer is large. Hence, we conjecture that the sweet spot for TAS may require thousands of workers. One piece of our future work is to use a high-performance system to conduct experiments with our tool in which thousands of workers can be used.

6.9 Limitations

It is necessary to make each sub-state space, especially in the final layer, much smaller than the original reachable state space of a system under model checking so as to use $L+1$-DCA2L2MC and its parallel version effectively. If a system under model checking has some long lasso loops, some sub-state spaces in the final layer may not be small enough. Thus, we should avoid any long lasso loops in a system under model checking. In general, it would not be straightforward to get rid of all long lasso loops from a system under model checking. Thus, we need to come up with a technique that can handle such long loops. One possible approach to it is as follows: each long lasso loop is divided into multiple short finite sequences of states, model checking experiments for these finite sequences are conducted and their model checking results are combined to conclude the model checking experiment for the long lasso loop.

There are some systems, such as TAS, such that many states are shared by many sub-state spaces in the final layer. If so, it is a non-trivial burden to tackle the final layer because it is necessary to visit the same states multiple times by multiple workers. Neither L+1-DCA2L2MC nor its parallel version can deal with such systems well.

Layer configuration selection needs efforts even though the analysis tool can make such efforts moderate. It would be almost impossible to conduct layer configuration selection if we do not have such an analysis tool. Thus, our analysis tool is very useful, which has been demonstrated by describing how to find a good layer configuration for Qlock. However, it would be preferable to make layer configuration selection fully automatic. We have almost all technical components to this end, among which the analysis tool is the most crucial. We need to conduct more case studies to find good parameter values, such as threshold and timeout, although we should make it possible for human users to set those parameters. This is one possible piece of our future work.

6.10 Summary

We have described a parallel version of $L + 1$-DCA2L2MC and a tool that supports it. The tool has been implemented in Maude. We have also reported on some case studies in which it is demonstrated that the tool can increase the running performance of model checking. We have also described a technique to generate all counterexamples at once and developed a new model checker to support the technique. Some experiments were conducted to demonstrate the power of the technique that can improve the running performance of our tool. Furthermore, layer configuration selection was addressed with an approach to finding a good layer configuration for $L + 1$-DCA2L2MC technique and an analysis tool that supports the approach. Some experiments were conducted to demonstrate the usefulness of the analysis tool as well as the approach for layer configuration selection. Besides, our research group has proved a theorem to guarantee the correctness of $L + 1$-DCA2L2MC [39]. Therefore, the correctness of our parallel version of $L + 1$-DCA2L2MC is also preserved.

Although the technique presents a good performance and improves the sequential version [125], we consider several lines of future work. We should conduct more case studies with the tool. We have used several mutual exclusion protocols but need to use different kinds of protocols, such as communication protocols.

Although we did not use any fairness assumptions in the case studies, it is often necessary to use some fairness assumptions to model check liveness properties, such as leads-to ones. An extension of Maude LTL model checker, Maude LTLRuLF model checker [145, 146], has been developed, which makes it possible to conduct model checking experiments under fairness, where LTLRuLF stands for linear temporal logic of rewriting under localized fairness. We could use Maude LTLRuLF model checker instead of Maude LTL model checker.

Chapter 7

Parallel Maude-NPA for Cryptographic Protocol Analysis

Maude-NPA is a formal verification tool for analyzing cryptographic protocols in the Dolev-Yao strand space model modulo an equational theory defining cryptographic primitives. It starts from an attack state to find counterexamples or conclude that the attack concerned cannot be conducted by performing a backward narrowing reachability analysis. Although Maude-NPA is a powerful analyzer, its running performance can be improved by taking advantage of parallel and/or distributed computing when dealing with non-trivial protocols whose state space is huge. This chapter describes a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption at each layer in Maude-NPA are conducted in parallel. A tool supporting the parallel version has been implemented in Maude with a master-worker model. We report on some experiments of various kinds of protocols that demonstrate that the tool can increase the running performance of Maude-NPA by 44% on average for all non-trivial case studies experimented in which the number of states located at each layer is considerably large.

7.1 Introduction

Maude-NPA uses a breadth-first search (BFS) to explore the reachable state space. Given a set of states in layer l (or depth l from an attack state, where the attack state is located at layer 0), for each state in the set, Maude-NPA performs the backward narrowing just by one step to obtain its successor states in layer $l + 1$, which is referred to as step (1). The backward narrowing for each given state from the set of states can be conducted independently, which opens an opportunity for parallelization so as to improve the running performance of Maude-NPA. In addition, as soon as the successor states at each layer are obtained from step (1), Maude-NPA conducts the transition subsumption, which is referred to as step (2). The transition subsumption can be regarded as the partial order reduction for narrowing-based state exploration, to remove states that are implied by either other states in the successor states or

visited states (history states) from the set of successor states.

We have parallelized step (1) in our previous work [147]. In this chapter, we describe how to parallelize steps (1) and (2) as well to improve the running performance of Maude-NPA furthermore. Our parallel version of Maude-NPA is called Par-Maude-NPA. To distinguish Par-Maude-NPA from the version described in [147], it may be called Par-Maude-NPA-2, while the version described in [147] may be called Par-Maude-NPA-1. Maude-NPA is implemented in Maude [13] and so are Par-Maude-NPA-1 and Par-Maude-NPA-2, where Maude is one direct successor language of OBJ3 [126], an algebraic specification language, and based on rewriting logic [127] as its theoretical foundation. Both Par-Maude-NPA-1 and Par-Maude-NPA-2 use a master-worker model. Par-Maude-NPA-1 uses Maude sockets to transmit data between the master and a worker. Maude sockets work well for Par-Maude-NPA-1 because only small data are transmitted between the master and a worker in step (1). It is necessary to transmit large data between the master and a worker in step (2). Thus, Par-Maude-NPA-2 uses Maude meta-interpreters, a new feature of Maude, instead of Maude sockets.

Meta-interpreters can be run in a separate process to handle jobs independently and processes can communicate with each other by using Unix domain sockets, which create filesystem objects to communicate between processes on the same host with no IP address required. Meanwhile, Maude sockets use TCP/IP sockets that require a unique IP address and a port to communicate between two parties in the same host or different hosts. In the paper [148], they demonstrate that Unix domain sockets are about two times faster than TCP/IP sockets. Besides, it is mandatory to convert data to a string representation before sending them over the network with Maude sockets, while it is not with meta-interpreters. Therefore, the use of meta-interpreters is effective in Par-Maude-NPA-2.

7.2 Maude

Maude is a declarative language and high-performance tool that focuses on simplicity, expressiveness, and performance to support the formal specification and analysis of concurrent programs/systems in rewriting logic. The language can directly specify membership equational logic [67], which extends many-sorted and order-sorted equational logics with extra membership predicates, and rewriting logic [127], and the tool provides several formal analysis methods, such as reachability analysis and LTL model checking. This section gives the syntax of the Maude language in a nutshell (see [13] for more detail).

Functional Modules

In Maude, a membership equational theory (Σ, E) is specified by a *functional model* M, declared with the syntax: `fmod` M `is` (Σ, E) `endfm`, where (Σ, E) may contain a set of declarations as follows:

- importations of previously defined modules (`protecting` ... or `extending` ... or `including` ...)
- declarations of sorts (`sort` s `.` or `sorts` s s' `.`)
- subsort declarations (`subsort` s `<` s' `.`)
- declarations of function symbols (`op` f `:` s_1 ... $s_n \to s$ `.`)
- declarations of variables (`vars` v v' `.`)
- unconditional equations (`eq` t `=` t' `.`)
- conditional equations (`ceq` t `=` t' `if` *cond* `.`)
- membership axioms (`mb` t `:` s `.` or `cmb` t `:` t' `if` *cond* `.`)

where $s, s_1, \ldots, s_n$ are sort names, v, v' are variable names, t, t' are terms, and *cond* is a conjunction of equations (e.g., t = t') and/or memberships (e.g., t : s). `protecting` requires to use an imported module such that the model denoted by it should be preserved. `extending` allows using an imported module such that new entities can be added to the model denoted by it. `including` allows using an imported module such that a relation, such as equality, between two different entities of the model denoted by it that can be introduced as well.

The following functional module defines natural number addition in Peano notation.

```
fmod PNAT is
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .

  op 0 : -> Zero [ctor] .
  op s : Nat -> NzNat [ctor] .
  op _+_ : NzNat Nat -> NzNat [assoc comm] .
  op _+_ : Nat Nat -> Nat [ditto] .

  vars N M : Nat .

  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

The module name is `PNAT`. There are some sorts: `Zero` for zero domain, `NzNat` for non-zero natural number domain, `Nat` for natural number domain. The sorts `Zero` and `NzNat` are subsorts of sort `Nat`. `0` is a constant of sort `Zero`. There are two function symbols: `s` and `_+_`, where underbars indicate argument positions. The function symbol `_+_` has two operators with

different sorts of arguments, which is called *subsort overloading*. The ctor attribute is declared for 0 and s as data *constructor* to distinguish them from the defined function _+_, which is defined by two equations. We declare _+_ with the assoc and comm attributes as an associative and commutative operator. The second operator for the function symbol _+_ contains the ditto attribute, meaning that it also has the assoc and comm attributes as the first operator. The two equations are used as *equational rules* to perform the simplification in which instances of the lefthand side pattern are replaced by the corresponding instances of the righthand side. The process is called *term rewriting*, and the result of simplifying a term is called its *normal form*. We can use the red command in Maude to reduce a term to its normal form with respect to the equations in the specification. For example, we can reduce the term s(s(s(0))) + s(0) and get its normal form as follows:

```
Maude> red in PNAT : s(s(s(0))) + s(0) .
result NzNat: s(s(s(s(0))))
```

System Modules

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is specified by a *system module*, declared with the syntax: mod $\mathcal{R}$ is (Σ, E, R) endm, where (Σ, E, R) may contain all possible declarations in (Σ, E) and *rewrite rules* in R as follows:

- unconditional rewrite rules (rl [*label*] : u => v .)
- conditional rewrite rules (crl [*label*] : u => v if *cond* .)

where *label* is the name of a rewrite rule, u, v are terms, and *cond* is a conjunction of equations, memberships, and/or rewrites (e.g., t => t'). Rewrite rules are also computed by rewriting from left to right modulo the equations in the system module and regarded as *local transition rules* in a possibly concurrent system. Furthermore, we can formally specify concurrent systems in an object-oriented style. Concurrent object systems are modeled by a set of *objects* that can interact with each other by sending and receiving *messages*.

The module CONFIGURATION in the prelude.maude file in Maude provides basic sorts and constructors for modeling object-based systems as follows:

```
mod CONFIGURATION is
  sorts Object Msg Configuration .
  subsorts Object Msg < Configuration .
  op none : -> Configuration [ctor].
  op __ : Configuration Configuration -> Configuration [assoc comm config id: none] .

  sorts Oid Cid .
  sorts Attribute AttributeSet .
```

```
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: none] .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
endm
```

In Maude, each state of a system is called a *configuration* that is modeled as a term of sort `Configuration`, which has the structure of a *multiset* made up of objects and messages. Objects and messages are modeled as terms of sorts `Object` and `Msg`, respectively. Multiset union for configurations is denoted by empty syntax (`none`) and a juxtaposition operator (`__`), which is declared associative and commutative and has `none` as its identity. For object syntax, there are four sorts introduced: `Oid` (object identifiers), `Cid` (class identifiers), `Attribute` (an attribute of an object), and `AttributeSet` (multisets of attributes). In this syntax, an *object* is a term of sort `Object` that has the following form:

< O : C | att_1 , ..., att_n >

where `O` is the object's identifier of sort `Oid`, `C` is the object's class identifier of sort `Cid`, and $att_1, \ldots, att_n$ are the object's attributes of sort `AttributeSet`. A *message* is a term of sort `Msg`, where the declaration defines the syntax of the message $m(v_1, \ldots, v_n)$ and the sorts $s_1, \ldots, s_n$ of its parameters $v_1, \ldots, v_n$ as follows:

op m : s_1 ... s_n -> Msg [ctor] .

Although messages do not have a fixed syntactic form, we follow a convention that the first and second arguments of a message are the identifier of its destination object and the identifier of its source object, respectively. For example, let us suppose to specify a client-server communication in which there are several clients and servers, and the status of each server or client is either *idle* or *busy*. Each server can have many clients but each client can communicate with only one server. If a client C is *idle*, the client sends a request `N`, a natural number, to a server `S` and becomes *busy*. If the server `S` is *idle*, then `S` receives the request (message), becomes *busy*, and returns `N + 1` to C as a message, meaning that the server increments `N`. Suppose that only the server knows how to increment a natural number. A *busy* client can receive an answer and become *idle*, and a *busy* server can become *idle* at any time. The system can be specified by the following system module:

```
mod CLIENT-SERVER is
  protecting NAT .
  including CONFIGURATION .
  sorts Status .

  ops Client Server : -> Cid [ctor] .          ops idle busy : -> Status [ctor] .
  op status :_ : Status -> Attribute [ctor] .   op val :_ : Nat -> Attribute [ctor] .
```

```
  op to :_ : Oid -> Attribute [ctor] .          op m : Oid Oid Nat -> Msg [ctor] .

  vars N N' : Nat .
  vars C S : Oid .

  rl [req]: < C : Client | status : idle, val : N, to : S >
  => < C : Client | status : busy, val : N, to : S > m(S, C, N) .

  rl [repl]: < S : Server | status : idle > m(S, C, N)
  => < S : Server | status : busy > m(C, S, (N + 1)) .

  rl [recv]: < C : Client | status : busy, val : N, to : S > m(C, S, N')
  => < C : Client | status : idle, val : N', to : S > .

  rl [idle]: < C : Server | status : busy >
  => < C : Server | status : idle > .
endm
```

where the four rewrite rules are given by the names `req`, `repl`, `recv`, and `idle`, respectively. We can freely define the syntax of attributes. However, we follow the standard Maude notation. For example, the attribute `status` with syntax `status :_` we are able to write server objects as `< S : Server | status : idle >`.

Let us suppose that there is a sever `s` communicating with two clients `c1` and `c2` in the client-server system. Initially, the status of each `s`, `c1`, and `c2` is *idle*, the values of `val` attributes of `c1` and `c2` are 3 and 4, respectively, and the value of the `to` attribute of each `c1` and `c2` is `s`. The initial state (referred to as `init`) and the object identifiers (`s`, `c1`, and `c2`) are defined in the following module:

```
mod CLIENT-SERVER-TEST is
  extending CLIENT-SERVER .
  ops s c1 c2 : -> Oid .
  op init : -> Configuration .
  eq init = < s : Server | status : idle >
            < c1 : Client | status : idle, val : 3, to : s >
            < c2 : Client | status : idle, val : 4, to : s > .
endm
```

Given `init`, the rewrite rule `req` can be applied to the term expressing it at two positions, meaning that there are at least two execution traces that start with `init`. Let us show a fragment of one possible execution trace of the system in Fig. 7.1. Note that we ignore the `to` attribute for simplicity because only one server is used. The first state transition from `init`

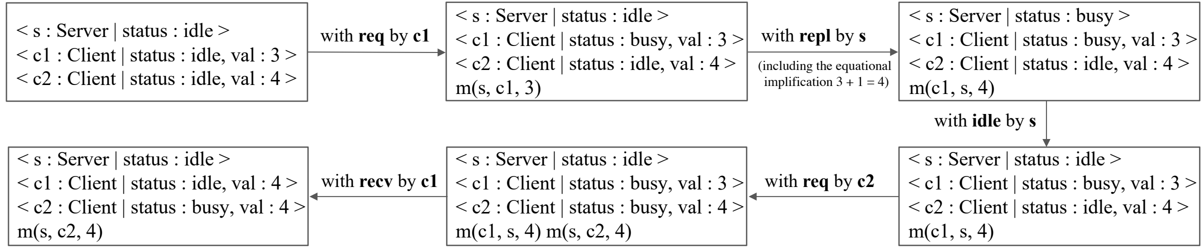


Figure 7.1: A fragment of one possible execution trace of the client-server system

to another state says that `c1` sends the message `m(s, c1, 3)` to `s` and changes its status to *busy* by applying the rewrite rule `req`. Likewise, the second state transition indicates that `s` receives the message, replies to `c1` with the message `m(c1,s,4)`, and changes its status to *busy* by applying the rewrite rule `repl`, where 4 is obtained from the equational simplification of $3 + 1$. The third state transition says that the status of `s` changes from *busy* to *idle* by applying the rewrite rule `idle`. The fourth state transition indicates that `c2` sends the message `m(s, c2, 4)` to `s` and changes its status to *busy* by applying the rewrite rule `req`. The last transition says that `c1` receives the replied message from `s` and changes its status and the value of the `val` attribute to *idle* and 4, respectively, by applying the rewrite rule `recv`.

Meta-interpreters

A meta-interpreter is an external object with an independent Maude interpreter, which has its own module and view databases. Each meta-interpreter is run in a separate process that can send and receive messages to/from other objects. We can work with meta-interpreters via the module `META-INTERPRETER` in the `meta-interpreter.maude` file in Maude, which contains several sorts, constructors, a built-in object identifier `interpreterManager`, and a collection of command and response messages as follows:

```
mod META-INTERPRETER is
   protecting META-LEVEL .
   including CONFIGURATION .

   sort RewriteCount .
   subsort Nat < RewriteCount .
   sorts InterpreterOption InterpreterOptionSet .
   subsort InterpreterOption < InterpreterOptionSet .

   op none : -> InterpreterOptionSet [ctor] .
   op interpreter : Nat -> Oid [ctor] .
```

```
  op createInterpreter : Oid Oid InterpreterOptionSet -> Msg [ctor msg ...] .
  op createdInterpreter : Oid Oid Oid -> Msg [ctor msg ...] .

  op insertModule : Oid Oid Module -> Msg [ctor msg ...] .
  op insertedModule : Oid Oid -> Msg [ctor msg ...] .
  ...
  op reduceTerm : Oid Oid Qid Term -> Msg [ctor msg ...] .
  op reducedTerm : Oid Oid RewriteCount Term Type -> Msg [ctor msg ...] .
  ...
  op quit : Oid Oid -> Msg [ctor msg ...] .
  op bye : Oid Oid -> Msg [ctor msg ...] .
  op interpreterManager : -> Oid [special (...)].
endm
```

where some attributes and operators are omitted, where `...` is written. The module imports two modules `META-LEVEL` and `CONFIGURATION` so that we can use reflective programming (meta-programming) facilities, such as moving up and down between reflective levels, and object-based programming facilities, such as communicating with external objects, respectively.

Let us suppose to specify a system that has a server and the server is requested to increment a natural number. However, the increment of the natural number is not carried out by the server but by a meta-interpreter independently. The system can be specified by the following module:

```
mod SERVER is
  extending META-INTERPRETER .

  op Server : -> Cid .
  op aServer : -> Oid .
  op val :_ : Nat -> Attribute .

  vars O O' MI : Oid .
  var AS : AttributeSet .
  var T : Term .
  var RT : Type .
  vars N C : Nat .

  rl [loadMod]: < O : Server | AS > createdInterpreter(O, O', MI)
  => < O : Server | AS > insertModule(MI, O, upModule('NAT, true)) .

  crl [redTerm]: < O : Server | val : N, AS > insertedModule(O, MI)
  => < O : Server | val : N, AS > reduceTerm(MI, O, 'NAT, T)
  if T := '_+_[upTerm(N), upTerm(1)] .
```

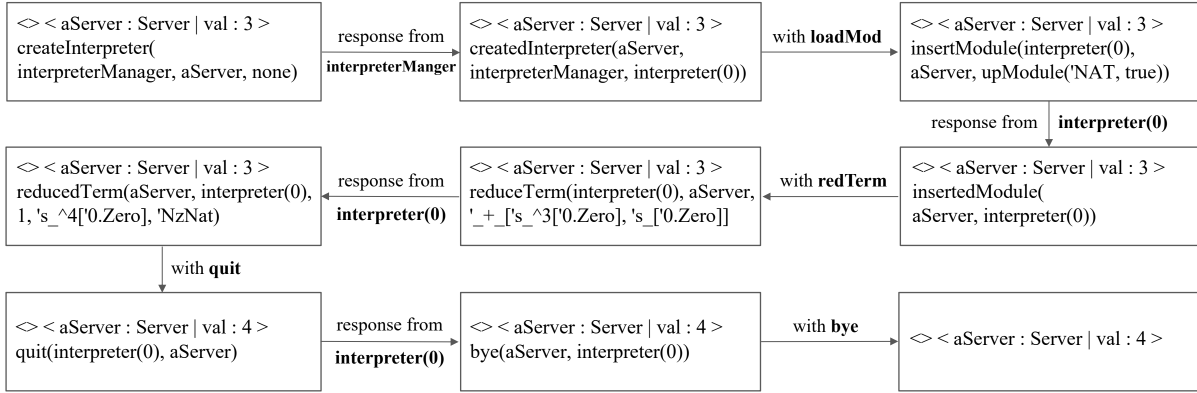


Figure 7.2: The execution trace of the system with a server and a meta-interpreter for incrementing a natural number.

```
  rl [quit]: < O : Server | val : N, AS > reducedTerm(O, MI, C, T, RT)
  => < O : Server | val : downTerm(T, 0), AS > quit(MI, O) .

  rl [bye]: < O : Server | AS > bye(O, MI)
  => < O : Server | AS > .
endm
```

where the four rewrite rules are given by the names `loadMod`, `redTerm`, `quit`, and `bye`, respectively. A server has the `val` attribute that stores a natural number. Some messages are exchanged by a server and a meta-interpreter to increment the natural number stored in the `val` attribute. We can request the system to increment a natural number, namely 3, stored in the `val` attribute of a server and get its result by the following command:

```
Maude> erewrite in SERVER :
   <> < aServer : Server | val : 3 >
   createInterpreter(interpreterManager, aServer, none) .
result Configuration: <> < aServer : Server | val : 4 >
```

where `<>` symbol means communicating with external objects, namely meta-interpreters, `aServer` is an object identifier of class `Server`, and `none` is an empty set of options. Rewriting with external objects is conducted by the external rewrite command `erewrite` in Maude.

Fig. 7.2 shows the execution trace for the rewrite above. Initially, the (command) message `createInterpreter` is sent by `aServer` to the built-in object whose identifier is `interpreterManager` in order to create a meta-interpreter that increments the natural number stored in the `val` attribute of `aServer`. The first state transition says that `interpreterManager` creates successfully a meta-interpreter whose id is `interpreter(0)` and answers using the (response) message `createdInterpreter` to `aServer`. The second state transition indicates that `aServer` receives the message, where the third parameter of the message is `interpreter(0)`, and sends the

message `insertModule` to `interpreter(0)` so as to load the module `NAT` into the interpreter, where the third parameter of the message is the meta-representation of module `NAT`. The second state transition is carried out by applying the rewrite rule `loadMod`. The third state transition says that `interpreter(0)` loads the module `NAT` into its own module database successfully and sends the message `insertedModule` to `aServer`. The fourth state transition indicates that `aServer` receives the message; the term $3 + 1$ is meta-represented as `'_+_['s_^3['0.Zero], 's_['0.Zero]]`; and the message `reduceTerm` in which the term is used as the third parameter is sent to `interpreter(0)` by `aServer`. The fourth state transition is carried out by applying the rewrite rule `redTerm`. The fifth state transition says that `interpreter(0)` reduces the term successfully and sends the message `reducedTerm` to `aServer`, where the number of rewrites, the reduced term, and the type of the reduced term are in the third, fourth, and fifth parameters, respectively. The reduced term is currently meta-represented as `'s_^4['0.Zero]`. The sixth state transition indicates that `aServer` receives the message, downs the reduced term to a natural number, namely 4, sets it to the value of the `val` attribute, and sends the message `quit` to `interpreter(0)` to delete the meta-interpreter because we do not need it anymore. The sixth state transition is carried out by applying the rewrite rule `quit`. The seventh state transition says that at the same time when `interpreter(0)` is deleted successfully, the message `bye` is sent to `aServer` by `interpreter(0)`. The last state transition indicates that `aServer` just ignores the message by applying the rewrite rule `bye`. The system terminates subsequently and the incremented natural number, namely 4, can be observed as shown in the output of Maude. Note that we do not take any error into account to make the example concise.

7.3 Maude-NPA

Maude-NPA [16] is a verification tool for analyzing cryptographic protocols modulo an equational theory, which is written in Maude and so its specifications follow the Maude syntax. This section gives an overview of Maude-NPA in a nutshell (see [16] for more detail).

7.3.1 A Protocol Specification in Maude-NPA

A protocol specification specified in Maude-NPA consists of three modules as follows:

- `PROTOCOL-EXAMPLE-SYMBOLS` defines the syntax of the protocol, which consists of sorts, subsorts, and operators.
- `PROTOCOL-EXAMPLE-ALGEBRAIC` defines the algebraic properties of the operators, consisting of equational rules (equations) and equational axioms (axioms).
- `PROTOCOL-SPECIFICATION` defines the actual behaviors of the protocol using the Dolev-Yao strand space model [58, 59] and attack states, consisting of at least three equations by which three pre-defined operators in the module are defined as follows:

- STRANDS-DOLVEYAO describes the capabilities of the intruder.
- STRANDS-PROTOCOL describes the strands of the honest principals. Note that the intruder also can do anything that honest principals can do.
- ATTACK-STATE(N), where N is a natural number representing the attack state id, allows us to specify an attack for which we would like to prove the protocol secure or insecure.

Let us illustrate how to specify the Needham-Schroeder Public Key (NSPK) Protocol [46] in Maude-NPA as an example. The NSPK protocol uses the standard Alice-and-Bob notation with three messages exchanged as follows:

1. $A \to B : pk(B, A; N_A)$
2. $B \to A : pk(A, N_A; N_B)$
3. $A \to B : pk(B, N_B)$

where A and B denote Alice and Bob principal identifiers (names), N_A and N_B denote the nonces generated by A and B, and $pk(A, ...)$ and $pk(B, ...)$ are the encrypted data of names and/or nonces, where ... is written, with the public keys of A and B, respectively. $A \to B$ denotes that the data is sent from A to B.

The following modules are a Maude-NPA specification of the NSPK protocol, where three dashes (---) denote comments in Maude.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .

  sorts Name Nonce Key .
  subsort Name Nonce Key < Msg .
  subsort Name < Key .
  subsort Name < Public .

  op pk : Key Msg -> Msg [frozen] .
  op sk : Key Msg -> Msg [frozen] .

  op n : Name Fresh -> Nonce [frozen] .

  op a : -> Name . --- Alice
  op b : -> Name . --- Bob
  op i : -> Name . --- Intruder

  op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
endfm
```

The module PROTOCOL-EXAMPLE-SYMBOLS imports in protecting mode the module DEFINITION-PROTOCOL-RULES that is a module in Maude-NPA and provides some sorts, such as the sort Msg that represents the messages in the protocol, the sort Fresh that is used to identify terms that must be unique, and the sort Public that is used to identify terms that are publicly available. The NSPK protocol uses public-key cryptography and its principals exchange encrypted data that are made from names and/or nonces. Hence, we define sorts Name, Key, and Nonce to represent names, keys, and nonces, respectively. All three sorts are subsorts of sort Msg. Principal names are publicly available and used as keys to encrypt data, and so the sort Name is a subsort of both sorts Public and Key. The following operators are defined: pk for public key encryption, sk for private key encryption, n for nonces that are unguessable values for principals, and _;_ for message concatenation. The frozen attribute tells Maude not to attempt to apply rewrites at arguments of those symbols. The gather (e E) attribute indicates that the operator _;_ should be parsed as right-associativity. We use some constants a, b, and i, denoting an initiator, a responder, and an intruder, respectively. Although there are some specific constants, the number of principals is unbounded, which is achieved by using variables in terms representing states by Maude-NPA. Those specific constants are used to define our attacks so that we can avoid unnecessary cases, for example, that the initiator and the responder are the same principal.

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  var Z : Msg .
  var Ke : Key .
  eq pk(Ke,sk(Ke,Z)) = Z [variant] .
  eq sk(Ke,pk(Ke,Z)) = Z [variant] .
endfm
```

The module PROTOCOL-EXAMPLE-ALGEBRAIC imports in protecting mode the module PROTOCOL-EXAMPLE-SYMBOLS defined above. We declare two equations describing the relationship between the public and private key encryption, which is called encryption/decryption cancellation. The variant attribute denotes that the two equations are not regular Maude equations used for simplification, but are equations used for variant-based equational unification [149] available in Maude.

```
fmod PROTOCOL-SPECIFICATION is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  protecting DEFINITION-PROTOCOL-RULES .
  protecting DEFINITION-CONSTRAINTS-INPUT .

  var Ke : Key .
  vars X Y Z : Msg .
```

```
  vars r r' : Fresh .
  vars A B : Name .
  vars N N1 N2 : Nonce .

  eq STRANDS-DOLEVYAO
   = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
     :: nil :: [ nil | -(X ; Y), +(X), nil ] &
     :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
     :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
     :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
     :: nil :: [ nil | +(A), nil ]
  [nonexec] .

  eq STRANDS-PROTOCOL
   = :: r ::
     [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; N)), +(pk(B, N)), nil ] &
     :: r ::
     [ nil | -(pk(B,A ; N)), +(pk(A, N ; n(B,r))), -(pk(B,n(B,r))), nil ]
  [nonexec] .

  eq ATTACK-STATE(0)
   = :: r ::
     [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
     || n(b,r) inI, empty
     || nil
     || nil
     || nil
  [nonexec] .
endfm
```

The module PROTOCOL-SPECIFICATION imports in protecting mode the module PROTOCOL-EXAMPLE-SYMBOLS defined above and two more modules in Maude-NPA. We declare some variables that are used for specifying intruder strands, principal strands, and attack states. Each strand is a sequence of positive and negative messages describing each principal's executing a protocol or the intruder's performing actions as follows:

$$:: r_1,\ \ldots, r_j ::\ [\ m_1^{\pm},\ \ldots, m_i^{\pm} \mid m_{i+1}^{\pm},\ \ldots, m_k^{\pm}]$$

where $r_1,\ \ldots, r_j$ are the variables of sort Fresh uniquely generated in the strand, a positive message m^+ describes sending the message m, and a negative message m^- describes receiving the message m. The vertical bar is used to distinguish between present and future when the strand appears in a state. Messages before the bar were sent or received in the past, while

messages after the bar will be sent or received in the future. Strands are also used in a protocol specification to build rewrite rules for the backward narrowing. The vertical bar in such a strand is not significant. By convention, the vertical bar is assumed to be put right after the initial `nil` in each strand in a protocol specification.

For specifying the intruder capabilities, all intruder strands follow a convention: a sequence of negative variables followed by at least one positive message combining all previous variables under a function symbol. For example, the intruder can concatenate two arbitrary messages represented by the following strand:

```
:: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ]
```

The full intruder capabilities of the NSPK protocol are specified in `STRANDS-DOLEVYAO`. For specifying the honest protocol principals, we represent each role of an initiator and a responder as a strand containing received messages and sent messages as the behavior of the NSPK protocol that is specified in `STRANDS-PROTOCOL`. For specifying attack states, each attack state contains several components separated by the symbol `||`. In this example, we specify only the first two components of the attack state: (i) a set of strands and (ii) some positive intruder knowledge expected to appear in the attack. The other three components are set to `nil`. In general, we should specify only the first two components and the last component that is used for authentication attacks. The attack state in the NSPK protocol specification is specified in `ATTACK-STATE(0)` meaning that the intruder can glean the nonce generated by Bob. Hence, we include Bob's strand in the attack and describe specific nonce `n(b, r)` in the intruder knowledge. To conduct the attack, we can use the following command in Maude-NPA:

```
Maude> reduce in MAUDE-NPA : run(0, unbounded) .
result ShortIdSystem: < 1 . 5 . 2 . 7 . 2 . 4 . 2 . 1 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh))),
   +(n(b, #0:Fresh)), nil]  &
:: nil ::
[ nil |
   -(pk(i, a ; n(a, #1:Fresh))),
   +(a ; n(a, #1:Fresh)), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(a ; n(a, #1:Fresh)),
```

```
   +(pk(b, a ; n(a, #1:Fresh))), nil]  &
:: #1:Fresh ::
[ nil |
   +(pk(i, a ; n(a, #1:Fresh))),
   -(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   +(pk(i, n(b, #0:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(b, a ; n(a, #1:Fresh))),
   +(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   -(pk(b, n(b, #0:Fresh))), nil] )
|
pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh)) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(b, a ; n(a, #1:Fresh)) !inI,
pk(i, n(b, #0:Fresh)) !inI,
pk(i, a ; n(a, #1:Fresh)) !inI,
n(b, #0:Fresh) !inI,
(a ; n(a, #1:Fresh)) !inI
|
+(pk(i, a ; n(a, #1:Fresh))),
-(pk(i, a ; n(a, #1:Fresh))),
+(a ; n(a, #1:Fresh)),
-(a ; n(a, #1:Fresh)),
+(pk(b, a ; n(a, #1:Fresh))),
-(pk(b, a ; n(a, #1:Fresh))),
+(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
-(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh))),
-(pk(i, n(b, #0:Fresh))),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

where 0 is the attack state id and `unbounded` is unbounded depth used in the attack. The result of the command returns an initial state as a counterexample. It means that the NSPK protocol does not satisfy the nonce secrecy property (NSP) as the flaw found by Lowe [45]. The initial state is represented by a unique state id followed by four sections separated by the symbol `|` in the following order: a set of current protocol and intruder strands, intruder

knowledge, a sequence of actual messages exchanged, and never pattern. Note that the initial state is simplified before it is returned. For example, the ghost list section is eliminated from the initial state that is explained in detail below.

7.3.2 How Maude-NPA Works

Maude-NPA starts from an attack state, a final insecure state, to perform a backward reachability analysis that determines whether or not it is reachable from an initial state, which has no further backward steps. If that is the case, the initial state is a counterexample. The backward search is performed by backward narrowing with symbolic execution because the attack state is a term with logical variables. Each backward narrowing step can be regarded as the reverse direction of state transitions, such as sending or receiving a message by principals, or manipulating a message by intruders. Given a symbolic state, a backward narrowing step is performed to return a previous symbolic state in the protocol. Thereby, we can obtain all successor states (in the backward sense) from the state. In Maude-NPA, each state found during the backward analysis is represented by a unique state id followed by five sections separated by the symbol `||` in the following order: a set of current protocol and intruder strands, intruder knowledge, a sequence of messages, a ghost list, and never pattern.

For each state, the state id is a unique id assigned to each state during the backward analysis. The set of current strands represents the messages that were sent or received in the past (those messages before the symbol `|`) and the messages that will be sent or received in the future (those messages after the symbol `|`) in each strand. The strand set also indicates how to advance each strand in the execution process by partial substitutions for the messages in each strand. The intruder knowledge represents what messages the intruder knows (symbol `_inI`) or does not know yet (symbol `_!inI`) at each state. The sequence of messages denotes the actual sequence of messages communicated to reach the state. The ghost list is extra information for optimization in the super lazy intruder technique to reduce the state space. The never pattern is used for authentication attacks.

We can divide the whole process of Maude-NPA into two main stages. In the first stage, given a protocol specification $\mathcal{P}$ and an equational theory $E_{\mathcal{P}}$, Maude-NPA needs to do as follows:

- Extracting the attack state St from the protocol given an attack state id file.
- Building rewrite rules $R_{\mathcal{P}}$ based on the behavior of the protocol specified in the form of intruder and regular strands along with some pre-defined rewrite rules in the Maude-NPA specification.
- Generating grammars that represent infinite sets of states unreachable for the intruder to reduce the state space.

In the second stage, Maude-NPA performs the backward narrowing reachability analysis from the attack state St using the relation $\rightsquigarrow_{R_{\mathcal{P}}^{-1},E_{\mathcal{P}}}$ where $R_{\mathcal{P}}^{-1}$ is the set of rewrite rules derived from $R_{\mathcal{P}}$ by inverting its rewrite rules. Maude-NPA basically uses the breadth-first search to explore the state space. There are three main steps needed to do for each layer exploration as follows:

- The first step is to generate all successor states for the next layer given a set of states in the current layer. The initial layer is layer 0, the attack state is located at layer 0, and all successor states of the attack state with respect to the relation $\rightsquigarrow_{R_{\mathcal{P}}^{-1},E_{\mathcal{P}}}$ are located at layer 1. This step also consists of almost all techniques to reduce the state space except for the transition subsumption technique, which is used in the second step below.
- The second step is to simplify the successor states by the transition subsumption technique for removing states that are subsumed by either other states in the successor states or visited states (history states).
- Lastly, the third step filters states from the previous step by using the history states to avoid state duplications and rules out initial states as counterexamples. The successor states without duplication are then stored in the history states as visited states and the depth bound is decreased by one.

The cycle repeats until any initial states (counterexamples) are found, a depth bound is reached, or no states are found for the next layer.

The first step in the second stage actually performs the backward narrowing just by one step to obtain all successor states from a given set of states in a layer. The successor states then go through a series of optimization steps, such as giving priority to input messages in strands, early detection of inconsistent states, the super lazy intruder, and filtering states by the grammars. Given a set of states in layer l, for each state in the set, Maude-NPA performs the backward narrowing to obtain its successor states in layer $l+1$, which is referred to as step (1) in this chapter. We are aware that the backward narrowing for each given state from the set of states can be executed independently, which opens an opportunity for parallelization. In the next section, we will describe how to parallelize step (1) at each layer in which the successor states are generated in parallel. Note that many reduction techniques are also included in this step and the parallel version does not alter the number or form of the states in the state space.

The second step in the second stage plays an important role to reduce the state space in Maude-NPA, which is referred to as step (2), which may transform an infinite-state system into a finite one [150] and is also time-consuming because of its complexity. Basically, it performs two sub-steps for the transition subsumption as follows:

- First, for each state in the successor states obtained from step (1), we check whether the state is implied by other states in the successor states, which is referred to as step (2.1)

in this chapter. If so, the state is discarded. Otherwise, we keep the state. Once step (2.1) is complete, we obtain a new set of successor states such that no state is implied by other states.

- Second, for each state in the successor states obtained from step (2.1), we check whether the state is implied by some states in the history states, which is referred to as step (2.2) in this chapter. If so, the state is discarded. Otherwise, we keep the state. Once step (2.2) is complete, we obtain a new set of successor states such that no state is implied by any state in the history states. The successor states are used as the input for the third step of the second stage, which is referred to as step (3).

The transition subsumption is complex and a heavy task in Maude-NPA when the number of successor states and history states is considerably large. Therefore, we can improve the transition subsumption in Maude-NPA by parallelizing step (2.1) and step (2.2) at each layer, which are described in detail in the next section. Meanwhile, we do not conduct step (3) in parallel because the task in step (3) is not complex and can be completed quickly in sequence even the number of history states is considerably large.

7.4 Parallel Maude-NPA and Its Tool Support

The support tool is implemented in Maude to conveniently extend the implementation of what has been developed in Maude-NPA. We use object-based programming that can model an object-based system and meta-interpreters that can handle jobs independently to build a parallel version of Maude-NPA with a master-worker model.

7.4.1 How to Parallelize Maude-NPA

As mentioned above, we parallelize the backward narrowing and the transition subsumption at each layer in Maude-NPA. In our tool, a master maintains a shared cache that is a set of visited states (history states), while each worker maintains some shared information, such as the module used to conduct the backward narrowing, the grammars generated from the protocol under verification for optimization, and the filter information specified by users. The use of the shared cache prevents jobs that have been processed from being assigned to workers. The use of the shared information prevents loading them again from each worker whenever it is requested to handle a job. We maintain the status of the master whose value is one of *narrowing*, *implication*, *implicationH*, *filter*, and *stop* to distinguish that the master is processing step (1), step (2.1), step (2.2), step (3), and checking for termination, respectively. Note that those steps are performed in this order at each layer in the tool. There are four kinds of jobs that are sent from the master to workers: *narrowing*, *implication*, *combination*, and *implicationH* when we use different kinds of jobs to parallelize each step. *narrowing* jobs are used in step

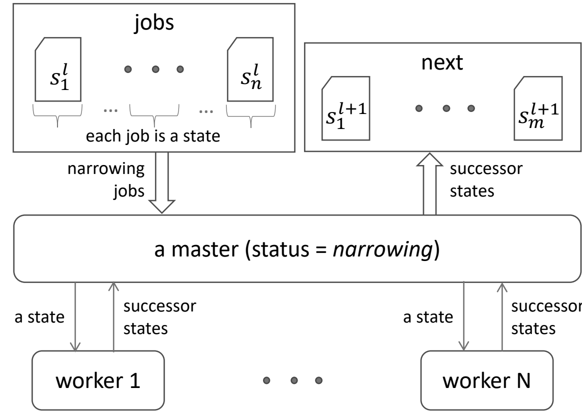


Figure 7.3: Conducting the backward narrowing in step (1) from the states at layer l in parallel.

(1), *implication* and *combination* jobs are used in step (2.1), and *implicationH* jobs are used in step (2.2). Depending on the kind of the job, a worker is requested to handle a different task as follows:

- *narrowing* requests a worker to conduct the backward narrowing given a state in step (1). The result is a set of successor states reachable from the state.

- *implication* requests a worker to conduct the implication for a given set of states in step (2.1). The result is a new set of states such that no state can be implied by other states in the set.

- *combination* requests a worker to combine two given sets of states in step (2.1) by a slightly different implication, which is explained in detail later. Note that each set of states has been simplified by the implication before. The result is a set of states such that no state can be implied by other states in the set.

- *implicationH* requests a worker to conduct the implication for a set of states with history states given in step (2.2). The result is a set of states such that no state in the set can be implied by any state in the history states.

There is only one kind of job that is sent from workers to the master. As soon as a worker completes a job assigned to it by the master, the worker sends its result in the form of a set of states to the master, delivering a job or a bunch of jobs made by the worker to the master. Depending on the status of the master, the jobs are stored accordingly. The very initial job is made by the master, while all the other jobs are made by workers and basically sent to the master. Jobs are assigned to workers by the master unless the jobs have been tackled.

Fig. 7.3 shows an overview of how to conduct step (1) in parallel for states located at layer l. The status of the master is currently *narrowing*. Given a set of states $s_1^l, \ldots, s_n^l$ stored in `jobs` at layer l as input, for each state, the master constructs a job whose type is *narrowing* and

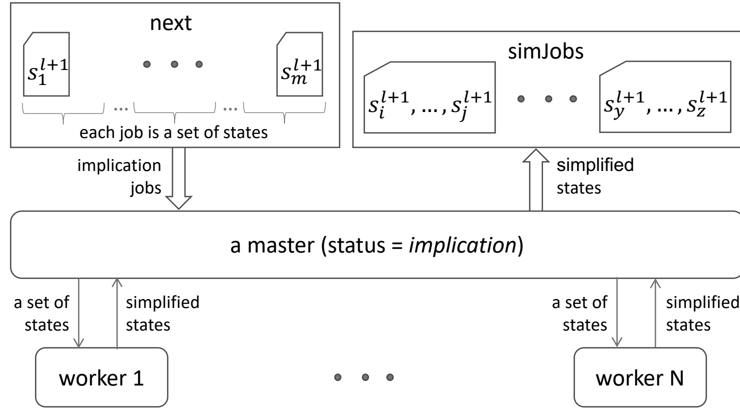


Figure 7.4: Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel.

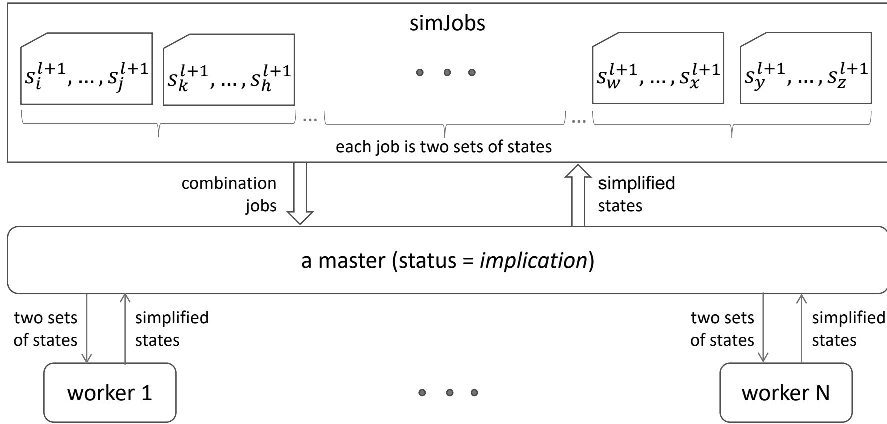


Figure 7.5: Conducting the transition subsumption in step (2.1) for the states at layer $l+1$ in parallel (continuously).

sends it to a worker. As soon as a worker receives the job, it conducts the backward narrowing just by one step from the state encapsulated in the job independently to obtain its successor states at layer $l+1$. The worker then sends the successor states to the master and waits for a new job. The master will store the successor states into `next` as output, all possible next jobs (successor states) of the next layer, namely layer $l+1$. If there are neither unprocessed jobs left nor jobs being processed by workers, step (1) is complete. The master changes its status to *implication* to move to conduct step (2.1).

Fig. 7.4 and Fig. 7.5 show an overview of how to conduct step (2.1) in parallel for those successor states in `next` obtained from step (1). The status of the master is currently *implication*. Firstly, the master divides the successor states in `next` into multiple partitions in which each partition is a set of states (see Fig. 7.4). The number of partitions is calculated based on the number of workers and the minimum number of states that should be assigned to a worker, such that the number of partitions is smaller than or equal to the number of workers, which will

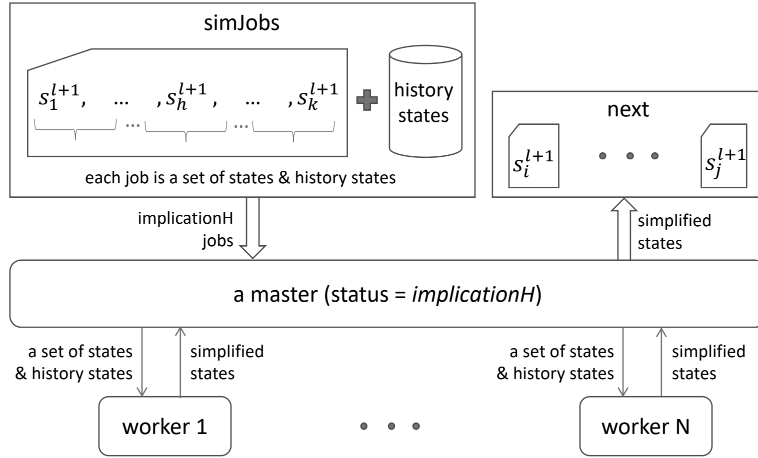


Figure 7.6: Conducting the transition subsumption in step (2.2) for states with history states at layer $l + 1$ in parallel.

be described in detail later. If the number of partitions is one, we do not conduct it in parallel but in sequence. For each partition, the master constructs a job whose type is *implication* and sends it to a worker. As soon as a worker receives the job, it conducts the implication inside the set of states encapsulated in the job to obtain a set of simplified states such that no state is implied by other states in the set. The worker then sends the simplified states to the master that will be stored in `simJobs`, a queue of simplified jobs in which each job is a set of simplified states. If there are no jobs being processed by workers, the master moves to combine the simplified jobs in `simJobs` (see Fig. 7.5). For every two simplified jobs in `simJobs`, the master constructs a job whose type is *combination* and sends it to a worker. As soon as a worker receives the job, it combines the two sets of simplified states encapsulated in the job by a slightly different implication, which will be described in detail later. The result is a set of simplified states such that no state is implied by other states in the set. The worker then sends the set of simplified states to the master and waits for a new job. The master will store the set of simplified states into `simJobs` as a simplified job so that it can be combined with another set of simplified states subsequently. If there is only one simplified job left in `simJobs` and no jobs are being processed by workers, step (2.1) is complete. The master changes its status to *implicationH* to move to conduct step (2.2).

