Title	分割統治による誘導性と条件付安定性のモデル検査
Author(s)	YATI, PHYO
Citation	
Issue Date	2022-09
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/18134
Rights	
Description	Supervisor:緒方 和博, 先端科学技術研究科, 博士

Doctoral Dissertation

A Divide & Conquer Approach to Leads-to and Conditional Stable Model Checking

Yati Phyo

Supervisor: Kazuhiro Ogata

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)
September, 2022

Abstract

Model checking is one of the most successful computer science achievements in the last few decades. This is why Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis were honored with 2007 A.M. Turing Award for their role in developing model checking into a highly effective verification technology. Model checking has been widely adopted in industries, especially in hardware ones. There are still some issues to tackle in model checking, one of which is the notorious state explosion. Many techniques to mitigate the state explosion, such as partial order reduction and abstraction, have been devised. Because these existing techniques are not too enough to deal with the state explosion, however, it is still worth tackling it.

To mitigate the state space explosion problem to some extent, we propose a divide & conquer approach to model checking (DCA2MC) leads-to properties and conditional stable properties, where leads-to properties are expressed in $\varphi_1 \rightsquigarrow \varphi_2$ and conditional stable properties are expressed in $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions or classical propositional formulas. Chandy and Misra 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. Moreover, Dwyer et al. 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. Conditional stable properties can be used to express core requirements in self-stabilizing systems, which were first introduced by Dijkstra and became a very important concept in fault tolerance to design robust systems. Thus, it is worth focusing on leads-to and conditional stable properties. DCA2MC divides an original leads-to (or conditional stable) model checking problem into multiple smaller model checking problems and tackles each smaller one. We prove a theorem that the multiple smaller model checking problems are equivalent to the original leads-to (or conditional stable) model checking problem. An algorithm is constructed based on the theorem to support model checking leads-to (or conditional stable) properties by DCA2MC. A support tool is developed in Maude, a rewriting logic-based specification/programming language and system, to support the technique based on the algorithm for each of leads-to and conditional stable properties. Some experiments are then conducted with the support tools to demonstrate that our tools/techniques can mitigate the state space explosion to some extent.

Both leads-to and conditional stable properties can be expressed as linear temporal logic (LTL) formulas that are very similar. However, how to deal with the two classes of properties

with DCA2MC is so different that we need to prove the correctness of DCA2MC for the two classes of properties in two different ways and come up with two different algorithms for the two classes of properties, from which the two support tools are built. Note that because the architecture of the tools is well designed, many components (data structures and functions) are shared by the two tools. This is because it suffices to take a look at the top state of an infinite sequence π of states so as to check if a state proposition φ_2 holds for π, while it is necessary to take a look at all states in π in order to check if an LTL formula $\Box\varphi_2$, where φ_2 is a state proposition, holds for π. One piece of our future work is to extend DCA2MC for the other classes of LTL properties and then come up with a unified DCA2MC for all LTL properties.

Keywords: LTL model checking; leads-to properties; conditional stable properties; state space explosion; divide & conquer; Maude; meta-programming; Full Maude.

Acknowledgments

Firstly, I want to deeply express my thanks from the bottom of my heart to my supervisor, Professor Kazuhiro Ogata who supports and kindly advises me before coming to Japan until now in any situation of me. In my most difficult situations, I could keep on doing only because of Ogata Sensei's motivation. Without Ogata Sensei, I have lost myself and I cannot be the one who is in this current situation. Best motivations, critical thinking, creative ideas, guidance, and some other factors from him are the main reasons that directly affect the completion of both Master's and Doctoral theses of mine.

I also want to give special thanks to my second supervisor, Professor Kunihiko Hiraishi, also my minor research supervisor Professor Masashi Unoki, Adrian Riesco Rodriguez, and my PhD thesis committee members, Professor Daisuke Ishii, Professor Kozo Okano from Sinshu University, and Professor Masaki Nakamura from Toyama Prefectural University, who are very kind, supportive, and give advice for having better research.

Moreover, I want to say thank you to all Professors, Associate Professors, Assistant Professors, and Lecturers who taught me valuable courses. And also, I want to thank all staff members at the Japan Advanced Institute of Science and Technology who helped me indirectly.

I would also like to express my gratitude to my lab members for sharing good ideas, and memories not only in my research but also in my daily life. I also thank to my friends who share happiness and help me to fulfill my happiness.

Finally, I want to send a very big thanks to my family who always believes in me and supports me in whatever I do. I cannot be myself without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Because many systems/protocols have been used for various roles in our daily lives, it is very important to make sure that such systems/protocols are trustworthy. To this end, formal verification is promising. As a formal verification technique, model checking is popular. Model checking has been used in hardware industries. To make it possible to use model checking as a daily life technique in software industries, there is a big challenge to conquer: the state space explosion problem. Many techniques have been proposed to mitigate the problem, but the problem still degrades model checking running performance or even makes it impossible to carry out model checking experiments when tackling large-state systems/protocols. We came up with two new techniques to alleviate the problem for two important classes of properties (leads-to and conditional stable properties), developed two tools that support the two techniques, and conducted case studies that demonstrate that the techniques/tools can reasonably alleviate the state space explosion problem. In this chapter, we briefly introduce model checking, state our main contributions, and describe our thesis structure.

1.1 Model Checking

Model checking is one of the most successful computer science achievements in the last few decades. This is why Edmund M. Clarke, together with E. Allen Emerson and Joseph Sifakis, was honored with 2007 A. M. Turing Award[1] for their role in developing model checking into a highly effective verification technology. To verify systems by model checking, we need to prepare two specifications: one is a system specification and the other one is a property specification. A system specification describes the behaviors of a system, which can be formalized as a state machine or a Kripke structure, an extension of a state machine. Meanwhile, a property specification describes a desired requirement/property that the system should satisfy, which can be often formalized in a temporal logic, such as linear temporal logic (LTL) and computational tree logic (CTL). A model checker automates to formally verify that a system (formalized as

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

a Kripke structure K) satisfies a property (formalized as a temporal logic formula φ), which is expressed as $K \models \varphi$. If K does not satisfy φ, expressed as $K \not\models \varphi$, then a model checker returns a counterexample, an infinite sequence of states, that can be expressed as a pair of a finite sequence of states from an initial state and a finite cycle of states. The automata-theoretic approach [1, 2] to model checking is often used to implement model checkers. In this approach, a system (or a system specification) is transformed to a Kripke structure K_{sys}, the negation of a property (or a property specification) is also transformed to a Kripke structure $K_{\neg prop}$, the product $K_{sys} \otimes K_{\neg prop}$ is constructed from the two Kripke structures, and it is checked whether the infinite languages accepted by $K_{sys} \otimes K_{\neg prop}$ is empty. If it is empty, the system satisfies the property; otherwise, it does not, and an infinite language accepted by $K_{sys} \otimes K_{\neg prop}$ is a counterexample. Another approach to model checking that is also often used is bounded model checking (BMC) based on Boolean satisfiability solvers or such solvers modulo theories (SAT/SMT-based BMC) [3, 4, 5]. In this approach, BMC is transformed into a Boolean formula and such a formula is solved by a SAT/SMT solver. If there is a solution, it is a counterexample; otherwise, the system satisfies the property up to some specific depth from each initial state of the system. Several optimization techniques for model checking have been proposed. Among them are binary decision diagrams (BDDs) [6, 7], symmetry reduction [8] and partial order reduction [9]. Many model checkers have been developed, such as Spin [10], NuSMV [11], Symbolic Analysis Laboratory (SAL) [12], Process Analysis Toolkit (PAT) [13], TLA+ model checker (TLC) [14] and Maude LTL model checker [15, 16]. These model checkers may be called design model checkers because their main targets are systems designs. There are model checkers whose main targets are programs written in programming languages, such as Java, C and C++. Such model checkers may be called software model checkers. Among software model checkers are Java Pathfinder (JPF) [17], BLAST [18] and DiVinE [19].

It has been proven to be a tremendously successful technique to verify requirements for a variety of hardware and software systems. However, there are still some challenges to tackle in model checking, one of which is the state space explosion that occurs when the number of states becomes exponential in the number of processes. Many techniques have been devised to mitigate the state explosion, such as abstraction (and abstraction refinement) [20, 21, 22, 23, 24, 25, 26, 27], SAT/SMT-based BMC (and its extension) [28, 29, 30, 31, 32, 33] and non-exhaustive model checking [34, 35, 36] as well as those above-mentioned. Because those existing techniques are not enough to deal with the state explosion, however, it is still worth tackling it.

To address the problem to some extent, we propose a technique called a divide & conquer approach to leads-to model checking (L + 1-DCA2L2MC). As indicated by the name, the technique is dedicated to leads-to properties. An informal description of leads-to properties is that whenever something occurs, something else will eventually occur. For example, one desired property mutual exclusion protocols should enjoy can be expressed as a leads-to property: whenever a process wants to enter a critical section, it will eventually be there. Chandy and

Misra [37] 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. Moreover, Dwyer et al. [38] 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. Thus, it is worth focusing on leads-to properties.

We have extended our approach to model checking to another important class of properties, conditional stable properties. The extended approach is called a divide & conquer approach to conditional stable model checking (L+1-DCA2CSMC). As indicated by the name, the technique is dedicated to conditional stable properties that informally say that whenever something is true, it will eventually happen that something else will be always true (or will be stable). The properties can be used to express desired properties that self-stabilizing systems [39] should satisfy. As known, the term of self-stabilization in distributed systems was first introduced by Dijkstra [40] and became a very important concept in fault tolerance to design a robust system because the system is subject to transient errors at any time, such as process crashes. A system is self-stabilizing with respect to a set of legitimate states if starting from an arbitrary initial state, the system guarantees to converge to a legitimate state in a finite number of state transitions and remains in the legitimate states thereafter. A state is legitimate if starting from this state the system satisfies its desired properties. Designing self-stabilizing systems needs much more efforts than non-stabilizing ones because transient errors can occur at any time in a system, which often drives the system into an arbitrary state after each transient error. Therefore, it is worth dedicating to formal verification of the conditional stable properties in order to guarantee that self-stabilizing systems can reach a legitimate state from an arbitrary state after a finite number of state transitions.

Although leads-to properties and conditional stable properties can be expressed in LTL, it is necessary to individually prove the correctness of each of the two divide & conquer approaches to leads-to and conditional stable model checking, come up with each algorithm, and develop each tool supporting each approach. This is because it is not straightforward to uniformly deal with the two different properties so as to mitigate the state space explosion. Moreover, each of the two properties can be used to express many systems requirements/properties. Thus, it is worth focusing on the two approaches to model checking the two classes of properties, the correctness of the two approaches, their algorithms as well as their support tools. We also report on some experiments showing that our tools/techniques can mitigate the state space explosion to some extent.

1.2 Contributions

In summary, we concentrate on mitigating the state space explosion in model checking with leads-to and conditional stable properties that many systems should satisfy. The following are our contributions:

Leads-to Model Checking

- We propose a new technique to mitigate the state space explosion in model checking that is dedicated to leads-to properties. The technique is called $L + 1$-layer divide & conquer approach to leads-to model checking ($L + 1$-DCA2L2MC).
- We prove a theorem that the original leads-to model checking problem is equivalent to the smaller model checking problems tackled by our technique. An algorithm is constructed based on the theorem so as to conduct model checking.
- We develop a tool in Maude to support the technique and conduct some experiments showing that the tool/technique can mitigate the state space explosion to some extent.
- The tool and case studies are publicly available at the webpage.[2]

Conditional Stable Model Checking

- We propose the $L+1$-layer divide & conquer approach to conditional stable model checking ($L + 1$-DCA2CSMC) that is an extension of $L + 1$-DCA2L2MC to handle the conditional stable properties that can be used to formalize desired properties in self-stabilizing systems.
- We prove a theorem that the original conditional stable model checking problem is equivalent to the smaller model checking problems tackled by our technique. An algorithm is constructed based on the theorem so as to conduct model checking.
- We develop a tool in Maude to support the technique and conduct some experiments showing that the tool/technique can mitigate the state space explosion to some extent. The tool and case studies are publicly available in Footnote 2 as well.

1.3 Thesis Structure

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

[2] `https://github.com/yatiphyo/DCA2MC`

- **Chapter 1** introduces model checking and the notorious state space explosion in model checking. Some techniques to mitigate the problem are mentioned along with our techniques that can mitigate state space explosion to some extent. Lastly, our contributions and the thesis structure are described.

- **Chapter 2** provides some preliminaries needed for this thesis. They are Kripke structure, Linear temporal logic (LTL), and Maude. A simple mutual exclusion protocol (Qlock) is used as an example to explain how to formally specify the protocol in Maude and how to model check the protocol enjoys a desired property. Meta-programming in Maude is described with the example. The state space explosion problem is presented at the end.

- **Chapter 3** investigates some classical techniques as well as other techniques used to mitigate the state space explosion problem in model checking. We also compare our techniques with some existing techniques as well.

- **Chapter 4** proposes $L + 1$-DCA2L2MC so as to mitigate the state space explosion in model checking. This chapter sketches out the technique with an example (TAS) and proves a theorem that the original leads-to model checking problem is equivalent to the smaller model checking problems tackled by L+1-DCA2L2MC. This chapter also describes an algorithm that is constructed from the theorem to conduct model checking with the technique.

- **Chapter 5** proposes $L + 1$-DCA2CSMC so as to mitigate the state space explosion in model checking. This chapter sketches out the technique with an example (KM) and proves a theorem that the original conditional stable model checking problem is equivalent to the smaller model checking problems tackled by $L + 1$-DCACSMC. This chapter also describes an algorithm that is constructed from the theorem to conduct model checking with the technique.

- **Chapter 6** describes how to implement a support tool in Maude for $L + 1$-DCA2L2MC. Some experiments are conducted to demonstrate that the tool/technique can mitigate the state space explosion to some extent.

- **Chapter 7** describes how to implement a support tool in Maude for $L + 1$-DCA2CSMC. Some experiments are conducted to demonstrate that the tool/technique can mitigate the state space explosion to some extent.

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

Chapter 2

Preliminaries

Some preliminaries needed to read the remaining part of the thesis are written. They are basically Kripke structures, linear temporal logic, and Maude. Qlock, an abstract version of Dijkstra binary semaphore, is used as an example to describe how to formally specify systems/protocol in Maude and how to model check that a system/protocol satisfies a desired property. Tools are implemented in Maude with its meta-programming facilities. Therefore, meta-programming in Maude is described, in which Qlock is used as an example. The state space explosion problem is mentioned by using a model checking experiment with Maude in which Qlock is also used.

2.1 Kripke Structures

We use the symbol $\triangleq$ as "if and only if" or "be defined as."

Definition 2.1 (Kripke structures) *A Kripke structure* $\boldsymbol{K} \triangleq \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{T}, \boldsymbol{A}, \boldsymbol{L} \rangle$ *consists of a set* $\boldsymbol{S}$ *of states, a set* $\boldsymbol{I} \subseteq \boldsymbol{S}$ *of initial states, a left-total binary relation* $\boldsymbol{T} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ *over states, a set* $\boldsymbol{A}$ *of atomic propositions and a labeling function* $\boldsymbol{L}$ *whose type is* $\boldsymbol{S} \to 2^{\boldsymbol{A}}$*. An element* $(s, s') \in \boldsymbol{T}$ *is called a (state) transition from* s *to* s' *and may be written as* $s \to_{\boldsymbol{K}} s'$*.*

$\boldsymbol{S}$ does not need to be finite. The set $\boldsymbol{R}$ of reachable states is inductively defined as follows: $\boldsymbol{I} \subseteq \boldsymbol{R}$ and if $s \in \boldsymbol{R}$ and $(s, s') \in \boldsymbol{T}$, then $s' \in \boldsymbol{R}$. We suppose that $\boldsymbol{R}$ is finite. $\boldsymbol{K}$ in $s \to_{\boldsymbol{K}} s'$ may be omitted if it is clear from the context.

An infinite sequence $s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$ of states is called a path of $\boldsymbol{K}$ if and only if for any natural number i, $(s_i, s_{i+1}) \in \boldsymbol{T}$. Let π be $s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$ and some path notations

are defined as follows:

$$
\begin{aligned}
&\pi(i) \triangleq s_i \\
&\pi^i \triangleq s_i, s_{i+1}, \ldots \\
&\pi_i \triangleq s_0, s_1, \ldots, s_i, s_i, \ldots \\
&\pi_\infty \triangleq \pi \\
&\pi^{(i,j)} \triangleq \begin{cases} s_i, s_{i+1}, \ldots, s_j, s_j, \ldots & \textbf{if } i \leq j \\ s_i, s_i, \ldots & \textbf{otherwise} \end{cases} \\
&\pi^{(i,\infty)} \triangleq \pi^i \\
&\pi^i_j \triangleq \pi^{(i,j)}
\end{aligned}
$$

where i and j are any natural numbers. Note that $\pi^{(0,j)} = \pi_j$. Note that $\pi_i(k) = \pi(k)$ if $k = 0, \ldots, i$ and $\pi_i(k) = \pi(i)$ if $k > i$. Note that $\pi^{(i,j)}(k) = \pi(i+k)$ if $i \leq j$ & $k = 0, \ldots, m$, where $j = i + m$, $\pi^{(i,j)}(k) = \pi(j)$ if $i \leq j$ & $k > j$ and $\pi^{(i,j)}(k) = \pi(i)$ if $i > j$ & k is a natural number. A path π of $\boldsymbol{K}$ is called a computation of $\boldsymbol{K}$ if and only if $\pi(0) \in \boldsymbol{I}$.

Let $\boldsymbol{P_K}$ be the set of all paths of $\boldsymbol{K}$. Let $\boldsymbol{P}_{(\boldsymbol{K},s)}$ be $\{\pi \mid \pi \in \boldsymbol{P_K}, \pi(0) = s\}$, where $s \in \boldsymbol{S}$. Let $\boldsymbol{P}^b_{(\boldsymbol{K},s)}$ be $\{\pi_b \mid \pi \in \boldsymbol{P}_{(\boldsymbol{K},s)}\}$, where $s \in \boldsymbol{S}$ and b is a natural number. Note that $\boldsymbol{P}^\infty_{(\boldsymbol{K},s)}$ is $\boldsymbol{P}_{(\boldsymbol{K},s)}$. If $\boldsymbol{R}$ is finite and $s \in \boldsymbol{R}$, then $\boldsymbol{R}^b_{(\boldsymbol{K},s)}$ is finite because the bound $\boldsymbol{b}$ is used. Even if $\boldsymbol{R}$ is finite and $s \in \boldsymbol{R}$, $\boldsymbol{P}_{(\boldsymbol{K},s)}$ may not be finite. Because there are a finite number of different states in each path in $\boldsymbol{R}_{(\boldsymbol{K},s)}$, it is possible to check $\boldsymbol{K}, \pi \models \varphi$ for all $\pi \in P_{(\boldsymbol{K},s)}$ in a finite amount of time.

2.2 Linear Temporal Logic (LTL)

Definition 2.2 (Syntax of LTL) *The syntax of linear temporal logic (LTL) is as follows:*

$$\varphi ::= a \mid \top \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \, \mathcal{U} \, \varphi$$

where $a \in \boldsymbol{A}$.

Definition 2.3 (Semantics of LTL) *For any Kripke structure $\boldsymbol{K}$, any path π of $\boldsymbol{K}$ and any LTL formulas φ, $\boldsymbol{K}, \pi \models \varphi$ is inductively defined as follows:*

- $\boldsymbol{K}, \pi \models a$ *if and only if* $a \in \boldsymbol{L}(\pi(0))$
- $\boldsymbol{K}, \pi \models \top$
- $\boldsymbol{K}, \pi \models \neg\varphi_1$ *if and only if* $\boldsymbol{K}, \pi \not\models \varphi_1$
- $\boldsymbol{K}, \pi \models \varphi_1 \vee \varphi_2$ *if and only if* $\boldsymbol{K}, \pi \models \varphi_1$ *and/or* $\boldsymbol{K}, \pi \models \varphi_2$
- $\boldsymbol{K}, \pi \models \bigcirc \varphi_1$ *if and only if* $\boldsymbol{K}, \pi^1 \models \varphi_1$

- $\boldsymbol{K}, \pi \models \varphi_1 \, \mathcal{U} \, \varphi_2$ *if and only if there exists a natural number* i *such that* $\boldsymbol{K}, \pi^i \models \varphi_2$ *and for each natural number* $j < i$, $\boldsymbol{K}, \pi^j \models \varphi_1$

where φ_1 *and* φ_2 *are LTL formulas. Then,* $\boldsymbol{K} \models \varphi$ *if and only if* $\boldsymbol{K}, \pi \models \varphi$ *for all computations* π *of* $\boldsymbol{K}$.