Fig. 7.6 shows an overview of how to conduct step (2.2) in parallel from a set of simplified successors states obtained from step (2.1). The status of the master is currently *implicationH* and there is only one simplified job in `simJobs` that is the set of successor states. The master dequeues `simJobs` to obtain the set of successor states and divides it into multiple partitions in which each partition is a set of states. The number of partitions is calculated based on the number of workers, the number of history states, and the minimum number of states that should be assigned to a worker, such that the number of partitions is smaller than or equal to the number of workers, which will be described in detail later. If the number of partitions is

one, we do not conduct it in parallel but in sequence. For each partition with history states, the master constructs a job whose type is *implicationH* and sends it to a worker. As soon as a worker receives the job, it conducts the implication for the set of states with the history states encapsulated in the job as described in step (2.2). The result is a set of simplified states such that no state in the set is implied by any state in the history states. The worker then sends it to the master to store in `next`. If there are no jobs being processed by workers, step (2.2) is complete. The master changes its status to *filter* to conduct step (3). The master then conducts step (3) in sequence and changes its status to *narrowing* for the next layer. The cycle repeats until any initial states (counterexamples) are found, a depth bound is reached, or no states are found for the next layer. If so, the master changes its status to *stop* to terminate the tool subsequently.

7.4.2 Job Scheduling by the Master

In summary, the master is in charge of the following tasks. For step (1), the master is in charge of collecting all successor states (jobs) from workers. For step (2.1), if it is necessary to conduct step (2.1) in parallel, the master is in charge of collecting (simplified) jobs from workers in which each job is a set of simplified states. Otherwise, the master is in charge of conducting step (2.1) in sequence. For step (2.2), if it is necessary to conduct step (2.2) in parallel, the master is in charge of collecting all states (jobs) from workers such that no state is implied by any state in history states. Otherwise, the master is in charge of conducting step (2.2) in sequence. For step (3), the master is in charge of conducting step (3) in sequence as Maude-NPA does. In addition, the master is in charge of distributing (or assigning) unprocessed jobs to workers. The master can stop the tool whenever counterexamples are found, there are no unprocessed jobs left for the next layer, or the tool reaches a depth bound. Meanwhile, each worker is responsible for processing a job assigned to it by the master. Depending on the kind of the job, the worker needs to handle its corresponding task as described above. The worker then constructs a job or a bunch of jobs and then sends them to the master. The master uses a set of states to store jobs in step (1), a queue of sets of states to store (simplified) jobs in step (2.1), and a queue of worker identifiers to distribute jobs to workers so that job distribution can be well-balanced. Note that the number of jobs in step (2.2) is always smaller than or equal to the number of available workers. Hence, we simply distribute each job to a worker and do not need to store such jobs.

Algorithm 7 shows the pseudo-code for job scheduling by the master. `workers` is a queue data structure that contains worker identifiers so that the master can assign jobs to workers. `jobs`, `next`, and `history` are set data structures. `jobs` contains jobs (states) that are distributed to workers in step (1), while `next` contains all possible next jobs (successor states) of the next layer. `history` contains all states explored at the moment. `simJobs` is a queue of sets of states that contains (simplified) jobs that are distributed to workers in step (2.1). Initially,

Algorithm 7: Job scheduling by the master.

```
input  : 𝒫 – a protocol specification
         Id – an attack state id in the protocol specification
         F – a filter
         BStep – the maximum number of backward steps
         N – a number of workers
output: empty or counterexamples

1  (jobs, simJobs, next, history) ← (empty, empty, empty, empty);
2  (M, GS, IS) ← initialize(𝒫, Id, BStep, F);
3  workers ← initializeInterpreters(M, GS, F, N);
4  jobs ← {IS}; history ← {IS}; initStates ← empty;
5  status ← narrowing;
6  while True do
7      narrowingStep() /* conduct step (1) in parallel                  */
8      implicationStep() /* conduct step (2.1) in parallel              */
9      implicationWithHistoryStep() /* conduct step (2.2) in parallel   */
10     filterStep() /* conduct step (3) in sequence                     */
11     if status = stop then
12         closeAllConnections();
13         return initStates;

14 function receivingData() is
15     for k ← 1 to N do
16         if DATA ← recv(worker_k) then
17             enqueue(workers, worker_k);
18             (IST) ← DATA;
19             if status = narrowing or status = implicationH then
20                 next ← next ∪ IST;
21             if status = implication then
22                 enqueue(simJobs, {IST});
```

simJobs is set to the empty queue, while jobs, next, and history are set to the empty set at line 1. In the first stage, Maude-NPA needs to build rewrite rules $R_{\mathcal{P}}^{-1}$ in the form of a module, an attack state, and grammars from a protocol specification $\mathcal{P}$. The module is used to perform the backward reachability analysis, the attack state is used as the beginning state, and the grammars are used to remove unreachable states for intruders. This stage is proceeded by the initialize function at line 2 with $\mathcal{P}$, *Id*, *BStep*, and F parameters that are a protocol specification, the id of an attack state in the specification, the maximum number of backward

steps, and a filter that is *+parallel* as default denoting the parallelization mode, respectively. The result of the function is deconstructed and stored in `M`, `GS`, and `IS`, which stand for the module, the grammars, and the attack state, respectively. Once the first stage is completed, we then prepare N workers by calling the `initializeInterpreters` function at line 3 in which some shared information, such as `M`, `GS`, and `F`, are loaded. Note that we also need to load many other modules into each interpreter from Maude as well as Maude-NPA in order to be able to handle jobs assigned to workers by the master. The function returns a set of worker identifiers, which is assigned to `workers` so that the master can request each worker to handle a job in parallel subsequently. `jobs` and `history` are updated to contain only the attack state at line 4. We use `initialStates` to store initial states (counterexamples) found by the tool that also is initially set to the empty set at line 4. We maintain `status` as the status of the master whose value is one of *narrowing*, *implication*, *implicationH*, *filter*, and *stop* as described above. The status of the master is initially set to *narrowing*, meaning that the master is ready for processing the backward narrowing at the first layer. The first `while` loop in the code fragment at lines 6–13 conducts the backward narrowing analysis for the protocol under verification. The master calls the `narrowingStep`, `impliationStep`, `implicationWithHistoryStep`, and `filterStep` functions in order to conduct step (1), step (2.1), and step (2.2) in parallel, and step (3) in sequence at each layer, respectively. Those functions are described in detail below. At the end of the `while` loop, the master checks whether `status` is *stop* for termination. If so, the master closes all connections and returns `initialStates` as the result. Otherwise, the master keeps on doing for the next layer.

In Algorithm 7, the code fragment at lines 14–22 defines the `receivingData` function that will be used in the `narrowingStep`, `impliationStep`, and `implicationWithHistoryStep` functions to receive data from workers for parallelization. Whenever the master receives `DATA` from each $worker_k$, the worker identifier $worker_k$ is enqueued to `workers` at line 17 so that a job can be assigned to $worker_k$ subsequently. Because `DATA` is only in the form of a set of states, the master deconstructs `DATA` into a set of states `IST` at line 18. The master then checks whether `status` is either *narrowing* or *implicationH*. If that is the case, the master inserts all states in `IST` (a bunch of jobs) to the set of possible successor states `next` at line 20. Otherwise, `status` is *implication* and so the master adds `IST` as an element (a simplified job) to `simJobs` at line 22. `next` is used to collect all possible successor states in step (1) and step (2.2), while `simJobs` is used to collect (simplified) jobs to distribute to workers in step (2.1).

Algorithm 8 shows the pseudo-code of the `narrowingStep` function in which we conduct the backward narrowing in step (1) in parallel for a given set of states at the layer being concerned (see Fig. 7.3 as well). At first, the master checks whether `status` is not *narrowing* at line 2. If so, it is not the case that we need to conduct step (1) here. The master exits the function immediately. Otherwise, the status of the master is currently *narrowing* and the jobs needed to be processed are currently stored in `jobs`. We start conducting step (1) in parallel in the code fragment at lines 4–15. In the first `while` loop, the master calls the `receivingData` function

Algorithm 8: Conducting the backward narrowing in step (1) at each layer in parallel.

```
1  function narrowingStep() is
2      if status != narrowing then
3          return;
4      while True do
5          receivingData();
6          while not isEmpty(workers) and not isEmpty(jobs) do
7              worker ← dequeue(workers);
8              IS ← getOne(jobs);
9              send(worker, narrowing, IS);
10         if isEmpty(jobs) and size(workers) = N then
11             if not isEmpty(next) then
12                 status ← implication;
13             else
14                 status ← stop;
15             return;
```

to collect jobs sent from workers at line 5. The code fragment at lines 6–9 checks continuously whether `workers` and `jobs` are not empty. If that is the case, the master dequeues `workers` to obtain a worker identifier, gets one job from `jobs`, constructs a job whose type is *narrowing*, and sends it to the worker at line 9. The code fragment at lines 10–15 checks whether there are neither unprocessed jobs left nor jobs being processed by workers. If so, the master checks whether `next` is not empty. If so, the master changes its status to *implication* at line 12 to move to conduct step (2.1). Otherwise, the master changes its status to *stop* at line 14 to terminate the tool because there is no state for the next layer. Lastly, the master exits the function at line 15. At this point, we have completed step (1) in parallel and the successor states are currently stored in `next`.

Algorithm 9 shows the pseudo-code of the `implicationStep` function in which we conduct the implication in step (2.1) in parallel for the successor states obtained from step (1) (see Fig. 7.4 and Fig. 7.5 as well). The master first checks whether `status` is not *implication* at line 2. If so, it is not the case that we need to conduct step (2.1) here. The master exits the function immediately. Otherwise, the status of the master is currently *implication* and the successor states are currently stored in `next`. As described above with Fig. 7.4, to conduct step (2.1) in parallel, we first divide the successor states into multiple partitions so that each partition can be simplified independently as described in step (2.1) in Maude-NPA. The size of each partition may affect the running performance of the tool. Because if each partition

Algorithm 9: Conducting the implication in step (2.1) for states at each layer in parallel.

```
1  function implicationStep() is
2      if status != implication then
3          return;
4      nW ← neededWorkersForSim(size(next), N, simBatch);
5      if nW = 1 then
           /* step (2.1) is conducted in sequence                              */
6          status ← implicationH;
7          next ← simplifyByImplicationL(next);
8          return;
       /* step (2.1) is conducted in parallel                                  */
9      (unusedW, usedW) ← getWorkers(workers, nW);
10     workers ← unusedW;
11     sendJobs(usedW, implication, next);
12     next ← empty;
13     while True do
14         receivingData();
15         while not isEmpty(workers) and size(simJobs) > 1 do
16             worker ← dequeue(workers);
17             IST1 ← dequeue(simJobs);
18             IST2 ← dequeue(simJobs);
19             send(worker, combination, IST1, IST2);
20         if size(simJobs) = 1 and size(workers) = N then
21             status ← implicationH;
22             next ← dequeue(simJobs);
23             return;
```

contains a small number of states (a light job), a worker may finish its task quickly and so the communication cost may be larger than the benefit able to be gained from parallelization. Hence, we use `simBatch` denoting the minimum number of states in each partition that should be assigned to a worker. Based on the number of successor states in `next`, the number of workers N, and `simBatch`, we calculate the number of needed workers for parallelization as follows: each selected worker has a job such that the number of states is at least `simBatch` states. Note that `simBatch` is set to 20 as default and the number of needed workers is a positive natural number and less than or equal to N because we only have N workers available.

We then call the `neededWorkersForSim` function at line 4 to calculate the number of needed workers to conduct step (2.1) in parallel. The master checks whether the number of needed workers is equal to one at line 5. If so, we do not need to parallelize step (2.1) and conduct it in sequence in the code fragment at lines 6–8. We first set `status` to *implicationH* at line 6 and call the `simplifyByImplicationL` function with `next` as its parameter at line 7. The function is defined in Maude-NPA to conduct step (2.1) in sequence. `next` is then updated to the result of the function. Lastly, we exit the function at line 8 to move to conduct step (2.2). If the number of needed workers is greater than one, we conduct step (2.1) in parallel in the code fragment at lines 9–22. We first pick up `nW` workers from `workers` by calling the `getWorkers` function at line 9. `unusedW` and `usedW` are assigned the workers that are not used and the ones that are used for parallelization, respectively. Note that there may be some workers that are not used depending on the number of successor states because of the use of `simBatch`. We then assign `unusedW` to `workers` at line 10, while we use `usedW` and `next` to start parallelizing step (2.1) by calling the `sendJobs` function at line 11. The function divides the successor states in `next` evenly to each worker in `usedW`. Each divided partition is assigned to a worker as a job whose type is *implication*. We then set `next` to the empty set at line 12. As soon as a worker completes a job assigned to it by the master, it returns a set of simplified states as a result and sends the result to the master. In the code fragment at lines 13–23, we call the `receivingData` function at line 14 to receive such results as (simplified) jobs and store them in `simJobs`.

As described above with Fig. 7.5, for every two jobs in `simJob`, we combine the two sets of states into one set of states such that no state is implied by other states in the set. The code fragment at lines 15–19 checks continuously whether `workers` is not empty and `simJobs` contains more than one job. If that is the case, the master dequeues `workers` and `simJobs` to obtain a worker identifier and two jobs, constructs a job whose type is *combination*, and sends it to the worker at line 19. As soon as the worker receives the job, it combines the two sets of states into one set and sends it to the master. The master will store it into `simJob` as a (simplified) job so that it can be combined with another job subsequently. The code fragment at lines 20–23 checks whether `simJobs` contains only one job and no jobs are being processed by workers. If that is the case, the master changes its status to *implicationH* and dequeues `simJobs` to obtain the set of successor states and assign it to `next` at lines 21 and 22, respectively. Lastly, the master exits the function at line 23 to move to conduct step (2.2). At this point, we have completed step (2.1) in parallel and the successor states are currently stored in `next`.