Some other connectives are defined as follows:

- $\bot \triangleq \neg\top$,
- $\varphi_1 \wedge \varphi_2 \triangleq \neg((\neg\varphi_1) \vee (\neg\varphi_2))$,
- $\varphi_1 \Rightarrow \varphi_2 \triangleq (\neg\varphi_1) \vee \varphi_2$,
- $\varphi_1 \Leftrightarrow \varphi_2 \triangleq (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$,
- $\Diamond\varphi_1 \triangleq \top \, \mathcal{U} \, \varphi_1$,
- $\Box\varphi_1 \triangleq \neg(\Diamond\neg\varphi_1)$, and
- $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$.

where $\bigcirc$, $\mathcal{U}$, $\Diamond$, $\Box$ and $\rightsquigarrow$ are called next, until, eventually, always, and leads-to temporal connectives, respectively.

The thesis focuses on the LTL formula $\varphi_1 \rightsquigarrow \varphi_2$ such that φ_1 is an LTL formula in which no temporal connective is used (called a state proposition) and φ_2 is an LTL formula that is in the form $\Box\varphi_3$ where φ_3 is a state proposition. The properties expressed as the former LTL formulas are called leads-to properties, while the properties expressed as the latter LTL formulas are called conditional stable properties in the thesis. This is because the two classes of temporal properties are often used to express important properties of systems and how to treat the two forms of formulas in the proposed technique (a divide & conquer approach to model checking) is different, implying that how to treat $\varphi_1 \rightsquigarrow \varphi_2$ depends on the forms of φ_1 and φ_2. Thus, we define state propositions, which is as follows:

Definition 2.4 (State propositions) *State propositions are LTL formulas such that they do not have any temporal connectives.*

Proposition 2.1 *Let* $\boldsymbol{K}$ *be any Kripke structure. If* φ *is any state proposition, then* $(\boldsymbol{K}, \pi \models \varphi) \Leftrightarrow (\boldsymbol{K}, \pi' \models \varphi)$ *for any paths* π *&* π' *of* $\boldsymbol{K}$ *such that* $\pi(0) = \pi'(0)$.

Proof 2.1 *The first state* $\pi(0)$ *decides if* $\boldsymbol{K}, \pi \models \varphi$ *holds.* $\Box$

Let $\boldsymbol{K}, s \models \varphi$, where $s \in \boldsymbol{S}$, be $\boldsymbol{K}, \pi \models \varphi$ for all $\pi \in \boldsymbol{P}_{(\boldsymbol{K},s)}$. Note that $\boldsymbol{K}, s \models \varphi$ for all $s \in \boldsymbol{I}$ is equivalent to $\boldsymbol{K} \models \varphi$. Let $\boldsymbol{K}, s, b \models \varphi$, where $s \in \boldsymbol{S}$ and b is a natural number or ∞, be $\boldsymbol{K}, \pi \models \varphi$ for all $\pi \in \boldsymbol{P}^b_{(\boldsymbol{K},s)}$. Note that $\boldsymbol{K}, s, \infty \models \varphi$ is $\boldsymbol{K}, s \models \varphi$.

Some logical connectives are used as meta-logical connectives for $\boldsymbol{K}, \pi \models \varphi$ as follows:

- $(\boldsymbol{K}, \pi \models \varphi) \wedge (\boldsymbol{K'}, \pi' \models \varphi') \triangleq \boldsymbol{K}, \pi \models \varphi$ and $\boldsymbol{K'}, \pi' \models \varphi'$
- $(\boldsymbol{K}, \pi \models \varphi) \vee (\boldsymbol{K'}, \pi' \models \varphi') \triangleq \boldsymbol{K}, \pi \models \varphi$ or $\boldsymbol{K'}, \pi' \models \varphi'$
- $(\boldsymbol{K}, \pi \models \varphi) \Rightarrow (\boldsymbol{K'}, \pi' \models \varphi') \triangleq$ if $\boldsymbol{K}, \pi \models \varphi$, then $\boldsymbol{K'}, \pi' \models \varphi'$
- $(\boldsymbol{K}, \pi \models \varphi) \Leftrightarrow (\boldsymbol{K'}, \pi' \models \varphi') \triangleq \boldsymbol{K}, \pi \models \varphi$ if and only if $\boldsymbol{K'}, \pi' \models \varphi'$

Leads-to Properties and Conditional Stable Properties

We rephrased the semantics of $\varphi_1 \rightsquigarrow \varphi_2$ and $\varphi_1 \rightsquigarrow \Box\varphi_2$ as follows so that readers can more intuitively comprehend it:

- $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ if and only if for each natural number i such that $\boldsymbol{K}, \pi^i \models \varphi_1$, there exists a natural number $j \geq i$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$.
- $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$ if and only if for each natural number i such that $\boldsymbol{K}, \pi^i \models \varphi_1$, there exists a natural number $j \geq i$ such that $\boldsymbol{K}, \pi^j \models \Box\varphi_2$.

2.3 Maude

We use Maude [41], a specification/programming language based on rewriting logic, as a specification language and an LTL model checker. The reason why we use Maude is that Maude makes it possible to concisely specify complex systems because it is possible to use inductively defined data structures including associative-commutative binary operators in systems specifications model checked. It is also possible to use what are called matching equations as part of the condition of a conditional rewrite rule, which makes it possible to specify complex state transitions in a reasonably concise way. Its accompanying LTL model checker is reasonably fast, comparable to Spin in terms of both running time and memory consumption.

2.3.1 Formal Specification and Model Checking

Formal Specification

There are multiple possible ways to specify state transitions. We use rewrite rules specified in Maude to formalize the state transitions. Besides, we can use Maude LTL model checker to conduct model checking experiments.

A soup is a collection of elements such that the order of elements is irrelevant, such as multi-sets. Soups are constructed with a constant that represents the empty soup and a binary operator that is associative and commutative, where the constant is an identity of the binary operator. A pair of name and value, such as (pc[p] : cs), is called an observable component. Observable components are used to characterize states. For example, (pc[p] : cs) means that

process p is located at cs (critical-section). A state is formalized as a braced soup of observable components. For example, if a state can be characterized by three observable components oc_1, oc_2, oc_3 for some specific purpose, a state is formalized as $\{oc_1\ oc_2\ oc_3\}$, where the empty syntax (or the juxtaposition operator) is used as the binary operator with which soups are constructed. Transitions can be described in terms of rewrite rules.

In this section, we use a simple mutual exclusion protocol as an example to sketch out the approach. The protocol is called Qlock, which can also be regarded as an abstract version of the Dijkstra binary semaphore in which an atomic queue is employed. Qlock for each process p can be described as:

"Start Section"
ss : enq($queue, p$);
ws : **while until** top($queue$) = p;
"Critical Section"
cs : deq($queue$);
"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. We are also not interested in what p does in Critial Section and then what p does at cs is to move to fs from cs. Furthermore, we are not interested in what p does in Final Section and then what p does at fs stays there forever.

queue is an atomic queue of process IDs shared by all processes. enq, top, and deq are the atomic operators of atomic queues. *queue* is initially empty. When p would like to come into cs, it puts its ID into *queue* and goes to ws. It waits at ws while the top of *queue* is not p. Whenever the top of *queue* is p, p comes into cs. When it exits cs, it gets rid of the top from *queue* and goes to fs. We suppose that each process goes to cs at most once.

When n processes taking part in Qlock, each state in S_{Qlock} is formalized as:

`{(queue:` q`) (pc[`p_1`]:` l_1`) ... (pc[`p_n`]:` l_n`) (cnt:` x`)}`

The value of the atomic queue is saved in the value q of the `queue` observable component. q is initially an empty queue denoted `empq`. The place of each process p_i is saved in the label l_i of the `pc[`p_i`]` observable component. l_i is initially ss. The number of processes that have not reached fs yet is saved in the number x of the `cnt` observable component. x is initially n.

Let us suppose that two processes `p1` and `p2` take part in Qlock. The initial state denoted `init` is as:

```
{(queue: empq) (pc[p1]: ss) (pc[p2]: ss) (cnt: 2)}
```

I_{Qlock} has one state `init`.

T_{Qlock} is described in rewrite rules in what follows:

```
rl [start] : {(queue: Q) (pc[I]: ss) OCs} => {(queue: (Q | I)) (pc[I]: ws) OCs} .
rl [wait] : {(queue: (I | Q)) (pc[I]: ws) OCs} => {(queue: (I | Q)) (pc[I]: cs) OCs} .
rl [exit] :  {(queue: Q) (pc[I]: cs) (cnt: N) OCs}
  => {(queue: deq(Q)) (pc[I]: fs) (cnt: dec(N)) OCs} .
rl [fin] : {(cnt: 0) OCs} => {(cnt: 0) OCs} .
```

The four rewrite rules are named `start`, `wait`, `exit`, and `fin`. `Q`, `I`, `N`, and `OCs` are Maude variables whose sorts are queues of process IDs, process IDs, natural numbers, and observable component soups, respectively. `dec` is used to decrease a non-zero natural number by one, especially if `dec` takes 0, it returns 0. `_|_` is the constructor of non-empty queues of processes IDs. Given a state formalized as `{(queue: p1 | p2) (pc[p1]: ws) (pc[p2]: ws) (cnt: 2)}`, rule `wait` can be applied to change it as: `{(queue: p1 | p2) (pc[p1]: cs) (pc[p2]: ws) (cnt: 2)}`. Fig. 2.1 depicts the reachable state space made from S_{Qlock}, I_{Qlock}, and T_{Qlock}. The total number of states is 16 in the reachable state space.

Model Checking

Maude is equipped with an LTL model checker so that we can verify that a system satisfies a desired property by model checking. Let us consider the lockout freedom property for Qlock that informally states that whenever a process wants to enter its critical section, that process will eventually enter the critical section.