Algorithm 10 shows the pseudo-code of the `implicationWithHistoryStep` function in which we conduct the implication with history states in step (2.2) in parallel for the successor states obtained from step (2.1) (see Fig. 7.6 as well). The master first checks whether `status` is not *implicationH* at line 2. If so, it is not the case that we need to conduct step (2.2) here. The master exits the function immediately. Otherwise, the status of the master is currently *implicationH* and the successor states are currently stored in `next`. As described

Algorithm 10: Conducting the implication in step (2.2) for states with history states at each layer in parallel.

```
1  function implicationWithHistoryStep() is
2      if status != implicationH then
3          return;
4      nW ← neededWorkersForSimH(size(history), size(next), N, simBatchH);
5      if nW = 1 then
           /* step (2.2) is conducted in sequence                          */
6          next ← simplifyByImplicationH(history, next);
7          status ← filter;
8          return;
       /* step (2.2) is conducted in parallel                              */
9      (unusedW, usedW) ← getWorkers(workers, nW);
10     workers ← unusedW;
11     sendJobs(usedW, implicationH, history, next);
12     next ← empty;
13     while True do
14         receivingData();
15         if size(workers) = N then
16             status ← filter;
17             return;
```

above with Fig. 7.6, we divide the successor states into multiple partitions so that the implication for each partition with history states can be conducted independently as described in step (2.2) in Maude-NPA. Note that in Fig. 7.6 we divide the set of successor states in the last job in `simJobs` on the fly instead of assigning it to `next` and then dividing the set of successor states in `next` for simplicity. The size of each partition and the number of history states may affect the running performance of the tool. Because if each partition and history states contain a small number of states (a light job), a worker may finish its task quickly and so the communication cost may be larger than the benefit able to be gained from parallelization. Hence, we use `simBatchH` that denotes the minimum number of the multiplication of the number of history states and the number of states in each partition that should be assigned to a worker. Based on the number of successor states in `next`, the number of history states in `history`, the number of workers N, and `simBatchH`, we calculate the number of needed workers to conduct step (2.2) in parallel as follows: each selected worker has a job such that the multiplication of the number of states in its partition and the number of history states is at least `simBatchH`.

Note that `simBatchH` is set to 50 as default and the number of needed workers is a positive natural number and less than or equal to N because we only have N workers available.

We then call the `neededWorkersForSimH` function to calculate the number of needed workers to conduct step (2.2) in parallel at line 4. The master checks whether the number of needed workers is equal to one at line 5. If that is the case, we do not need to parallelize step (2.2) and conduct it in sequence by calling the `simplifyByImplicationH` function with two parameters: `history` and `next` at line 6. The function is defined in Maude-NPA to conduct step (2.2) in sequence. The result of the function is assigned to `next`. The master then changes its status to *filter* and exits the function to move to conduct step (3) at lines 7 and 8, respectively. If the number of needed workers is greater than one, we conduct step (2.2) in parallel in the code fragment at lines 9–17. We first pick up `nW` workers from `workers` by calling the `getWorkers` function at line 9. Note that there may be some unused workers depending on the number of successors and the number of history states because of the use of `simBatchH`. `unusedW` and `usedW` are assigned the workers that are not used and the ones that are used for parallelization, respectively. We then assign `unusedW` to `workers` at line 10, while we use `usedW`, `next`, and `history` to conduct step (2.2) in parallel by calling the `sendJobs` function at line 11. The function divides the successor states in `next` evenly to each worker in `usedW`. Each divided partition and history states are assigned to a worker as a job whose type is *implicationH*. We then set `next` to the empty set at line 12. As soon as a worker completes a job assigned to it by the master, it returns a set of simplified states as a result and sends the result to the master. In the code fragment at lines 13–17, we call the `receivingData` function at line 14 to receive such results and store them in `next`. The code fragment at lines 15–17 checks whether no jobs are being processed by workers. If that is the case, the master changes its status to *filter* and exits the function to move to conduct step (3) at lines 16 and 17, respectively. At this point, we have completed step (2.2) and the successor states are currently stored in `next`.

Algorithm 11 shows the pseudo-code of the `filterStep` function in which we conduct step (3) in sequence at the master. The master first checks whether `status` is not *filter* at line 2. If so, it is not the case that we need to conduct step (3) here. The master exits the function immediately. Otherwise, the status of the master is currently *filter* and the successor states are currently stored in `next`. The master first filters state duplications from `next` with history states and rules out initial states from `next` as counterexamples by calling the `filterWithHistoryAndInit` function at line 4. The initial states and the final successor states of the layer being concerned are assigned to `INIT` and `IST`, respectively. The master then updates `jobs`, `next`, and `BStep` to `IST`, empty, and `BStep`−1 at line 5 for the next layer, respectively. Note that if `BStep` is unbounded, then `BStep`−1 is still unbounded. `IST` is then added to `history` to mark as visited states at line 6. The master changes its status to *narrowing* at line 7 to ready for the next layer. Lastly, the master checks if `INIT` is not empty, `BStep` is equal to 0, or `jobs` is empty. If that is the case, the master assigns `INIT` to `initialStates` and changes its status to *stop* for termination. If there are any counterexamples, they will be stored

Algorithm 11: Filtering state duplications and ruling out initial states at each layer in step (3) in sequence.

```
1  function filterStep() is
2      if status != filter then
3          return;
4      (INIT, IST) ← filterWithHistoryAndInit(M, history, next);
5      (jobs, next, BStep) ← (IST, empty, BStep − 1);
6      history ← history ∪ IST;
7      status ← narrowing;
8      if not isEmpty(INIT) or BStep = 0 or isEmpty(jobs) then
9          initStates ← INIT;
10         status ← stop;
```

in `initialState`. At this point, we have completed step (3) and move to check for termination at lines 11–13 in Algorithm 7.

7.4.3 Job Handing by Workers

Algorithm 12 shows the pseudo-code for job handling by workers. Each worker maintains some shared information, such as `M`, `GS`, and `F`. Whenever a worker is requested to handle a job, the worker receives `DATA` from the master. The kind of the job is *narrowing*, *implication*, *combination*, or *implicationH*. The code fragment at lines 3–5 checks whether the kind of the job is *narrowing*. If that is the case, the worker deconstructs `DATA` to obtain the state `IS` at line 3. Given `M`, `GS`, `F`, and `IS`, the `nextBackNarrowForParallel` function performs the backward narrowing just by one step to obtain its successor states `IST` reachable from `IS` at line 4. `IST` is then sent to the master as a bunch of jobs at line 5. The code fragment at lines 6–8 checks whether the kind of the job is *implication*. If that is the case, the worker deconstructs `DATA` to obtain the set of states `IST` at line 6. Given `IST`, the `simplifyByImplicationL` function performs the implication in step (2.1) for the set of states `IST` at line 7. The function returns a set of simplified states, which is assigned to `IST'`, such that no state is implied by other states in the set. `IST'` is then sent to the master as a simplified job at line 8. The code fragment at lines 9–11 checks whether the kind of the job is *combination*. If that is the case, the worker deconstructs `DATA` to obtain two sets of states `IST1` and `IST2` at line 9. Note that each set of states has been simplified before. Given `IST1` and `IST2`, the `combineSimplifyByImplicationL` function combines the two sets in step (2.1) with a slightly different implication. Basically, we do not need to check the implication for the states in each set. For each state in the set `IST1`, if the state is implied by a state in the set `IST2`, then the state is removed from the set `IST1`. For

Algorithm 12: Job handling by workers.

```
input : M – the module used to conduct the backward narrowing
        GS – the grammars generated from the protocol under verification
        F – a filter specified from users
1  while isOpen() do
2      if DATA ← recv(master) then
3          if (narrowing, IS) ← DATA) then
4              IST ← nextBackNarrowForParallel(M, GS, F, IS);
5              send(master, IST);
6          else if (implication, IST) ← DATA then
7              IST′ ← simplifyByImplicationL(IST);
8              send(master, IST′);
9          else if (combination, IST1, IST2) ← DATA then
10             IST ← combineSimplifyByImplicationL(IST1, IST2);
11             send(master, IST);
12         else if (implicationH, History, IST) ← DATA then
13             IST′ ← simplifyByImplicationH(History, IST);
14             send(master, IST′);
```

each state in the set `IST2`, if the state is implied by a state in the set `IST1` that has been checked, then the state is removed from the set `IST2`. Thereby, we can obtain a set of states such that no state can be implied by other states in the set. The result of the function is assigned to `IST`, which is then sent to the master as a simplified job at line 11. The code fragment at lines 12–14 checks whether the kind of the job is *implicationH*. If that is the case, the worker deconstructs `DATA` to obtain the history states `history` and the set of states `IST` at line 12. Given `history` and `IST`, the `simplifyByImplicationH` function performs the implication in step (2.2) at line 13. The result of the function is a set of simplified states, which is assigned to `IST'`, such that no state is implied by any state in the history states. `IST'` is then sent to the master as a bunch of jobs at line 14. Note that workers terminate if and only if the master closes all connections. The `nextBackNarrowForParallel` and `combineSimplifyByImplicationL` functions are built based on existing functions in Maude-NPA, while the `simplifyByImplicationL` and `simplifyByImplicationH` functions are defined in Maude-NPA to conduct step (2.1) and step (2.2) in sequence, respectively.

7.5 Experiments

We have used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory to conduct experiments. We use Maude-NPA and multiple parallel versions of Maude-NPA including Par-Maude-NPA-1 and Par-Maude-NPA-2 in our case studies. The tool and the case studies for the experiments are publicly available at the webpage in Footnote 5. Besides, the original source code of the case studies and more protocols are listed at the webpage.[8]

We have conducted experiments on various kinds of protocols to confirm the usefulness of Par-Maude-NPA-2 such as Symmetric Key Protocols, Homomorphism Protocols, Exclusive OR Protocols, API Protocols, PKCS Protocols, Choice Protocols, and Distance-Bounding Protocols. The experimental data are shown in Tables 7.1–7.2. The first, second, and third columns denote the name of the protocols, the attack state id used in protocol specifications, and the depth bound, respectively. The fourth and fifth columns denote the verification time excluding the time taken to generate the grammars for protocols when conducting formal verification/-analysis with Maude-NPA and Par-Maude-NPA-2, respectively. In a row, the bold value is either in the fourth column or the fifth column denoting the corresponding winning tool. The sixth column denotes the percentage of improvement when using Par-Maude-NPA-2. If the value is a positive number, namely X, it means that the Par-Maude-NPA-2 is $X\%$ faster than Maude-NPA. Conversely, if the value is a negative number, namely $-X$, it means that Maude-NPA is $X\%$ faster than Par-Maude-NPA-2. The last column denotes the average number of states at each layer for each worker to handle, respectively. Furthermore, we inspect the number of states located at each layer for each protocol shown in a document publicly available at Footnote 5, which resides under the *documents* folder. Formal verification experiments terminate as soon as initial states (counterexamples) are found, the depth bound is reached, or no states are found for the next layer.

Our previous tool, Par-Maude-NPA-1, supports a parallel version of Maude-NPA [147] that uses Maude sockets to communicate the master and workers and so we can flexibly choose to use a shared-memory machine or a distributed environment. Meanwhile, the tool in this work, Par-Maude-NPA-2, supports a new parallel version of Maude-NPA that uses meta-interpreters and so we can use only a shared-memory machine. For all experiments in Tables 7.1–7.2, we use a master and eight workers with a shared-memory machine, the MacPro computer. Notice that we conduct the experiments with Par-Maude-NPA-2 that are the same as what we have done with Par-Maude-NPA-1 in [147]. The experimental data say that for simple case studies (25 experiments) whose verification time is less than 40 seconds, Maude-NPA is faster than Par-Maude-NPA-2 because the number of states located at each layer is very small and the verification time is so short that the communication cost between the master and workers as well as the cost to prepare workers and load many modules into each worker become

[8] `http://personales.upv.es/sanesro/Maude-NPA_Protocols/index.html`

Table 7.1: Experimental results of Maude-NPA and Par-Maude-NPA-2.

Protocol	**Attack State**	**Bound**	**Maude-NPA (seconds)**	**Par-Maude-NPA-2 (seconds)**	**P(%)**	**States/ Layer/ Worker**
1. Symmetric Key Protocols						
Amended Needham Schroeder	0	7	4588.61	**1967.806**	57	11.11
Carlsen Secret Key Initiator	0	5	224.175	**115.507**	48	3.73
Denning Sacco	0	11	35.243	**30.269**	14	0.43
Diffie Hellman	0	11	284.211	**113.55**	60	1.56
	1	12	286.663	**104.365**	64	1.45
	2	13	35.371	**23.073**	35	0.32
ISO-5 Pass Authentication	0	5	101.649	**54.764**	46	2.1
Kao-Chow RA	0	4	52.235	**28.643**	45	2
Kao-Chow RAHK	0	4	**4.027**	14.42	-72	0.19
Kao-Chow RAT	0	4	114.414	**67.203**	41	1.94
Otway-Rees	0	4	72.516	**42.553**	41	2.16
Secret 06	0	2	**1.732**	6.887	-75	0.38
Secret 07	0	4	**2.589**	9.154	-72	0.28
Wide Mouthed Frog	0	3	16.11	**15.009**	7	1.92
Woo and Lam Authentication	0	4	**1.371**	8.3	-83	0.28
Yahalom	0	4	45.216	**27.557**	39	1.91
2. Homomorphism Protocols						
Needham Schroeder Lowe ECB	0	7	73.869	**44.835**	39	1.11
3. Exclusive OR Protocols						
Needham Schroeder Lowe XOR	0	8	**10.22**	13.856	-26	0.31
SK3	0	3	**4.162**	12.037	-65	0.17
TMN ltv-F-tmn-asy	0	5	157.442	**37.72**	76	0.88
WIRED ltv-C-wep-asy	0	5	**14.392**	20.574	-30	0.15
WIRED ltv-C-wep-variant	0	5	**15.571**	23.035	-32	0.15
4. API Protocols						
YubiKey	0	9	**3.487**	12.657	-72	0.17
	1	7	93824.875	**28988.722**	69	5.13
	21	8	341.529	**134.66**	61	0.8
	3	7	13092.864	**4225.64**	68	3
YubiHSM attack(d)	0	9	843.388	**404.978**	52	2.38

burdensome. However, Par-Maude-NPA-2 still can finish in a reasonably short amount of time. As described above we use `simBatch` and `simBatchH` to decide whether we should conduct step (2.1) and step (2.2) in parallel, respectively. For simple case studies, we do not need to conduct step (2.1) and step (2.2) in parallel because the number of successor states and history states at each layer is small. For non-trivial case studies (34 experiments) in which the number of

Table 7.2: Experimental results of Maude-NPA and Par-Maude-NPA-2.

Protocol	Attack State	Bound	Maude-NPA (seconds)	Par-Maude-NPA-2 (seconds)	P(%)	States/ Layer/ Worker
5. PKCS Protocols						
PKCS11 a1-noComp	0	4	24.815	**23.551**	5	0.81
PKCS11 a2-noComp	0	6	69.532	**44.621**	36	0.75
PKCS11 a3-noComp	0	6	296.424	**164.974**	44	1.6
PKCS11 a4-noComp	0	7	62.886	**39.183**	38	0.88
PKCS11 a5-noComp	0	9	382.498	**225.497**	41	1.82
6. Choice Protocols						
encryption mode	0	4	**3.164**	10.715	-70	0.28
	1	4	**8.587**	11.811	-27	0.78
	2	10	67.941	**39.871**	41	1.1
	3	11	136.958	**56.155**	59	1.61
rock paper scissors	0	9	125.615	**52.584**	58	1.81
	1	1	**0.389**	5.305	-93	0.13
	2	2	**1.003**	6.574	-85	0.38
TLS regular	0	3	**6.727**	13.827	-51	0.17
TLS attack	0	11	8695.211	**3150.536**	64	3.15
7. Distance-Bounding Protocols						
brands chaum	1	4	**6.236**	11.817	-47	0.25
	2	6	**16.186**	17.121	-5	0.29
CRCS	1	9	766.746	**291.909**	62	0.75
	2	8	121.77	**83.297**	32	0.42
H&K	1	5	16.792	**15.401**	8	0.35
	2	2	**1.174**	7.136	-84	0.13
MAD	1	9	175.382	**96.522**	45	0.67
	2	6	967.156	**396.371**	59	2.42
Meadows v1-DH	1	4	**1.646**	9.092	-82	0.13
	2	8	**32.153**	32.836	-2	0.28
Meadows v2-DH	1	4	**1.694**	9.08	-81	0.13
	2	3	**2.464**	8.884	-72	0.17
Munilla	1	7	186.24	**66.813**	64	1.45
	2	4	**6.279**	12.942	-51	0.19
Swiss Knife	1	4	**6.69**	12.201	-45	0.25
	2	4	26.862	**25.465**	5	0.38
TREAD	1	4	**6.444**	12.205	-47	0.25
	2	4	**5.166**	11.785	-56	0.25

Table 7.3: The effectiveness of the use of meta-interpreters and the parallelization of step (2.1) and step (2.2).

Protocol	Attack State	Bound	Maude-NPA (seconds)	Par-Maude-NPA-1 (seconds)	P1(%)	Par-Maude-NPA-1' (seconds)	P1'(%)	Par-Maude-NPA-2 (seconds)	P2(%)
Amended NS	0	7	4588.61	**2821.933**	39	**2615.698**	43	**1967.806**	57
YubiKey	1	7	93824.875	**65294.633**	30	**59173.668**	37	**28988.722**	69
TLS attack	0	11	8695.211	**6997.392**	20	**4366.156**	50	**3150.536**	64

states located at each layer is larger, Par-Maude-NPA-2 has a very good performance that is 44% faster than Maude-NPA on average, demonstrating its potential. Among the non-trivial case studies, there are some case studies for which we may not need to conduct step (2.1) and step (2.2) in parallel at some layers, where the number of states and history states at each layer is small, because of the use of `simBatch` and `simBatchH`. For the three protocols Amended Needham Schroeder, YubiKey, and TLS Attack whose verification time is the largest among all protocols, Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA by 57%, 69%, and 65%, respectively. Meanwhile, Par-Maude-NPA-1 can only improve the running performance of Maude-NPA by 39%, 30%, and 20%, respectively [147], which demonstrates that Par-Maude-NPA-2 can largely improve the running performance of Par-Maude-NPA-1 as well as Maude-NPA. This is because the tool uses meta-interpreters instead of Maude sockets and parallelizes not only the backward narrowing in step (1) but also the transition subsumption in step (2.1) and step (2.2) in Maude-NPA. For step (1), the average number of states at each layer for a worker is measured to let us know how busy each worker is, which reflects the number of states located at each layer. The busier workers are and the deeper the depth bound is, the more benefit we may gain from the use of parallelization for step (1). For step (2.1) and step (2.2), the more successor states and history states at each layer, the more benefit we may gain from the use of parallelization also.

In addition, we would like to demonstrate the effectiveness of the use of meta-interpreters and the parallelization of the transition subsumption in step (2.1) and step (2.2). We use another version of Par-Maude-NPA-2, called Par-Maude-NPA-1', in which only the backward narrowing in step (1) is parallelized while the transition subsumption in step (2.1) and step (2.2) are performed in sequence by setting `simBatch` and `simBatchH` to unbounded. Par-Maude-NPA-1' can be regarded as another version of Par-Maude-NPA-1 in which meta-interpreters are used instead of Maude sockets. We conduct experiments for the three protocols whose verification time is the largest among all protocols because the number of states at each layer of the three protocols is large and so we can see the difference between Par-Maude-NPA-1, Par-Maude-NPA-1', and Par-Maude-NPA-2 in the running performance. We use one master and eight workers with the MacPro machine to conduct experiments. The experimental results

Table 7.4: Par-Maude-NPA-1 and Par-Maude-NPA-2 with various numbers of workers.

Protocol	**Attack State**	**Bound**	**Maude-NPA (seconds)**	**#Workers**	**Par-Maude-NPA-1 (seconds)**	**P1(%)**	**Par-Maude-NPA-2 (seconds)**	**P2(%)**	**States/ Layer/ Worker**
Amended NS	0	7	4588.61	8	**2821.933**	39	**1967.806**	57	11.11
				16	**2651.974**	42	**1787.24**	61	5.55
				24	**2646.479**	42	**1790.715**	61	3.7
YubiKey	1	7	93824.875	8	**65294.633**	30	**28988.722**	69	5.13
				16	**61365.349**	35	**24756.166**	74	2.56
				24	**60937.287**	35	**22088.416**	76	1.71
TLS attack	0	11	8695.211	8	**6997.392**	20	**3150.536**	64	3.15
				16	**6960.781**	20	**2393.488**	72	1.57
				24	**7193.625**	17	**2392.824**	72	1.05

are shown in Table 7.3. Note that Amended NS denotes the Amended Needham Schroeder protocol. The sixth, eighth, and tenth columns denote the percentage of improvement for Par-Maude-NPA-1, Par-Maude-NPA-1', and Par-Maude-NPA-2 compared to Maude-NPA, respectively. For the three protocols, Par-Maude-NPA-1' can improve the running performance of Maude-NPA from 39% to 43% (4% increased), 30% to 37% (7% increased), 20% to 50% (30% increased) compared to Par-Maude-NPA-1, respectively, demonstrating that the use of meta-interpreters instead of Maude sockets is effective. Meanwhile, Par-Maude-NPA-2 can improve the running performance of Maude-NPA from 43% to 57% (14% increased), 37% to 69% (32% increased), 50% to 64% (14% increased) for the three protocols compared to Par-Maude-NPA-1', respectively, demonstrating that the parallelization of the transition subsumption in step (2.1) and step (2.2) is effective. Therefore, Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA for the three protocols because of the use of meta-interpreters and parallelization of step (1), step (2.1), and step (2.2).

Furthermore, we would like to demonstrate the power of Par-Maude-NPA-2 further by conducting more experiments with various numbers of workers for the three protocols whose verification time is the largest among all protocols. The experimental data are shown in Table 7.4. The fifth column denotes the number of workers used in the experiments. The sixth and eighth columns denote the verification time for Par-Maude-NPA-1 and Par-Maude-NPA-2, respectively. The seventh and ninth columns denote the percentage of improvement for Par-Maude-NPA-1 and Par-Maude-NPA-2 compared to Maude-NPA, respectively. We can see that Par-Maude-NPA-2 can largely improve the running performance of Maude-NPA from 39% to 57% (18% increased), 30% to 69% (39% increased), and 20% to 64% (44% increased) for the three protocols compared to Par-Maude-NPA-1, respectively, when eight workers are used. When we increase the number of workers, the tool improves the running performance of Maude-NPA further, such as 61%, 76%, and 72% for the three protocols, respectively, when the number

of workers is 24. We can see that the average number of states at each layer for a worker is subject to the number of workers used in the experiments. When the average number of states at each layer for a worker is high, we may increase the number of workers to improve the running performance of the tool. However, up to a certain point, the more workers used, the less busy workers are and the more burden the master needs to handle and communicate with workers, making the running performance not improve. For example, when the number of workers is increased from 16 to 24, the running performance does not improve and even becomes a bit worse for the first and third case studies. In addition, we use `simBatch` and `simBatchH` to calculate the number of needed workers to conduct step (2.1) and step (2.2) in parallel at each layer, respectively. At the very first layers, the number of successor states and history states is small. Therefore, we may not need to conduct step (2.1) and step (2.2) in parallel. For deeper layers, when the number of successor states and history states is larger, step (2.1) and step (2.2) may be conducted in parallel. When we use `simBatch` and `simBatchH`, there is a case in which the number of needed workers is less than the number of workers available. Hence, even if we increase the number of workers, it does not contribute to the running performance. Furthermore, when we parallelize step (2.1), we need to generate (simplified) jobs and then combine every two jobs into one job until there is only one job left. Whenever we combine two (simplified) jobs produced by two workers, we only need one worker to conduct the combination while the other worker is free. Therefore, all workers do not always work at any time. Up to a certain point, even if the number of workers is increased, the running performance does not improve.

As described above we use *simBatch* and *simBatchH* to avoid assigning a light job to a worker. In all experiments, we set *simBatch* and *simBatchH* to 20 and 50 as default. The use of *simBatch* and *simBatchH* is applicable when the number of states and history states at a layer is reasonably small. When the number of states and history states at a layer is large, the states at a layer should be divided evenly into each worker to take advantage of parallelization as much as possible. From our experiences when conducting experiments for those protocols with the tool, we set *simBatch* and *simBatchH* to 20 and 50, respectively. For example, we need to conduct the transition subsumption in step (2.1) for 20 states. In the worst case, no state is implied by other states in 20 states, Maude-NPA needs to spend about 380 (i.g. $2 \times 19 \times 20 \div 2$) computations in which each computation is to check the implication between two states. Hence, 20 states may be the minimum number of states that should be assigned to a worker. Likewise, *simBatchH* is used in the transition subsumption with history states in step (2.2). There is a trade-off between increasing and decreasing those numbers because it affects the number of jobs as well as the number of workers used at each layer. However, we should use reasonably small values as our default values because we are concerning the minimum number of states that should be assigned to a worker. Nevertheless, we need to conduct more experiments with different values for *simBatch* and *simBatchH* from which we may select good values for *simBatch* and *simBatchH* with our tool as one piece of our future work.

In summary, Par-Maude-NPA-2 can improve the running performance of Maude-NPA effectively when dealing with all non-trivial case studies in which the number of states located at each layer is considerably large. For the backward narrowing in step (1), the more states located at each layer and the deeper the search space is, the more improvement may be obtained by parallelization. For the transition subsumption in step (2.1) and step (2.2), the more successor states and history states at each layer, the more improvement also may be obtained by parallelization. For simple case studies, whose verification time is very small, for example, less than 40 seconds in our case studies, we do not need to use Par-Maude-NPA-2, although we still can use it to obtain a result in a reasonably short amount of time. We can see that the verification time for simple case studies is very small and so the use of Par-Maude-NPA-2 is not much different compared to Maude-NPA in terms of verification time. Hence, it is sufficient to use solely Par-Maude-NPA-2 for analyzing any cryptographic protocols even though they are simple.

7.6 Summary

We have described a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption are conducted in parallel. A tool has been developed in Maude to support the parallel version with a master-worker model by using meta-interpreters instead of Maude sockets as in our previous work. We have also reported on some experiments of various kinds of protocols in which the tool can increase the running performance of both Maude-NPA (about 44%) and our previous tool (about 23 %) on average for all non-trivial case studies where one master and eight workers are used. Especially, the tool can largely improve the running performance of Maude-NPA by 57%, 69%, and 65% for the three case studies whose verification time is the largest among all protocols, respectively, demonstrating its potential to deal with case studies whose state space is large. As one piece of our future work, we should conduct more case studies and use various numbers of workers with the tool to demonstrate its usefulness. Last but not least, the basic techniques used to parallelize Maude-NPA are not necessarily specific to Maude-NPA, but could be used to parallelize other tools dedicated to cryptographic protocol analysis, especially Tamarin, because Tamarin is the closest to Maude-NPA.

Regarding the correctness of our parallel version of Maude-NPA, we only parallelize what to do at each layer that consists of (1) backward narrowing by only one step and (2) some optimization. For (1), because there are multiple symbolic states, we can conduct backward narrowing by only one step in parallel. For (2), what to do is basically to reduce the number of states. States concerned are divided into multiple collections that can be tackled in parallel. Hence, parallelization used for Maude-NPA is conservative, respecting what is done by the sequential version of Maude-NPA.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis has proposed some techniques to mitigate the state space explosion problem and improve the running performance of model checking to some extent by parallelization to make the best use of multi-core architectures. We have demonstrated the proposed techniques with three non-trivial cases: (1) parallelization of Java Pathfinder with the specification-based testing technique for concurrent programs, (2) parallelization of Maude LTL model checker with the $L+1$-layer divide & conquer approach to leads-to model checking, and (3) parallelization of Maude-NPA for cryptographic protocol analysis.

For testing concurrent programs in (1), we have proposed the specification-based testing technique for concurrent programs in a stratified way. The proposed technique could be processed naturally in parallel, which has been utilized by the tool supporting the technique. The tool has been implemented in Java with a master-worker model, where JPF and Maude are used as backends to generate state sequences from a program and check such state sequences with a specification, respectively. We have conducted some experiments to demonstrate that the proposed technique can mitigate the state space explosion problem and largely improve the timing performance for testing for all cases except for one case, which cannot be achieved with the straightforward use of only one JPF instance. Besides, we have described how to extend the technique to test concurrent Java programs without checking if execution sequences generated from Java programs can be accepted by Maude specifications. We have made some progress toward making it possible to detect a violated state located at a deep depth. Last but not least, the specification-based testing technique can be used to complement the very last step in correct-by-construction software development.

For LTL model checking in (2), we have described a parallel version of $L+1$-DCA2L2MC, which is dedicated to leads-to model checking. Leads-to properties are often used to express systems requirements in model checking and can also express safety properties (invariant properties). Hence, it is worth focusing on leads-to model checking. A tool supporting the parallel

version has been implemented in Maude with a master-worker model by using Maude sockets. We have conducted experiments to demonstrate that the tool can mitigate the state space explosion problem and largely improve the running performance of model checking for all case studies except for one simple mutual exclusion protocol, namely test&set. Counterexample generation is one main task in the tool that can be optimized to improve the running performance of the tool. We have proposed a technique to generate all counterexamples at once in model checking based on the Tarjan algorithm with a new model checker to support the technique. The new model checker is implemented in C++ and is mostly based on facilities existing in the Maude LTL model checker. We have conducted some experiments to demonstrate the power of the technique. Furthermore, layer configuration selection has been addressed with an approach to finding good layer configurations for $L+1$-DCA2L2MC technique, and an analysis tool that supports the approach. The analysis tool has been implemented in Maude with a client-server model by using meta-interpreters. We have conducted some experiments to demonstrate the usefulness of the analysis tool as well as the approach for layer configuration selection.

For cryptographic protocol analysis, it is extremely important to make cryptographic protocols secure, safe, and reliable over an open network, such as the Internet. We have described a parallel version of Maude-NPA in which the backward narrowing and the transition subsumption are conducted in parallel at each layer. A tool supporting the technique has been developed in Maude with a master-worker model by using meta-interpreters instead of Maude sockets. We have reported on some experiments of various kinds of protocols in which the tool could increase the running performance of Maude-NPA by 44% on average for all non-trivial case studies experimented where one master and eight workers are used. Especially, the tool can largely improve the running performance of Maude-NPA by 57%, 69%, and 65% for the three case studies whose verification time is the largest among all protocols used for the experiments, demonstrating its potential to deal with case studies whose state space is huge.

In summary, with rapidly expanding software systems as well as their complexity nowadays, verifying software systems for their desired properties is indispensable to make them secure and reliable. Model checking has been proven to be a tremendously successful technique to verify requirements for a variety of systems. However, the most challenge in model checking is the state space explosion problem. Another fundamental challenge is to improve the running performance of model checking because it needs to verify the entire reachable state space. Recently, high-performance computing technologies, such as multicore processors and clusters have emerged. Hence, speeding up model checking with parallelization is one promising approach. This thesis has given an effort to parallelize the three cases: (1), (2), and (3) to mitigate the two challenges to some extent. We expect that this thesis will benefit researchers as well as engineers in both academia and industry when they use model checking to verify software systems. They can speed up their process of designing and verifying software systems by using parallel model checking techniques presented in this thesis.

8.2 Future Work

Although three parallel versions (1), (2), and (3) present a good performance and improve the running performance of model checking, we consider several lines of future work beside some pieces of future work mentioned in each chapter. As described in Chapter 4, some techniques are shared between (1), (2), and (3), meaning that the techniques are generic enough and we can apply our techniques to parallelization of other formal verification tools. Hence, we would parallelize some other tools used for formal methods, which are described in this section.