We consider an atomic proposition denoted `inWs1` and `inCs1`. Therefore, P_{Qlock} has `inWs1` and `inCs1`. L_{Qlock} is defined as follows:

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

`OCs` and `PROP` are Maude variables of observable component soups and atomic propositions. The definitions indicate that $L_{\text{Qlock}}(s)$ consists of `inWs1` if a state s has process `p1` located at ws and $L_{\text{Qlock}}(s)$ consists of `inCs1` if a state s has process `p1` located at cs otherwise, $L_{\text{Qlock}}(s)$ is empty.

We can check the lockout freedom property for Qlock denoted $K_{\text{Qlock}} \models \texttt{inWs1} \rightsquigarrow \texttt{inCs1}$ by reducing the following command:

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

`|->` is the Maude operator that expresses $\rightsquigarrow$. Because no counterexample is found, it concludes that Qlock satisfies the property when two processes are used.

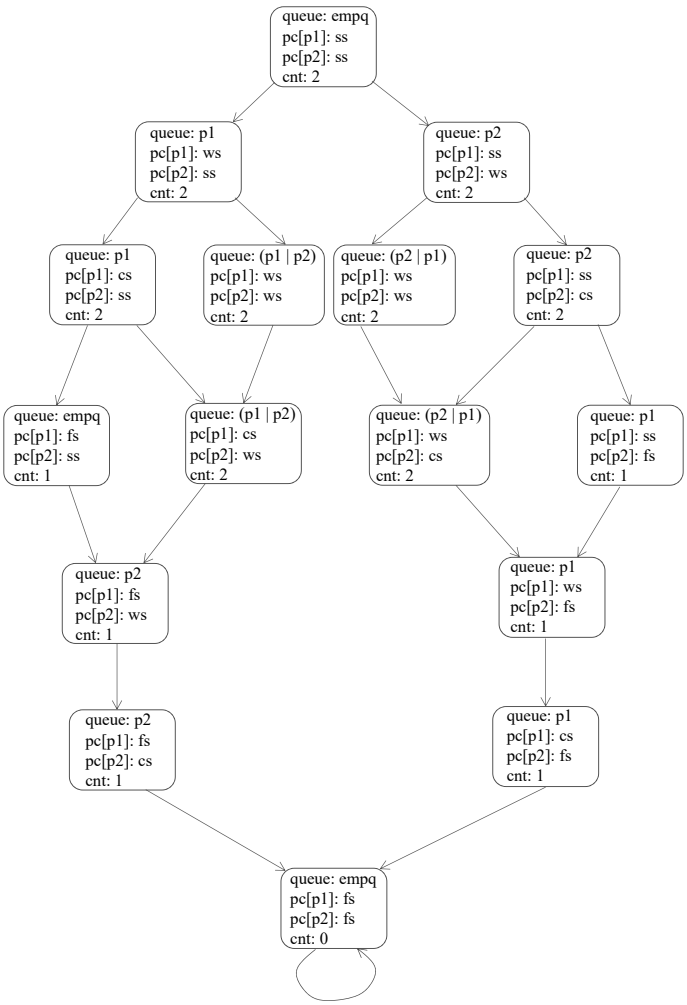

Figure 2.1: The reachable state space of Qlock

2.3.2 Meta-programming

Maude is a high-level language and a high-performance system [41] 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. We borrow some descriptions on them from [41]. 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, 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 meta-represented as elements of a data type `Term` of terms.
- in the module `META−STRATEGY`, the Maude strategy language is meta-represented as terms in a data type `Strategy` of strategy expressions.
- in the module `META−MODULE`, Maude modules are meta-represented as terms in a data type `Module` of modules.
- in the module `META−LEVEL`,
  - operations `upModule`, `upTerm`, `downTerm` and others allow moving between reflection levels;
  - the process of reducing a term to canonical form using Maude's `reduce` command is meta-represented by a built-in function `metaReduce`;
  - the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are meta-represented 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 meta-represented 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 meta-represented 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 meta-represented 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 meta-represented 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 at 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`.

In the module `META-TERM`, terms are meta-represented as elements of data type `Term` of terms. For the base cases, constants and variables in the meta-representation of terms are given by subsorts `Constant` and `Variable` of the sort `Qid`.

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .
op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .
```

where some attributed are omitted where `...` is written. Constants are quoted identifiers consisting of the constant's name and its type separated by a period (e.g. `'0.Nat`). Likewise, Variables consist of their name and type separated by a colon (e.g. `'N:Nat`).

A term different from a constant or a variable is meta-represented by using the following symbols:

```
sorts NeTermList TermList .
subsorts Term < NeTermList < TermList .
op _,_ : TermList TermList -> TermList
      [ctor assoc id: empty gather (e E) prec 121] .
op _,_ : NeTermList TermList -> NeTermList [ctor ditto] .
op _,_ : TermList NeTermList -> NeTermList [ctor ditto] .
op _[_] : Qid NeTermList -> Term [ctor] .
```

where `_,_` is the constructor of non-empty term lists whose sort is `TermList`, `_[_]` is the constructor of terms. A term list may contain only a term or a non-empty term list. Given a term at object-level, the function `upTerm` can change it to a meta-representation at meta-level. For example, the initial state `init` used in Qlock can be meta-represented at meta-level by calling the function `upTerm` and the result is as follows (denoted `init'`):

```
''{_`}['__['queue:_['empq.Queue`{Pid`}],'cnt:_['s_^2['0.Zero]],
       'pc`[_`]:_['p1.Pid,'ss.Loc],'pc`[_`]:_['p2.Pid,'ss.Loc]]]
```

where `‘{_‘}`, `’__` are quoted identifiers for the braces and the soup of observable components. `’queue:_`, `’cnt:_`, and `pc‘[_]‘` are quoted identifiers for observable components. `’empq.Queue‘{Pid‘}`, `’s_^2[’0.Zero]`, `’p1.Pid`, `’ss.Loc`, `’p2.Pid`, and `’ss.Loc` are constants specifying the values of observable components. Similarly, the meta-representation of the formula `inWs1 |-> inCs1` expressing the lockout freedom property is as follows:

```
'|->['inWs1.Prop,'inCs1.Prop]
```

Let `lofree` denote the whole term and `e-prop` denote the term `’inCs1.Prop`.

In the module `META-MODULE`, a module is meta-represented as a term whose type is `Module`. We assume that the formal specification of Qlock contains one module `QLOCK-CHECK` in which Qlock and the lockout freedom property are described. We can obtain the meta-representation of the module by using the following function:

```
upModule('QLOCK-CHECK, true) .
```

where `’QLOCK-CHECK` is a quoted identifier of the module `QLOCK-CHECK` so that Maude can find the module in its module database, true indicates that the meta-representation of the module includes the corresponding statements that are imported from its submodules. The result of the function is as follows:

```
mod 'QLOCK-CHECK is
  ...
  rl '‘{_‘}['__['OCs:Soup‘{OComp‘},'cnt:_['0.Zero]]] =>
  '‘{_‘}['__['OCs:Soup‘{OComp‘},'cnt:_['0.Zero]]] [label('fin)] .
  rl '‘{_‘}['__['OCs:Soup‘{OComp‘},'cnt:_['N:Nat],'pc‘[_‘]:_['I:Pid,'cs.Loc]]]
  => '‘{_‘}['__['cnt:_['dec['N:Nat]],'__['OCs:Soup‘{OComp‘},
            'pc‘[_‘]:_['I:Pid,'ws.Loc]]]] [label('flaw)] .
  rl '‘{_‘}['__['OCs:Soup‘{OComp‘},'queue:_['Q:Queue‘{Pid‘}],
            'pc‘[_‘]:_['I:Pid,'ss.Loc]]]
  => '‘{_‘}['__['queue:_['_|_['Q:Queue‘{Pid‘},'I:Pid]],'__['OCs:Soup‘{OComp‘},
            'pc‘[_‘]:_['I:Pid,'ws.Loc]]]] [label('start)] .
  rl '‘{_‘}['__['OCs:Soup‘{OComp‘},'queue:_['_|_['I:Pid,'Q:Queue‘{Pid‘}]],
            'pc‘[_‘]:_['I:Pid,'ws.Loc]]]
  => '‘{_‘}['__['queue:_['_|_['I:Pid,'Q:Queue‘{Pid‘}]],'__['OCs:Soup‘{OComp‘},
            'pc‘[_‘]:_['I:Pid,'cs.Loc]]]] [label('wait)] .
endm
```

where `...` denotes some parts omitted. The rewrite rules used in Qlock are meta-represented as terms at meta-level. Let `M’` denote the meta-representation of the module `QLOCK-CHECK`.

All Counterexample State Generation in Meta-programming

As a concrete example, we describe a meta-program in Maude that finds all counterexamples for a Kripke structure K and a property φ. Some constraints are given to K and φ. Each computation π of K is in the form $s_0, ..., s_{i-1}, s_i, s_i, ...$, namely that it is composed of a finite state sequence $s_0, ..., s_{i-1}$ and a singleton loop $s_i, s_i, ...$ in which a single state s_i repeats forever. φ is either a leads-to property $\varphi_1 \rightsquigarrow \varphi_2$ or a conditional stable property $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions. If $K, \pi \not\models \varphi$, then s_i is called a counterexample state in this thesis. It is necessary to prevent π from being found as a counterexample again so as to find all counterexamples. To this end, we add the following equation:

`eq` $s_i \models \varphi_2 =$ `true` .

Let K' be the Kripke structure obtained from K by adding the equation. Note that the labeling function of K' is different from the one of K. If so, φ is satisfied by K' and π, namely K', $\pi \models \varphi$. Then, if there is any other counterexample $s'_0, ..., s'_{j-1}, s'_j, s'_j, ...$ such that s'_j is different from s_i, then it can be found, and if there is no more counterexample, no counterexample is found.

We intend to insert the following rule (a flaw) into the formal specification of Qlock so that the lockout freedom property does not hold and some counterexamples are discovered:

```
rl [flaw] : {(queue: (I | Q)) (pc[I]: ws) (cnt: N) OCs}
   => {(queue: Q) (pc[I]: flaw) (cnt: dec(N)) OCs} .
```

A new location, namely flaw, is introduced to which a process can move. The rule named flaw means that when the top of queue is I whose location is ws, process I moves to flaw instead of cs; I is removed from queue; and N is decremented. Not that a process reached flaw is supposed to stay there forever.

We can see that K_{Qlock} does not contain any lasso loops in its specification and any path π of K_{Qlock} contains a self-transition at the end where all processes are located at either fs or flaw. In order to generate all counterexample states for QLOCK with the lockout freedom property at meta-level, we first model check whether $K_{Qlock} \models inWs1 \rightsquigarrow inCs1$ by the following command:

```
metaReduce(M', 'modelCheck[init', lofree])
```

The function `metaReduce` takes two parameters: (1) the meta-representation of a module and (2) a meta-representation of a term to be reduced with respect to (1). `'modelCheck[_,_]` is the term expressing for the model checking experiment that we want to conduct under `M'`, where underscores denote the positions of the meta-representation `init'` of the initial state and the meta-representation `F` of the formula. The function `metaReduce` conducts the model checking experiment and returns a term representing a counterexample as follows:

```
{'counterexample['__[
```

```
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’empq.Queue‘{Pid‘}],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ss.Loc],’pc‘[_‘]:_[’p2.Pid,’ss.Loc]]],’’start.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’p1.Pid],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ws.Loc],’pc‘[_‘]:_[’p2.Pid,’ss.Loc]]],’’start.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’_|_[’p1.Pid,’p2.Pid]],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ws.Loc],’pc‘[_‘]:_[’p2.Pid,’ws.Loc]]],’’flaw.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’p2.Pid],’cnt:_[’s_[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’flaw.Loc],’pc‘[_‘]:_[’p2.Pid,’ws.Loc]]],’’wait.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’p2.Pid],’cnt:_[’s_[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’flaw.Loc],’pc‘[_‘]:_[’p2.Pid,’cs.Loc]]],’’exit.Sort]
  ],
    ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’empq.Queue‘{Pid‘}],’cnt:_[’0.Zero],
    ’pc‘[_‘]:_[’p1.Pid,’flaw.Loc],’pc‘[_‘]:_[’p2.Pid,’fs.Loc]]],’’fin.Sort]
  ], ’ModelCheckResult}
```

The counterexample consists of one term representing a finite state sequence that starts with the initial state and the other one representing a finite state sequence as a loop. The loop consists of only one state because there is a self-transition that is taken infinitely many times. The state is called a counterexample state and extracted from the counterexample as follows:

```
’‘{_‘}[’__[’queue:_[’empq.Queue‘{Pid‘}],’cnt:_[’0.Zero],
       ’pc‘[_‘]:_[’p1.Pid,’flaw.Loc],’pc‘[_‘]:_[’p2.Pid,’fs.Loc]]]
```

Let `cx1` denote the first counterexample state. There may be more counterexample states. In order to collect the next one, we need to construct the following term whose sort is `Equation`:

```
eq ’_|=_[cx1, e-prop] = ’true.Bool [none] .
```

and add it to `M’` so as to ignore `cx1`. We conduct the model checking again and obtain the following counterexample:

```
{’counterexample[’__[
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’empq.Queue‘{Pid‘}],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ss.Loc],’pc‘[_‘]:_[’p2.Pid,’ss.Loc]]],’’start.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’p1.Pid],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ws.Loc],’pc‘[_‘]:_[’p2.Pid,’ss.Loc]]],’’start.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’_|_[’p1.Pid,’p2.Pid]],’cnt:_[’s_^2[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’ws.Loc],’pc‘[_‘]:_[’p2.Pid,’ws.Loc]]],’’flaw.Sort],
  ’‘{_‘,_‘}[’‘{_‘}[’__[’queue:_[’p2.Pid],’cnt:_[’s_[’0.Zero]],
    ’pc‘[_‘]:_[’p1.Pid,’flaw.Loc],’pc‘[_‘]:_[’p2.Pid,’ws.Loc]]],’’flaw.Sort]
  ],
```

```
  ''{_`,_`}['`{_`}['__['queue:_['empq.Queue`{Pid`}],'cnt:_['0.Zero],
    'pc`[_`]:_['p1.Pid,'flaw.Loc],'pc`[_`]:_['p2.Pid,'flaw.Loc]]],''fin.Sort]
  ],'ModelCheckResult}
```

Similarly, we extract the second counterexample state denoted `cx2` from the counterexample as follows:

```
''{_`}['__['queue:_['empq.Queue`{Pid`}],'cnt:_['0.Zero],
       'pc`[_`]:_['p1.Pid,'flaw.Loc],'pc`[_`]:_['p2.Pid,'flaw.Loc]]]
```

We then construct a term whose sort is `Equation`, add it to `M'`, and keep on conducting the model checking experiment again. However, the function `metaReduce` returns as follows:

```
{'true.Bool,'Bool}
```

It means that there are no more counterexamples found. In summary, we can collect two counterexample states `cx1` and `cx2` by using meta-programming facilities in Maude. Note that all counterexample states are meta-represented as terms at meta-level.

What we described above can be automated by the following function in meta-programming:

```
ceq gen-cx-states(M, T, F, P)
= if (CX :: Constant)
  then empty-term-set
  else get-cx-state(CX) |
       gen-cx-states(add-eqs(build-eqs(get-cx-state(CX), P), M), T, F, P)
  fi
if CX := get-counter-example(M, 'modelCheck[T, F]) .
```

where `M` is a Maude variable whose sort is `Module` and `CX`, `T`, `F`, and `P` are Maude variables whose sorts are `Term`. The function returns a set of terms, whose sort is `TermSet`, presenting all counterexample states. We use the function `get-counter-example` to obtain a counterexample `CX` by carrying out the model checking experiment `'modelCheck[T, F]`. If `CX` is a constant, there is no more counterexample state; otherwise, we obtain the state in the loop of `CX` as a counterexample state by calling function `get-cx-state`, construct an equation and add it to the formal specification `M` to ignore `CX` by calling functions `build-eqs` and `add-eqs`, respectively, and continue calling function `gen-cx-states` to collect the next one. For Qlock example, the function `gen-cx-states` takes `M'`, `init'`, `lofree`, and `e-prop` as parameters to automate generating all counterexample states.

2.4 State Space Explosion

The number of states in the reachable state space of a system can be enormous. For example, a system has n processes and each process has m states. Then, the combination of these processes may produce m^n states, meaning that the number of states is exponential in the number of processes. In model checking, we call this problem the state space explosion. All model checkers suffers from this problem because they need to explore the entire (reachable) state space to verify that a system satisfies its desired properties.

For Qlock, when two processes are used, the Maude LTL model checker can quickly complete the model checking experiment and concludes that Qlock satisfies the lockout freedom property. However, if we increase the number of processes participating in Qlock, the number of states in the reachable state space grows dramatically. For example, when we increase the number of processes participating in Qlock to 9 processes and conduct the model checking experiment with 2GB memory restricted, the Maude LTL model checker could not complete the model checking experiment and ran out of memory because of the state space explosion problem. In other words, 2GB of memory was not enough for the model checking experiment. We could solve the problem for Qlock by using the $L + 1$ divide & conquer approach to leads-to model checking that is described in Chapters 4 and 6.

Chapter 3

Related Work

The state space explosion is one of the most notorious problems in model checking. Many researchers have tackled the problem and come up with several techniques to mitigate it. In this chapter, we mention some classical techniques, such as partial order reduction, binary decision diagrams (BDDs), and symmetry reduction. Moreover, we describe some other techniques, such as bounded model checking based on satisfiability solvers or satisfiability modulo theory solvers (SAT/SMT-based BMC) and its extensions, abstraction and abstraction refinement, and non-exhaustive model checking. We also compare some existing techniques with our divide & conquer approach to model checking ($L + 1$-DCA2MC) leads-to and conditional stable properties.

3.1 Some Classical Techniques

Binary decision diagrams (BDDs) are a data structure for representing Boolean formulas in a canonical and symbolic way. Many efficient algorithms as well as libraries have been developed to support manipulating Boolean operators with BDDs (e.g. AND, OR, and XOR operations). A binary decision diagram represents a Boolean formula as a rooted acyclic graph in which each node except for leaves represents a variable and has two branches denoting 0 (true) and 1 (false) assigned to the variable, respectively. There are two kinds of leaves with labels 0 and 1, respectively. A path from the root node to a leaf represents an evaluation of the Boolean formula in which the values of variables are their branches selected in the path and the evaluation is the label in the leaf. Note that the graph can be reduced into a compact graph and the order of variables affects the size of the graph [42]. In model checking, BDDs are one approach to encoding sets of states and the state transition relation from which a compressed representation of a state transition graph is constructed. Model checking that uses BDDs may be called symbolic model checking. By using an original model checking algorithm with the new representation of a state transition graph, it is possible to handle a system with more than 10^{20} states [6].

Symmetric reduction [8] is a technique to reduce the state space of finite state systems by identifying sub-state spaces that have replicated structures from which redundant structures are removed because of their symmetry. For example, the state space of Qlock with two processes depicted in Fig. 2.1 contains two replicated structures derived from the initial state because at the beginning each process has an equal chance to move from ss to ws. Hence, it is enough to investigate one structure because of its symmetry. A finite state system is formalized as a Kripke Structure denoted $K = (S, I, T, A, L)$. Let G be a group of permutations in which each permutation is a bijective mapping on S. For example, a permutation σ on a set of states $\{s_0, s_1, s_2\}$ is defined as $\sigma(s_0) = s_2$, $\sigma(s_2) = s_3$, and $\sigma(s_3) = s_1$. A permutation $\sigma \in G$ is called a *symmetry* of K if and only if $\forall s_1, s_2 \in S, (s_1, s_2) \in T \Rightarrow (\sigma(s_1), \sigma(s_2)) \in T$. G is a *symmetric* group if and only if every permutation σ is a symmetry of K. Let $\theta(s)$ be the *orbit* of s denoting a set of states by applying each of the permutations in G on s. For each orbit $\theta(s)$, we select only one state from $\theta(s)$ denoted $rep(\theta(s))$ as a representative of $\theta(s)$. If G is a symmetric group on K, a quotient model $K_G = (S_G, I_G, T_G, A, L_G)$ is constructed, where S_G is the set of orbits of the states in S, I_G is the set of orbits of the initial states in I, T_G is the state transition relation such that $(\theta(s_1), \theta(s_2)) \in T_G$ if and only if there exists $s_3 \in \theta(s_1)$ and $s_4 \in \theta(s_1)$ and $(s_3, s_4) \in T$, L_G is the labeling function such that $L_G(\theta(s)) = L(rep(\theta(s))$. G is an *invariant* group of K for an atomic proposition p if and only if $\forall \sigma \in G, \forall s \in S, p \in L(s) \Leftrightarrow p \in (L(\sigma(s)))$. They have proved that if a formula f is expressed in the temporal logic CTL* and G is an invariant group for all atomic propositions in f, then f holds in K if and only if f holds in K_G. Because the quotient model K_G selects only one element from each orbit, then the state space of K_G is smaller than that of K, by which the state space explosion is mitigated. Symmetric reduction can work well on systems that have replicated structures while $L + 1$-DCA2MC can work well on systems that have less symmetric structures. Hence, our technique can be used as a complementary technique to reduce the state space explosion further after symmetric reduction has been applied.

A sequential system can be modeled as a sequence of states such that the pair (s, s') of each adjacent states is a transition. An asynchronous concurrent system consists of multiple active entities, such as processes, threads and tasks. A concurrent system can be basically modeled as the set of all interleavings of such sequences generated by the active entities in the system. The set may be represented as a graph constructed from a Kripke structure formalizing the system. The set could be huge and so could the graph. A Kripke structure is written in a language, such as a formal specification language. We suppose that transitions are classified into a finite number of classes such that each class is given a name, such as α. Such a name, such as α, is referred to as a transition function. Given a state s, let $\alpha(s)$ be the successor state of s with respect to α. In general, however, α has a condition called an enabled condition. If the enabled condition of α holds in s, then $(s, \alpha(s))$ is a transition. For a state s, let $enabled(s)$ be the set of all transition functions whose enabled conditions hold in s. If we could use another set of transitions that is smaller than $enabled(s)$ or a proper subset of $enabled(s)$ so

as to model check properties in interest, we would reduce the graph constructed from a Kripke structure, which may alleviate the state space explosion problem. One such set is an ample set denoted $ample(s)$ for a state s [9]. It is necessary to define two concepts of transition functions (dependency and invisibility) in order to find out $ample(s)$. Independency of two transition functions α, β is as follows: the execution of α (or β) never disables β (or α) and α and β are commutative, namely that $\alpha(\beta(s)) = \beta(\alpha(s))$ for a state s in which both α and β are enabled. If α and β are not independent, they are dependent. Let L and A be the labeling function and the set of atomic propositions of a Kripke structure in interest. A transition function α is invisible with respect to a subset A' of A if and only if for each pair s, s' of states such that $s' = \alpha(s)$, $L(s) \cap A' = L(s') \cap A'$. There are four conditions so as to find out $ample(s)$. C0: $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$ for each state s. C1: along each path in the original graph (constructed by using $enabled(s)$) that starts at s, a transition that is dependent on a transition function in $ample(s)$ cannot be executed without one in $ample(s)$ being executed first. C2: each transition function in $ample(s)$ for each s is invisible. C3: a cycle is prohibited if it has a state in which there exists a transition that is enabled but not included in $ample(s)$ for any state s on the cycle. The original graph constructed by using $enabled(s)$ and the reduced graph constructed by using $amble(s)$ are stuttering equivalence. Properties written in LTL from which the next temporal connective is deleted are preserved between the original graph and the reduced graph.

3.2 SAT/SMT-based Bounded Model Checking and its Extensions

One effective technique to ease the problem and is related to $L+1$-DCA2MC is bounded model checking (BMC) [28] based on solvers of satisfiability (SAT) or satisfiability modulo theories (SMT), or SAT/SMT-based BMC. SAT/SMT-based BMC converts a BMC problem into a SAT/SMT problem and tackles the former problem by solving the latter one with a SAT/SMT solver, where a BMC problem is to model check a property for a bounded reachable state space up to a finite depth from each initial state. $L+1$-DCA2MC generates multiple sub-state spaces from the reachable state space from each initial state, where sub-state spaces obtained from non-final layers are bounded state spaces, although each state placed at the bottom of the sub-state spaces has a self-transition. Therefore, $L + 1$-DCA2MC can be considered an extension of SAT/SMT-based BMC, but never uses any SAT/SMT solvers.

Another extension of SAT/SMT-based BMC is k-induction [29, 30]. k-induction first carries out BMC for the bounded state space from each initial state up to a finite depth k, which corresponds to the base case of the standard induction. It then supposes that a property under verification is satisfied for each state sequence from an arbitrary state s such that its depth is k or it consists of $k + 1$ states, which corresponds to the induction hypothesis of the standard

induction. It finally checks if the property remains true in any successor states of the state sequence, where the successor states are placed at depth $k+1$ from s, which corresponds to the induction case (or step) of the standard induction. Properties that can be handled by k-induction are invariant properties. $L+1$-DCA2MC does not use any inductions but uses case splittings that can be regarded as a specific instance of induction. $L+1$-DCA2MC exhaustively splits the reachable state space from each initial state into multiple sub-state spaces.

Yet another extension of SAT-based BMC is model checking with Craig interpolants [31]. The formula tackled by SAT-BMC with depth k is divided into two sub-formulas A and B, where A expresses each initial state and one-step transition from each initial state and B expresses the other one-step transitions up to depth k and what is related to a property under model checking. If $A \wedge B$ is unsatisfiable, there exists an interpolant P such that $P \wedge B$ is unsatisfiable and each variable in P is included in A and B. Such P is constructed by SAT-BMC. $A \wedge P$ over-approximates the set of states that are reachable from each initial state up to one transition. A is replaced with $A \wedge P$, and then it is checked that $A \wedge B$ is unsatisfiable. If so, $A \wedge P$, where P is the next interpolant obtained, over-approximates the set of states that are reachable from each initial state up to two transitions. This process is repeated at most $k-1$ times until $A \wedge B$ becomes satisfiable or $A \wedge P$ is equivalent to A. If $A \wedge B$ becomes satisfiable meanwhile, it means that a counterexample is found. If $A \wedge P$ is equivalent to A, the property is satisfied. Otherwise, k is increased and then the whole process is repeated. Model checking with Craig interpolants is unbounded model checking in that unboundedly many transitions are taken into account instead of boundedly many transitions, and so is $L+1$-DCA2MC.

Another extension of SAT/SMT-based BMC to model check concurrent programs is Lazy Sequentialization (Lazy-CSeq) [32, 33]. 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. Lazy-CSeq seems to check safety properties only, while $L+1$-DCA2MC focuses on liveness properties.

3.3 Abstraction and Abstraction Refinement

Abstraction [20] is one of the most widely used in practice to mitigate the state space explosion. It starts from a concrete model M and produces an abstract model $\widehat{M}$ in which only certain information is retained while the other information in M is abstracted away. The state space of $\widehat{M}$ is usually smaller than that of M, making it easier to model check desired properties on $\widehat{M}$. However, it is non-trivial to construct directly $\widehat{M}$ from M, which often requires a series