Real-Time Maude (RT-Maude) [151] is a language and tool supporting the formal specification and analysis of real-time and hybrid systems where time information is taken into account. RT-Maude is implemented in Maude [13] as an extension of Full Maude [152]. RT-Maude specifications are executable and simulate the progress in systems under analysis by *timed rewriting*. *Tick rules* are used to formalize the time advance in systems in which the time increment is given in the form of either a concrete value or a variable. The former is called *time-deterministic* and is used in discrete time domains, while the latter is called *time-nondeterministic* because the time increment is an arbitrary number that will be decided based on the *time sampling strategy* (time mode) specified by users among some time sampling strategies supported by RT-Maude, and is used in dense time domains. RT-Maude supports time-bounded linear temporal logic model checking that can analyze all behaviors of a system from a given initial state up to a certain duration. By restricting model checking up to a certain duration, the set of reachable states is a finite set so that model checking experiments can be carried out. RT-Maude usually uses some properties specified in LTL to express systems requirements among which until and until stable properties are used [151, 153]. Besides, RT-Maude supports dedicated commands to check for until and until stable properties, meaning that until and until stable properties are interesting properties in real-time systems. Furthermore, the reachable state space of a real-time system is often huge because the behavior of the system changes over time. Therefore, it is worth focusing on mitigating the state space explosion in until and until stable model checking. We describe a divide & conquer approach to "Until" and "Until-stable" model checking in [154]. One piece of our future work is that we will implement a tool to support the approach in Maude both in sequential and parallel versions. Besides, we would integrate the parallel version of $L+1$-DCA2L2MC to RT-Maude so that we can check leads-to properties with RT-Maude in parallel.

CafeOBJ [155] is a language for writing formal specifications for a variety of software and hardware systems and for verifying their properties. One possible way to verify properties in CafeOBJ is by writing proof scores [156] that consist of reduction by means of equations, of goal-oriented terms in user-defined modules (the so-called open-close environments, which allow extension of previous theories by adding new operators and equations) and checking whether they are reduced to the expected value (usually true). Writing proof scores is flexible when we can write proofs by induction and case splittings. However, it is error-prone because

users may overlook some missing cases in their proofs. CafeInMaude, a CafeOBJ interpreter implemented in Maude, with the CafeInMaude Proof Assistant (CiMPA) and the CafeInMaude Proof Generator (CiMPG) [157] have been proposed to infer formal proof scripts from CafeOBJ proof scores to ensure that the proof scores written to verify properties are correct. CiMPG can generate proof scripts from proof scores while CiMPA can execute the proof scripts to build a proof tree, which is proven when each Boolean value of all leaves of the proof tree is true. If the proof tree has been proven, we can conclude that the proof scores written to verify properties are correct. Generating proof scripts from proof scores is time-consuming when the size of proof scores is moderately large because CiMPA needs to take into account many cases as well as many open-close code fragments from proof scores. For the base case, a proof script fragment is generated, and from each induction case, a proof script fragment is generated. Generating those fragments is completely independent of each other and so they have reimplemented CafeInMaude in order to support parallel execution [158] in which those fragments are generated in parallel, where each fragment generation can be regarded as a task. However, some tasks take much more time than others, and then it is necessary to come up with a technique to identify and divide a heavy task into smaller sub-tasks so that many tasks can be handled in parallel. As a weak point, CiMPG was only able to find the complete CiMPA script when the proof score was correct; otherwise, CiMPG just pointed out where the error was and left it to users to handle. The CafeInMaude Proof Generator & Fixer-upper (CiMPG+F) has been proposed [159] and then extended [158] to mitigate the problem. CiMPG+F uses a bounded depth-first algorithm where it tries to check whether the current subgoal can be reduced to true by using induction hypotheses as premises or case splittings that are extracted from the current subgoal. They are applied and, for each one, recursion is used to try to discharge the subgoal in the new context up to a depth bound. If CiMPG+F could discharge the subgoal up to the bound, it returns a corresponding sequence of proof scripts; otherwise, a `:postpone` command is returned, meaning that it ignores the subgoal and leaves it to users. Because backtracking algorithms are very expensive, they have presented some heuristics as well as decision procedures for lists for improving the generation of case splittings. However, we can improve the running performance of CiMPG+F to some extent by parallelization because trying to use each case splitting to discharge the subgoal in the new context is completely independent. To this end, we would improve the current parallel version of CiMPG and parallelize CiMPG+F to improve the running performance for generating proof scripts from proof scores as one direction of our future work.

Manually writing proof scores is somewhat tedious. Hence, Invariant Proof Score Generation (IPSG) [160] has been proposed to generate proof scores for one invariant candidate. Given a collection of lemma candidates and one invariant candidate, IPSG tries to generate proof scores for the invariant candidate based on the lemma candidates by induction and case splittings. For the base case, a proof score fragment is generated, for each induction case, a proof score fragment is generated. Generating those fragments is independent of each other, which opens an

opportunity for parallelization. For the base case and each induction case, IPSG may consider case splittings to generate multiple proof score fragments from one proof core fragment being concerned. For example, in the beginning, fragment M represents a proof score fragment for an induction case. IPSG tries to reduce the invariant candidate in M and check whether the result of the reduction is either true or false. If it is true, IPSG takes M as a proof score fragment generated. If it is false, IPSG tries to select some lemmas from the collection of lemmas as premises for the invariant candidate and keeps on reducing until the reduction returns true or there are no more lemmas concerned. If so, the last fragment is used as a proof score fragment generated. If it is neither true nor false, IPSG tries to analyze it to obtain one equation for case splittings. Let us suppose that M is split into two fragments M1 and M2 by one equation in which the equation and the negation of the equation are added to M1 and M2, respectively. IPSG treats each M1 and M2 as M from the beginning. The cycle is repeated until all proof score fragments are generated. For each case splitting, IPSG splits the current proof score fragment into two fragments and each new fragment can be conducted independently. Trying to select and use some lemmas from the collection of lemmas as premies for the invariant candidate is independent as well. Case splittings and trying to select and use some lemmas while generating proof scores are time-consuming and can be conducted in parallel. Hence, we would parallelize IPSG to speed up the proof score generation as one piece of our future work.

Bibliography

[1] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles Antony Richard Hoare. *Structured programming*, volume 8 of *A.P.I.C. Studies in data processing*. Academic Press, 1972.

[2] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018, doi:10.1007/978-3-319-10575-8.

[3] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999, doi:10.1007/s100090050035.

[4] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994, doi:10.1145/186025.186051.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003, doi:10.1145/876638.876643.

[6] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008, doi:10.1016/j.tcs.2008.04.040.

[7] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992, doi:10.1145/136035.136043.

[8] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[9] Fuyuan Zhang, Yongwang Zhao, David Sanán, Yang Liu, Alwen Tiu, Shang-Wei Lin, and Jun Sun. Compositional reasoning for shared-variable concurrent programs. In *Formal Methods - 22nd International Symposium, FM 2018*, volume 10951 of *Lecture Notes in Computer Science*, pages 523–541, 2018, doi:10.1007/978-3-319-95582-7_31.

[10] 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. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 457–507. Springer, 2018, doi:10.1007/978-3-319-63516-3_12.

[11] Kyungmin Bae, Santiago Escobar, and José Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013*, volume 21 of *LIPIcs*, pages 81–96, 2013, doi:10.4230/LIPIcs.RTA.2013.81.

[12] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003, doi:10.1023/A:1022920129859.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007, doi:10.1007/978-3-540-71999-1.

[14] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[15] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2002, doi:10.1016/S1571-0661(05)82534-4.

[16] Santiago Escobar, Catherine A. Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50, 2007, doi:10.1007/978-3-642-03829-7_1.

[17] Vinay Arora, Rajesh Kumar Bhatia, and Maninder Singh. A systematic review of approaches for testing concurrent programs. *Concurrency and Computation: Practice and Experience*, 28(5):1572–1611, 2016, doi:10.1002/cpe.3711.

[18] Mohammadsadegh Dalvandi, Michael J. Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi. Verifiable code generation from scheduled Event-B models. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018*, volume 10817 of *Lecture Notes in Computer Science*, pages 234–248, 2018, doi:10.1007/978-3-319-91271-4_16.

[19] Victor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for Event-B. *International Journal on Software Tools for Technology Transfer*, 19(1):31–52, 2017, doi:10.1007/s10009-015-0381-2.

[20] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen, and Gary T. Leavens. Automated translation of VDM to JML-annotated Java. *International Journal on Software Tools for Technology Transfer*, 20(2):211–235, 2018, doi:10.1007/s10009-017-0448-3.

[21] Clifford B. Jones. *Systematic software development using VDM (2. ed.).* Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[22] J. C. P. Woodcock and Jim Davies. *Using Z - specification, refinement, and proof.* Prentice Hall international series in computer science. Prentice Hall, 1996.

[23] Egon Börger. The ASM refinement method. *Formal Aspects of Computing*, 15(2-3):237–257, 2003, doi:10.1007/s00165-003-0012-7.

[24] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996, doi:10.1017/CBO9780511624162.

[25] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010, doi:10.1007/s10009-010-0145-y.

[26] Ning Ge, Arnaud Dieumegard, Eric Jenn, and Laurent Voisin. Correct-by-construction specification to verified code. *Journal of Software: Evolution and Process*, 30(10), 2018, doi:10.1002/smr.1959.

[27] Canh Minh Do and Kazuhiro Ogata. Specification-based testing with simulation relations. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*, pages 107–146, 2019, doi:10.18293/SEKE2019-027.

[28] Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 3–18, 1995, doi:10.1007/978-0-387-34892-6_1.

[29] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, FM 1999*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, 1999, doi:10.1007/3-540-48119-2_16.

[30] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Information and Computation*, 98(2):142–170, 1992, doi:10.1016/0890-5401(92)90017-A.

[31] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification, 8th International Conference, CAV 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–427. Springer, 1996, doi:10.1007/3-540-61474-5_93.

[32] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS 1999, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, 1999, doi:10.1007/3-540-49059-0_14.

[33] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking Software, 13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162, 2006, doi:10.1007/11691617_9.

[34] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008, doi:10.1016/j.tcs.2008.03.013.

[35] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000, doi:10.1007/s100090050046.

[36] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97, 2005, doi:10.1007/11513988_9.

[37] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602, 2014, doi:10.1007/978-3-319-08867-9_39.

[38] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded verification of multi-threaded programs via lazy sequentialization. *ACM Transactions on Programming Languages and Systems*, 44(1):1:1–1:50, 2022, doi:10.1145/3478536.

[39] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to leads-to model checking. *The Computer Journal*, 65(6):1353–1364, 02 2021, doi:10.1093/comjnl/bxaa183.

[40] K. Mani Chandy and Jayadev Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989.

[41] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999*, pages 411–420. ACM, 1999, doi:10.1145/302405.302672.

[42] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. *RFC*, 5246:1–104, 2008, doi:10.17487/RFC5246.

[43] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018, doi:10.17487/RFC8446.

[44] Ling Dong and Kefei Chen. *Introduction of Cryptographic Protocols*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[45] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995, doi:10.1016/0020-0190(95)00144-2.

[46] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978, doi:10.1145/359657.359659.

[47] Ashutosh Satapathy and Jenila Livingston. A comprehensive survey on ssl/ tls and their vulnerabilities. *International Journal of Computer Applications*, 153:31–38, 11 2016, doi:10.5120/ijca2016912063.

[48] DF Franke. How poodle happened. retrieved december 21, 2014, 2014.

[49] Dawn Xiaodong Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999*, pages 192–202, 1999, doi:10.1109/CSFW.1999.779773.

[50] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop, CSFW-14 2001*, pages 82–96, 2001, doi:10.1109/CSFW.2001.930138.

[51] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko,

Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285, 2005, doi:10.1007/11513988_27.

[52] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418, 2008, doi:10.1007/978-3-540-70545-1_38.

[53] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701, 2013, doi:10.1007/978-3-642-39799-8_48.

[54] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 529–546. IEEE Computer Society, 2018, doi:10.1109/SP.2018.00033.

[55] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. In *Progress in Cryptology - 21st International Conference on Cryptology, INDOCRYPT 2020*, volume 12578 of *Lecture Notes in Computer Science*, pages 151–202, 2020, doi:10.1007/978-3-030-65277-7_8.

[56] John D. Ramsdell. Cryptographic protocol analysis and compilation using CPSA and roletran. In *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday*, volume 13066 of *Lecture Notes in Computer Science*, pages 355–369, 2021, doi:10.1007/978-3-030-91631-2_20.

[57] David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 497–510. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, doi:10.4230/LIPIcs.CONCUR.2015.497.

[58] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983, doi:10.1109/TIT.1983.1056650.

[59] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy*, pages 160–171, 1998, doi:10.1109/SECPRI.1998.674832.

[60] Santiago Escobar, Catherine A. Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006, doi:10.1016/j.tcs.2006.08.035.

[61] Santiago Escobar, Catherine A. Meadows, and José Meseguer. State space reduction in the Maude-NRL protocol analyzer. In *Computer Security - 13th European Symposium on Research in Computer Security, ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562, 2008, doi:10.1007/978-3-540-88313-5_35.

[62] Kazuhiro Ogata and Kokichi Futatsugi. Simulation-based verification for invariant properties in the OTS/CafeOBJ method. *Electronic Notes in Theoretical Computer Science*, 201:127–154, 2008, doi:10.1016/j.entcs.2008.02.018.

[63] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, 1995, doi:10.1007/3-540-60915-6_6.

[64] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994, doi:10.1006/inco.1994.1092.

[65] Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

[66] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992, doi:10.1016/0304-3975(92)90182-F.

[67] José Meseguer. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT 1997*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61, 1997, doi:10.1007/3-540-64299-4_26.

[68] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. In *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications, WRLA 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 153–182, 2004, doi:10.1016/j.entcs.2004.06.024.

[69] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*

- *20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391, 2014, doi:10.1007/978-3-642-54862-8_26.

[70] Lucas C. Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtík. JBMC: A bounded model checking tool for verifying java bytecode. In *Computer Aided Verification - 30th International Conference, CAV 2018*, volume 10981 of *Lecture Notes in Computer Science*, pages 183–190, 2018, doi:10.1007/978-3-319-96145-3_10.

[71] Omar Inverso and Catia Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2020*, pages 202–216, 2020, doi:10.1145/3332466.3374529.

[72] Prantik Chatterjee, Subhajit Roy, Bui Phi Diep, and Akash Lal. Distributed bounded model checking. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020*, pages 47–56, 2020, doi:10.34727/2020/isbn.978-3-85448-042-6_11.

[73] Thomas Neele, Anton Wijs, Dragan Bosnacki, and Jaco van de Pol. Partial-order reduction for GPU model checking. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016*, volume 9938 of *Lecture Notes in Computer Science*, pages 357–374, 2016, doi:10.1007/978-3-319-46520-3_23.

[74] Anton Wijs, Thomas Neele, and Dragan Bosnacki. Gpuexplore 2.0: Unleashing GPU explicit-state model checking. In *Formal Methods - 21st International Symposium, FM 2016*, volume 9995 of *Lecture Notes in Computer Science*, pages 694–701, 2016, doi:10.1007/978-3-319-48989-6_42.

[75] Richard DeFrancisco, Shenghsun Cho, Michael Ferdman, and Scott A. Smolka. Swarm model checking on the GPU. *International Journal on Software Tools for Technology Transfer*, 22(5):583–599, 2020, doi:10.1007/s10009-020-00576-x.

[76] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. On scheduling constraint abstraction for multi-threaded program verification. *IEEE Transactions on Software Engineering*, 46(5):549–565, 2020, doi:10.1109/TSE.2018.2864122.

[77] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157, 2013, doi:10.1007/978-3-642-39799-8_9.

[78] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. Parallel refinement for multi-threaded program verification. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 643–653, 2019, doi:10.1109/ICSE.2019.00074.

[79] Canh Minh Do and Kazuhiro Ogata. A divide & conquer approach to testing concurrent programs with JPF. In *27th Asia-Pacific Software Engineering Conference, APSEC 2020*, pages 356–364, 2020, doi:10.1109/APSEC51365.2020.00044.

[80] K. Rustan M. Leino. Developing verified programs with Dafny. In *35th International Conference on Software Engineering, ICSE 2013*, pages 1488–1490, 2013, doi:10.1109/ICSE.2013.6606754.

[81] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005, doi:10.1007/s10009-004-0167-4.

[82] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *Object-Oriented Programming - 19th European Conference, ECOOP 2005*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576, 2005, doi:10.1007/11531142_24.

[83] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In *Computer Aided Verification - 25th International Conference, CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868, 2013, doi:10.1007/978-3-642-39799-8_60.

[84] Weiqiang Kong, Leyuan Liu, Takahiro Ando, Hirokazu Yatsu, Kenji Hisazumi, and Akira Fukuda. Facilitating multicore bounded model checking with stateless explicit-state exploration. *The Computer Journal*, 58(11):2824–2840, 2015, doi:10.1093/comjnl/bxu127.

[85] Weiqiang Kong, Gang Hou, Xiangpei Hu, Takahiro Ando, Kenji Hisazumi, and Akira Fukuda. Garakabu2: an SMT-based bounded model checker for HSTM designs in ZIPC. *Journal of Information Security and Applications*, 31:61–74, 2016, doi:10.1016/j.jisa.2016.08.001.

[86] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007, doi:10.1109/TSE.2007.70724.

[87] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39, 1999, doi:10.1007/3-540-48234-2_3.

[88] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011, doi:10.1109/TSE.2010.110.

[89] Cormac Flanagan and Shaz Qadeer. Assume-guarantee model checking. In *SPIN*, page 11, 2003.

[90] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976, doi:10.1145/360051.360224.

[91] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983, doi:10.1145/69575.69577.

[92] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977, doi:10.1109/TSE.1977.229904.

[93] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*, volume 13 of *NATO ASI Series*, pages 123–144, 1984, doi:10.1007/978-3-642-82453-1_5.

[94] Leslie Lamport. Composition: A way to make proofs harder. In *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423, 1997, doi:10.1007/3-540-49213-5_15.

[95] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 3–12, 2002, doi:10.1109/ASE.2002.1114984.

[96] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, 2003, doi:10.1007/3-540-36577-X_24.

[97] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Distributed CTL model checking using mapreduce: theory and practice. *Concurrency and Computation: Practice and Experience*, 28(11):3025–3041, 2016, doi:10.1002/cpe.3652.

[98] Xinyu Chen, Hansheng Wei, Xin Ye, Li Hao, Yanhong Huang, and Jianqi Shi. Efficient parallel CTL model-checking for pushdown systems. In *IEEE International*

Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, ISPA/IUCC/BDCloud/SocialCom/-SustainCom 2018, Melbourne, Australia, December 11-13, 2018, pages 23–30, 2018, doi:10.1109/BDCloud.2018.00018.

[99] Dong Wang, Jing Liu, Haiying Sun, Jin Xu, and Jiexiang Kang. A fully parallel approach of model checking via probe machine. *International Journal of Software Engineering and Knowledge Engineering*, 31(11&12):1761–1781, 2021, doi:10.1142/S0218194021400210.

[100] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. Parallel breadth-first search LTL model-checking. In *18th IEEE International Conference on Automated Software Engineering, ASE 2003*, pages 106–115, 2003, doi:10.1109/ASE.2003.1240299.

[101] Richard E. Korf and Peter Schultze. Large-scale parallel breadth-first search. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1380–1385, 2005.

[102] Andy Yoo, Edmond Chow, Keith W. Henderson, Will McLendon III, Bruce Hendrickson, and Ümit V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, page 25, 2005, doi:10.1109/SC.2005.4.

[103] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *The 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010*, pages 303–314, 2010, doi:10.1145/1810479.1810534.

[104] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011*, pages 65:1–65:12, 2011, doi:10.1145/2063384.2063471.

[105] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[106] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using maude-npa. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, volume 8743 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2014, doi:10.1007/978-3-319-11851-2_11.

[107] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In *Term Rewriting and Applications, 16th International*

Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005, doi:10.1007/978-3-540-32033-3_22.

[108] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115. ACM, 2001, doi:10.1145/360204.360213.

[109] Canh Minh Do, Yati Phyo, Adrián Riesco, and Kazuhiro Ogata. A parallel stratified model checking technique/tool for leads-to properties. In *2021 7th International Symposium on System and Software Reliability, ISSSR 2021*, pages 155–166, 2021, doi:10.1109/ISSSR53171.2021.00011.

[110] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008, doi:10.1145/1327452.1327492.

[111] 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 *Formal Methods, 15th International Symposium on Formal Methods, FM 2008*, volume 5014 of *Lecture Notes in Computer Science*, pages 425–429, 2008, doi:10.1007/978-3-540-68237-0_31.

[112] Canh Minh Do and Kazuhiro Ogata. A divide & conquer approach to testing concurrent java programs with JPF and Maude. In *Structured Object-Oriented Formal Language and Method - 9th International Workshop, SOFL+MSVL 2019*, volume 12028 of *Lecture Notes in Computer Science*, pages 42–58, 2019, doi:10.1007/978-3-030-41418-4_4.

[113] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001, doi:10.1023/A:1011276507260.

[114] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 446–455, 2007, doi:10.1145/1250734.1250785.

[115] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, volume 5505 of *Lecture Notes in Computer Science*, pages 107–123, 2009, doi:10.1007/978-3-642-00768-2_11.

[116] Open source. Redis. `https://redis.io/`, 2009. [Online; accessed 07-February-2022].

[117] Open source. Rabbitmq. `https://www.rabbitmq.com/`, 2007. [Online; accessed 07-February-2022].

[118] Open source. Apache Commons Lang. `https://commons.apache.org/proper/commons-lang/`, 2001. [Online; accessed 07-February-2022].

[119] Open Source. MySQL. `https://www.mysql.com/`, 1995. [Online; accessed 07-February-2022].

[120] 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. `http://www.jaist.ac.jp/~ogata/slides/ECNU2014Nov27-28.pdf`.

[121] Canh Minh Do and Kazuhiro Ogata. Parallel stratified random testing for concurrent programs. In *20th IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2020*, pages 79–86, 2020, doi:10.1109/QRS-C51114.2020.00024.

[122] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *10th Intl Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998, doi:10.1007/BFb0028741.

[123] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010, doi:10.1093/logcom/exn075.

[124] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013*, pages 854–860, 2013.

[125] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A support tool for the L + 1-layer divide & conquer approach to leads-to model checking. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021*, pages 854–863, 2021, doi:10.1109/COMPSAC51774.2021.00118.

[126] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. An introduction to OBJ3. In *Conditional Term Rewriting Systems, 1st International Workshop*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263, 1987, doi:10.1007/3-540-19242-5_22.

[127] José Meseguer. Twenty years of rewriting logic. *Journal of Logical and Algebraic Methods in Programming*, 81(7-8):721–781, 2012, doi:10.1016/j.jlap.2012.06.003.

[128] Peter Csaba Ölveczky and José Meseguer. The Real-Time Maude tool. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 332–336, 2008, doi:10.1007/978-3-540-78800-3_23.

[129] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972, doi:10.1137/0201010.

[130] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990, doi:10.1109/71.80120.

[131] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991, doi:10.1145/103727.103729.

[132] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. Chapter 7 - spin locks and contention. In Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear, editors, *The Art of Multiprocessor Programming (Second Edition)*, pages 147–182. Morgan Kaufmann, Boston, second edition edition, 2021, doi:https://doi.org/10.1016/B978-0-12-415950-1.00017-3.

[133] Edsger W. Dijkstra. Some comments on the aims of MIRFAC. *Communications of the ACM*, 7(3):190, 1964, doi:10.1145/363958.364002.

[134] Shreyas Gokhale, Sahil Dhoked, and Neeraj Mittal. On group mutual exclusion for dynamic systems. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2021*, pages 446–447, 2021, doi:10.1145/3437801.3441608.

[135] Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *ACM Symposium on Principles of Distributed Computing, PODC 2020*, pages 1–10, 2020, doi:10.1145/3382734.3405739.

[136] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[137] Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 3–18, 1995.

[138] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109, 2012, doi:10.1007/978-3-642-28756-5_8.

[139] Kousha Etessami and Gerard J. Holzmann. Optimizing büchi automata. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2000, doi:10.1007/3-540-44618-4_13.

[140] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263, 2000, doi:10.1007/10722167_21.

[141] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, 2001, doi:10.1007/3-540-44585-4_6.

[142] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, 1995, doi:10.1007/3-540-60915-6_6.

[143] Yaacov Choueka. Theories of automata on omega-Tapes: A simplified approach. *Journal of Computer and System Sciences*, 8(2):117–141, 1974, doi:10.1016/S0022-0000(74)80051-6.

[144] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219, 2004, doi:10.1007/978-3-540-24730-2_18.

[145] Kyungmin Bae and José Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 99:193–234, 2015, doi:10.1016/j.scico.2014.02.006.

[146] Kyungmin Bae and José Meseguer. State/event-based LTL model checking under parametric generalized fairness. In *Computer Aided Verification - 23rd International Con-*

ference, CAV 2011, volume 6806 of *Lecture Notes in Computer Science*, pages 132–148, 2011, doi:10.1007/978-3-642-22110-1_11.

[147] Canh Minh Do, Adrián Riesco, Santiago Escobar, and Kazuhiro Ogata. Parallel maude-npa for cryptographic protocol analysis. In *Rewriting Logic and Its Applications - 14th International Workshop*, pages 253–273, 2022, doi:10.1007/978-3-031-12441-9_13.

[148] Kwame-Lante Wright and Kartik Gopalan. Performance analysis of inter-process communication mechanisms. Technical report, Binghamton University, Tech. Rep. TR-20070820, 2007.

[149] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 52–68, 2010, doi:10.1007/978-3-642-16310-4_5.

[150] Santiago Escobar and José Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168, 2007, doi:10.1007/978-3-540-73449-9_13.

[151] Peter Csaba Ölveczky. Real-Time Maude and its applications. In *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014*, volume 8663 of *Lecture Notes in Computer Science*, pages 42–79, 2014, doi:10.1007/978-3-319-12904-4_3.

[152] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Full Maude: Extending core Maude. In *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*, pages 559–597. Springer, 2007, doi:10.1007/978-3-540-71999-1_18.

[153] Peter Csaba Ölveczky, Mark Keaton, José Meseguer, Carolyn L. Talcott, and Steve Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–348, 2001, doi:10.1007/3-540-45314-8_24.

[154] Canh Minh Do, Yati Phyo, and Kazuhiro Ogata. A divide & conquer approach to until and until stable model checking. In *The 34th International Conference on Software Engineering & Knowledge Engineering, SEKE 2022*, 2022, doi:10.18293/SEKE2022-058.

[155] Razvan Diaconescu and Kokichi Futatsugi. *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998, doi:10.1142/3831.

[156] Kokichi Futatsugi, Daniel Gâinâ, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. *Theoretical Computer Science*, 464:90–112, 2012, doi:10.1016/j.tcs.2012.07.041.

[157] Adrián Riesco and Kazuhiro Ogata. Prove it! inferring formal proof scripts from CafeOBJ proof scores. *ACM Transactions on Software Engineering and Methodology*, 27(2):6:1–6:32, 2018, doi:10.1145/3208951.

[158] Adrián Riesco and Kazuhiro Ogata. An integrated tool set for verifying CafeOBJ specifications. *Journal of Systems and Software*, 189:111302, 2022, doi:https://doi.org/10.1016/j.jss.2022.111302.

[159] Adrián Riesco and Kazuhiro Ogata. CiMPG+F: A proof generator and fixer-upper for CafeOBJ specifications. In *Theoretical Aspects of Computing - 17th International Colloquium, ICTAC 2020*, volume 12545 of *Lecture Notes in Computer Science*, pages 64–82, 2020, doi:10.1007/978-3-030-64276-1_4.

[160] Duong Dinh Tran and Kazuhiro Ogata. IPSG: invariant proof score generator. In *46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022*, pages 1050–1055. IEEE, 2022, doi:10.1109/COMPSAC54236.2022.00164.

First-author Publications

[1] Canh Minh Do and Kazuhiro Ogata. Specification-based testing with simulation relations. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*, pages 107–146, 2019, doi:10.18293/SEKE2019-027.

[2] Canh Minh Do and Kazuhiro Ogata. A divide & conquer approach to testing concurrent java programs with JPF and Maude. In *Structured Object-Oriented Formal Language and Method - 9th International Workshop, SOFL+MSVL 2019*, volume 12028 of *Lecture Notes in Computer Science*, pages 42–58, 2019, doi:10.1007/978-3-030-41418-4_4.

[3] Canh Minh Do and Kazuhiro Ogata. A divide & conquer approach to testing concurrent programs with JPF. In *27th Asia-Pacific Software Engineering Conference, APSEC 2020*, pages 356–364, 2020, doi:10.1109/APSEC51365.2020.00044.

[4] Canh Minh Do and Kazuhiro Ogata. Parallel stratified random testing for concurrent programs. In *20th IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2020*, pages 79–86, 2020, doi:10.1109/QRS-C51114.2020.00024.

[5] Canh Minh Do, Yati Phyo, Adrián Riesco, and Kazuhiro Ogata. A parallel stratified model checking technique/tool for leads-to properties. In *7th International Symposium on System and Software Reliability, ISSSR 2021*, pages 155–166, 2021, doi:10.1109/ISSSR53171.2021.00011.

[6] Canh Minh Do and Kazuhiro Ogata. Parallel specification-based testing for concurrent programs. *IEEE Access*, 10:24955–24975, 2022, doi:10.1109/ACCESS.2022.3155629.

[7] Canh Minh Do, Adrián Riesco, Santiago Escobar, and Kazuhiro Ogata. Parallel maude-npa for cryptographic protocol analysis. In *Rewriting Logic and Its Applications - 14th International Workshop*, pages 253–273, 2022, doi:10.1007/978-3-031-12441-9_13.

[8] Canh Minh Do, Yati Phyo, and Kazuhiro Ogata. A divide & conquer approach to until and until stable model checking. In *The 34th International Conference on Software Engineering & Knowledge Engineering, SEKE 2022*, 2022, doi:10.18293/SEKE2022-058.

The Other Co-author Publications

[1] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. Toward development of a tool supporting a 2-layer divide & conquer approach to leads-to model checking. In *International Conference on Advanced Infocomm Technology, ICAIT 2019*, pages 250–255, 2019, doi:10.1109/AITC.2019.8920978.

[2] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to leads-to model checking. *The Computer Journal*, 65(6):1353–1364, 02 2021, doi:10.1093/comjnl/bxaa183.

[3] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A support tool for the L + 1-layer divide & conquer approach to leads-to model checking. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021*, pages 854–863, 2021, doi:10.1109/COMPSAC51774.2021.00118.

[4] Moe Nandi Aung, Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to eventual model checking. *Mathematics*, 9(4):16 pages, 2021, doi:10.3390/math9040368.

[5] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to conditional stable model checking. In *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium*, volume 12819 of *Lecture Notes in Computer Science*, pages 105–111, 2021, doi:10.1007/978-3-030-85315-0_7.

[6] Thet Wai Mon, Dang Duy Bui, Duong Dinh Tran, Canh Minh Do, and Kazuhiro Ogata. Graphical animations of the NSLPK authentication protocols. *Journal of Visual Language and Computing*, 2021(2):39–51, 2021, doi:10.18293/jvlc2021-n2-005.

[7] Moe Nandi Aung, Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A tool for model checking eventual model checking in a stratified way. In *The 9th International Conference on Dependable Systems and Their Applications, DSA 2022*, 2022. to appear.