of refinement steps. For example, when an invariant property holds in $\widehat{M}$, we can conclude that it also holds in M. However, if the invariant property does not hold in $\widehat{M}$, we cannot conclude anything yet because the counterexample may come from $\widehat{M}$ but not M. In that case, abstraction refinement is required to eliminate the counterexample from $\widehat{M}$. One approach to the construction is counterexample-guided abstraction refinement (CEGAR) [21]. For a system model whose (reachable) state space is enormous, CEGAR starts with one abstract model whose (reachable) state space is reasonably small such that it can be tractable by a model checker. If a counterexample is found, CEGAR checks if the counterexample is real. If no, CEGAR refines the abstract model based on the counterexample and repeats the experiment for the refined model until a real counterexample is found or no counterexample is found for some refined model (including the initial abstract model). It may be the case that the (reachable) state space of some refined model becomes enormous and then it will be infeasible to do model checking for the model.

It has been demonstrated that BMC is a useful technique to detect errors lurking in concurrent programs, such as multi-threaded ones because most errors reside in small loop unwinding depths. Although BMC is a technique to alleviate the state space explosion problem in model checking, a huge formula that is supposed to be tackled by a SAT/SMT solver is derived from a concurrent program in which function calls (except for those creating threads) are inlined and loops are boundedly unwound. This is mainly because of the scheduling constraint caused by the complex inter-thread communication. Let the original program refer to a multi-threaded program in which non-thread creating function calls are inlined and loops are boundedly unwound. The scheduling constraint is that for any pair $< w, r >$ of operations such that r reads the value stored in a variable v that has been written by w, any other write operations to the variable v should not be carried out between w and r. Such a formula should be made smaller so as to make BMC more practical. To address the challenge, Yin, et al. propose a scheduling constraint based abstraction refinement (SCAR) [22, 23]. SCAR initially neglects the scheduling constraint and gets an over-approximation abstraction of the original program. If the property being interested holds for the abstraction, so does for the original program. Otherwise, SCAR confirms that a counterexample found is real. If not, it refines the abstraction by taking the scheduling constraint into account to some extent, obtaining a refined version of the abstraction that is another over-approximation abstraction of the original program. This process repeats until the property holds for an abstraction or a real counterexample is found. To make both the number of the process iterations and the size of each refinement added moderately small, Yin, et al. came up with two graph-based algorithms over event order graph, where events are read and write events, so as to efficiently confirm that a counterexample is real and obtain a reasonably small refinement in each refinement iteration. They conducted experiments, demonstrating their tool outperforms Lazy-CSeq-Abs [24], a leading tool for model checking concurrent programs in the concurrency track of SV-COMP 2017.

Meseguer, Palomino and Martí-Oliet [25] came up with equational abstraction that is an

abstraction technique that can be used for model checking problems based on rewrite-theory formal systems specifications. In rewrite-theory formal systems specification, rewrite rules are used to describe state transitions and equations are used to define functions over data structures. Equations can also be used to make two different data values, such as two different states, equal, being able to make an infinite or huge number of different reachable states in a rewrite-theory formal systems specification a reasonably small finite number. This is a basic idea of equational abstraction. It is necessary to prove that whenever a property in interest holds for the abstraction obtained from a rewrite-theory formal systems specification by equational abstraction, the property also holds for the original rewrite-theory formal systems specification. Therefore, equational abstraction is regarded as an integration of theorem proving and model checking.

A logical model checking [26] based on rewrite-theory formal systems specification has been proposed by Bae, Escobar and Meseguer. In the logical model checking, equational abstraction can/should be used. States that can be handled by the standard model checking should be very concrete and should not contain any logical variables. The logical model checking makes it possible for state expressions or terms to have logical variables. State terms that have logical variables denote an infinite number of states. The logical model checking uses narrowing that denotes state transitions instead of reduction of rewriting. One-step rewriting from a concrete state term to another concrete state term represents one-step concrete state transition, while one-step narrowing from a state term in which variables are used to another state term in which variables are used represents a possibly infinite multiple one-step state transitions. In this sense, the logical model checking may make it possible to tackle rewrite-theory formal systems specifications that may have an infinite number of reachable states. To make such model checking practically usable, equational abstraction may have to be first used before the utilization of the logical model checking.

Predicate abstraction is one of the most widely used abstraction techniques because of its simplicity. Let $\mathcal{S}$ be a system and AP be a set of state predicates of $\mathcal{S}$. The abstract system $\mathcal{S}/AP$ of $\mathcal{S}$ is defined as follows: (1) the set of the abstract system states (called abstract states) is 2^{AP}, the power set of AP, and (2) there exists a transition (called an abstract transition) $s \to s'$ of $\mathcal{A}/AP$, where $s, s' \in 2^{AP}$, if and only if there exists a concrete transition $t \to t'$ of $\mathcal{S}$ such that t satisfies all state predicates in s and t' satisfies all state predicates in s'. It is non-trivial to make sure that (2) holds. Bae and Meseguer [27] have proposed a technique to automatically build a predicate abstraction of $\mathcal{S}$ when $\mathcal{S}$ is formalized as a rewrite-theory formal systems specification and AP is a set of state propositions defined in terms of equations. For each concrete transition $t \to t'$ of $\mathcal{S}$, there exists a rewrite rule $l \to r$ **if** C and a substitution σ such that $t =_E \sigma(l)$, $t' =_E \sigma(r)$, and $\sigma(u) =_E \sigma(v)$ for all conjunctions $u = v$ of C, where $=_E$ is equality modulo theory E (called E-equality), such as associativity and/or commutativity. The three E-equality conditions may be checked by E-unification, unification modulo theory E. s is $\{p \in AP \mid (t \models p) =_E true\}$, the set of state predicates satisfied by t, and s' is

$\{p \in AP \mid (t' \models p) =_E true\}$, the set of state predicates satisfied by t'. Then, $s \rightarrow s'$ is an abstract transition of $\mathcal{S}/AP$. This is how $\mathcal{S}/AP$ may be automatically obtained from $\mathcal{S}$ and AP. There may be a case such that the three E-equality conditions cannot be checked by E-unification. If that is the case, an over-approximation $\alpha(S/AP)$ is obtained by adding $s \rightarrow s'$ to $\mathcal{S}/AP$. Bae and Meseguer have proposed some procedures to confirm that if an LTL property φ holds for $\alpha(S/AP)$, so does for $\mathcal{S}/AP$.

3.4 Non-exhaustive Model Checking

SPIN [34] is one popular model checker used for verifying a variety of systems. Many techniques have been devised to deal with the state space explosion in SPIN, such as bit-state hashing and swarm verification. To tackle a large system that cannot be handled by an exhaustive verification mode, SPIN has a bit-state hashing verification mode [35] that may not exhaustively search the entire reachable state space of a large system but can achieve a higher coverage of large state spaces by using a few bits of memory per state stored. The idea of bit-state hashing derives from the standard hashing technique with linked lists used. Let us assume that there are R reachable states in a system under verification, S bytes are used to store a state, and M bytes of memory available are allowed to use. Regarding the standard hashing with linked lists, there are H slots in a hash table and each slot refers to a linked list where states can be stored as a chain. If R is much greater than H denoted $R \gg H$, then each slot will refer to a linked list that may contain R/H states on average. The total memory needed is at least $R \times S$. If R is very large and M is insufficient, it is impossible to conduct a model checking experiment. Regarding bit-state hashing, if $R \gg H$, each slot will refer to at most one state, so it is enough to use only one bit to record the existence of the state. From this observation, SPIN uses a large bit-state array to avoid state duplication. Initially, all values in the array are zero. Each state will be hashed by one or more hash functions depending on the number of hash functions selected from users and each hash value is associated with an index in the array. Let k be the number of hash functions. The state collision happens when k bit-positions in the bit-state array are one. The use of multiple hash functions is likely to avoid the hash collision. However, the state collision is still inevitable and in that case SPIN just ignores them. When using bit-state hashing, the order of states visited also affects the order of states stored in the bit-state array. Therefore, there are three essential parameters that impact the bit-state hashing: the size of the bit-state array, the number of hashing functions, and the search strategy. The advantage of the technique is that even with a small amount of memory it can achieve a higher coverage of large state spaces. However, the larger a system under verification becomes the higher chances the SPIN bit-state verification mode may overlook flaws lurking in the system.

Conducting bit-state verification, we can start with a small bit-state array size to try to find out a counterexample. If there is no counterexample, we can increase the bit-state array size

and conduct the verification again until the number of states that are explored stops increasing, meaning that all states are explored, or a counterexample is found. If we find a counterexample in the last run, the iterative process is very time-consuming compared with running the final run only. To overcome this situation, Swarm Verification [36] has been proposed. The key ideas of Swam Verification are parallelism and search diversity. For each of multiple different search strategies in which different seeds are used for search randomization, one instance of bit-state verification is conducted in which different hash-polynomials and numbers of hash functions are also used for diversification options. Each instance is supposed to use a small bit-state array size so that its verification can be completed in a reasonable amount of time. These instances are totally independent and can be conducted in parallel. Different search strategies with diversification options traverse different portions of the entire reachable state space, making it more likely to achieve higher coverage of the entire reachable state space and quickly find flaws lurking in a large system if any. $L+1$-DCA2MC splits the reachable state space from each initial state into multiple layers, generating multiple sub-state spaces. It then exhaustively searches each sub-state space with the Maude LTL model checker instead of using the idea of bit-state hashing used in each instance of Swarm Verification. However, 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. $L+1$-DCA2MC can be naturally parallelized because multiple model checking experiments for multiple sub-state spaces in the final layer are totally independent. However, parallelization of $L+1$-DCA2MC is out of the scope of this thesis.

Chapter 4

A Divide & Conquer Approach to Leads-to Model Checking

This chapter proposes a new technique to mitigate the state explosion in model checking. The technique is called a divide & conquer approach to leads-to model checking (L+1-DCA2L2MC). As indicated by the name, the technique is dedicated to leads-to properties. It is known that many important systems requirements can be expressed as leads-to properties and then it is worth focusing on leads-to properties. The technique divides an original leads-to model checking problem into multiple smaller model checking problems and tackles each smaller one. We prove a theorem that the multiple smaller model checking problems are equivalent to the original leads-to model checking problem. An algorithm is constructed based on the theorem to support model checking leads-to properties by our technique.

4.1 Outline of the Technique

L+1-DCA2L2MC splits each infinite state sequence $s_0^1, \ldots, s_{n_1}^1, \ldots, s_0^i, \ldots, s_{n_i}^i, \ldots, s_0^m, \ldots$ (generated from a Kripke structure) into multiple (say m) sub-sequences, for each $s_0^i, \ldots, s_{n_i}^i$ (for $i = 1, \ldots, m-1$) of all the sub-sequences except for the final one we add the final state $s_{n_i}^i$ infinitely many times to the end, generating the infinite sequence $s_0^i, \ldots, s_{n_i}^i, s_{n_i}^i, \ldots$ and conduct a model checking experiment for each infinite sequence. Note that the number of different states in the original infinite state sequence (and also each sub-sequence including the final one) is bounded if there is a bounded number of reachable states. If the number of different states in each sub-sequence is much smaller than the one in the original infinite state sequence, it would be feasible to conduct the model checking experiment for the infinite sequence generated from each sub-sequence (including the final one) even though it is impossible to conduct the model checking experiment for the original infinite state sequence due to the state explosion. The proposed technique makes it possible to use any existing linear temporal logic (LTL) model checking algorithm and then any existing LTL model checker. We prove a theorem that

the original leads-to model checking problem for the original infinite sequence is equivalent to the multiple model checking problems for the multiple infinite sequences. We design an algorithm based on the theorem from which a support tool is developed and some experiments are conducted to demonstrate the power of the technique in Chapter 6.

We use a test&set mutual exclusion protocol (TAS) as an example to outline the technique. 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 yet reached fs. Initially, b is `false`, each l_i is `ss` and x is n.

In this example, we suppose that there are two processes `p1` and `p2` participating in TAS. The initial state (referred 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. 4.1 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 example. 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] .
```

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

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

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

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

Although we do not need to use the proposed technique to tackle the model checking experiment, we use it to outline the $L+1$-layer divide & conquer approach to leads-to model checking. We split the reachable state space as shown in Fig. 4.1 to three layers such that the first layer depth is 2 and the second layer depth is 2 as shown in Fig. 4.2. Note that we do not need to specify the final layer depth (the third layer depth for the example used). Fig. 4.2 (a) shows the first layer, Fig. 4.2 (b) shows the second layer and Fig. 4.2 (c) shows the third layer. We have the 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

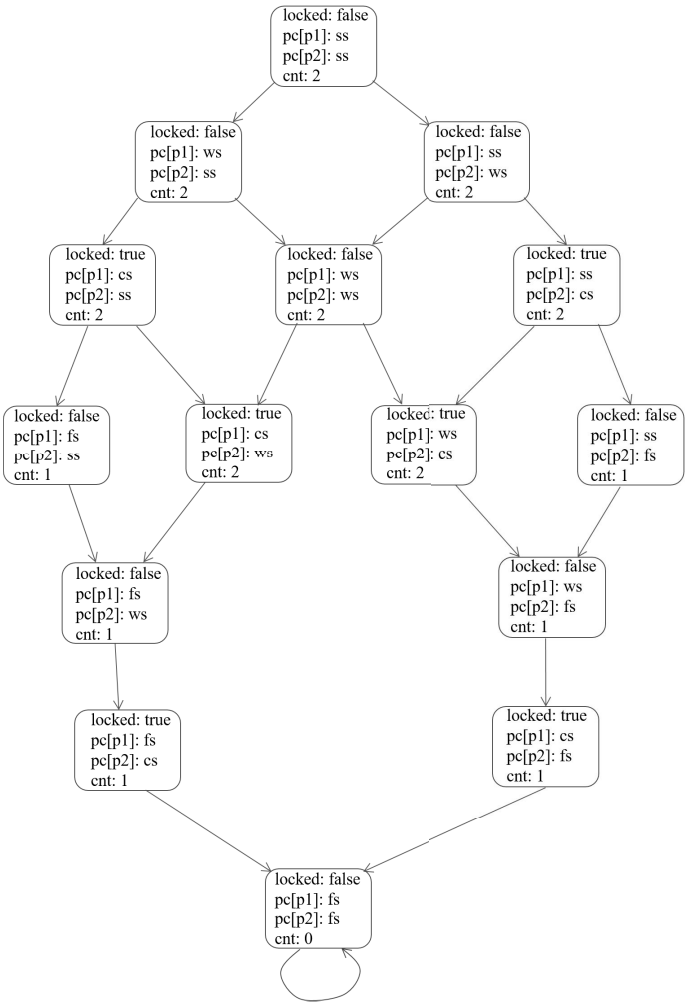

Figure 4.1: The reachable state space of TAS

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. That is, the number of the states in each sub-state space is less than the one in the whole reachable state space, which is the key to alleviate the state space explosion problem.

For model checking experiments for the first layer and the second layer, we need to revise the Maude specification of TAS. Each state expression is revised as follows:

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

The `depth` observable component has been added and is used to maintain the depth information d. The rewrite rules are revised as follows:

```
crl [start] : {(pc[I]: ss) (depth: D) OCs}
  => {(pc[I]: ws) (depth: (D + 1)) OCs} if D < Bound .
crl [wait] : {(locked: false) (pc[I]: ws) (depth: D) OCs}
```

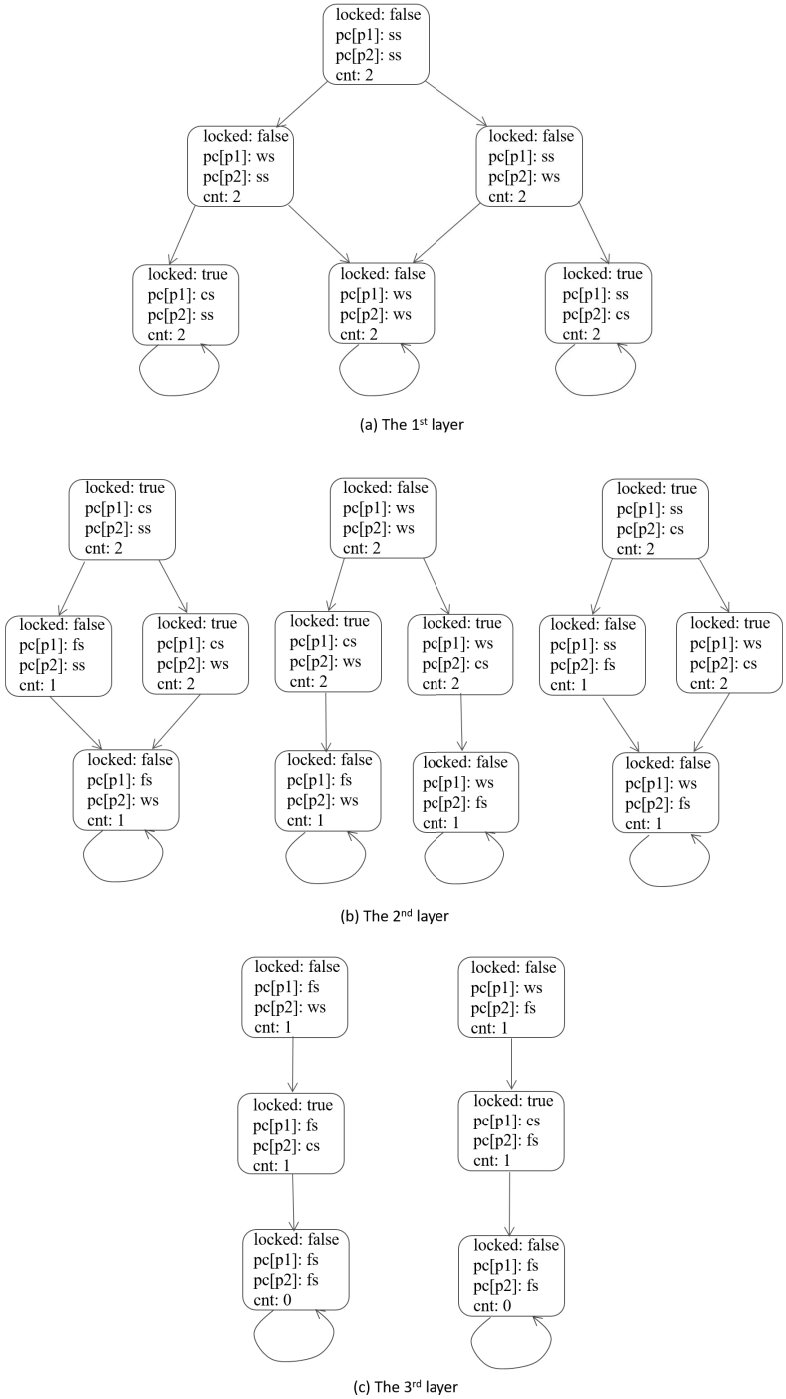

Figure 4.2: Three layers of TAS reachable state space

```
   => {(locked: true) (pc[I]: cs) (depth: (D + 1)) OCs} if D < Bound .
crl [exit] : {(locked: B) (pc[I]: cs) (cnt: N) (depth: D) OCs}
   => {(locked: false) (pc[I]: fs) (cnt: dec(N)) (depth: (D + 1)) OCs} if D < Bound .
crl [fin] : {(cnt: 0) (depth: D) OCs}
   => {(cnt: 0) (depth: (D + 1)) OCs} if D < Bound .
crl [stutter] : {(depth: D) OCs} => {(depth: D) OCs} if D >= Bound .
```

where `D` is a Maude variable of natural numbers and `Bound` is a natural number. `Bound` is `2` for the TAS case.

Let `init` be `{(locked: false) (pc[p1]: ss) (pc[p2]: ss) (cnt: 2) (depth: 0)}`. We are supposed to gather all states located at depth 2 from `init` and all states located at depth 2 to which there are counterexamples for the leads-to property concerned. The latter states are called counterexample states. We first check if `inWs1` $\rightsquigarrow$ `inCs1` holds for the first layer to find counterexample states by reducing the following term:

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

The Maude LTL model checker finds a counterexample. The counterexample state found is as follows:

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

There may be two or more counterexamples. To find them, we add the following equation so as to ignore the first counterexample:

```
eq {(locked: false) (cnt: 2) (depth: 2) (pc[p1]: ws) (pc[p2]: ws)} |= inCs1 = true .
```

We then conduct the model checking experiment again to find the second counterexample. The Maude LTL model checker, however, does not find any more counterexamples. Therefore, we gather all states located at depth 2. There are three such states that are as follows:

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

The three states are referred as `init3`, `init4` and `init5`, respectively. `init3` is the counterexample state.

For the first layer, we find three states `init3`, `init4` and `init5` located at depth 2 among which there is one counterexample state `init3`. Those states are used as initial states for the second layer. We need to update `Bound` to `4` for the second layer. For the three initial states `init3`, `init4` and `init5`, we check `inWs1` $\rightsquigarrow$ `inCs1` for the second layer. Moreover, for the one initial state `init3` that is the counterexample state for the first layer, we check $\Diamond$ `inCs1` for the second layer as well. The following four model checking experiments are conducted for the second layer:

```
modelCheck(init3, inWs1 |-> inCs1)
modelCheck(init4, inWs1 |-> inCs1)
modelCheck(init5, inWs1 |-> inCs1)
modelCheck(init3, <> inCs1)
```

where `<>_` is the Maude operator that expresses $\Diamond$. The first , third and fourth model checking experiments find one counterexample, respectively. There are two states located at depth 4 reachable from `init3`. There is one state located at depth 4 reachable from `init4`. There is one state located at depth 4 reachable from `init5`. The four states are referred as `init10`, `init10'`, `init11`, `init11'`, respectively, although `init10` is the same as `init10'` and `init11` is the same as `init11'`. Hence, there are actually two different states located at depth 4. Each of the three counterexample states is `init11` and then there is actually one counterexample state located at depth 4. The two states `init10` and `init11` are as follows:

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

We use the two states `init10` and `init11` from which `(depth: 4)` is deleted as the initial states in the third layer. The two initial states in the third layer are also referred as `init10` and `init11`, respectively. We use the original Maude specification of TAS for the third layer. We conduct the following model checking experiments for the third layer with the Maude LTL model checker:

```
modelCheck(init10, inWs1 |-> inCs1)
modelCheck(init11, inWs1 |-> inCs1)
modelCheck(init11, <> inCs1)
```

The Maude LTL model checker does not find any counterexamples for the three model checking experiments. Hence, we conclude that TAS enjoys the lockout freedom property when there are two processes by using our technique.

4.2 Multiple Layer Division of Leads-to Model Checking

We first prove in Theorem 4.1 that a leads-to model checking problem for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of three sub-model checking problems for $\boldsymbol{K}$ and two paths of $\boldsymbol{K}$, where the two paths are obtained by splitting π into two parts. We then prove in Theorem 4.1 that a leads-to model checking problem is equivalent to a model checking problem that is composed of $2L + 1$ sub-model checking problems for $\boldsymbol{K}$ and $L+1$ paths of $\boldsymbol{K}$, where $L \geq 1$ and the $L+1$ paths are obtained by splitting π into $L + 1$ parts.

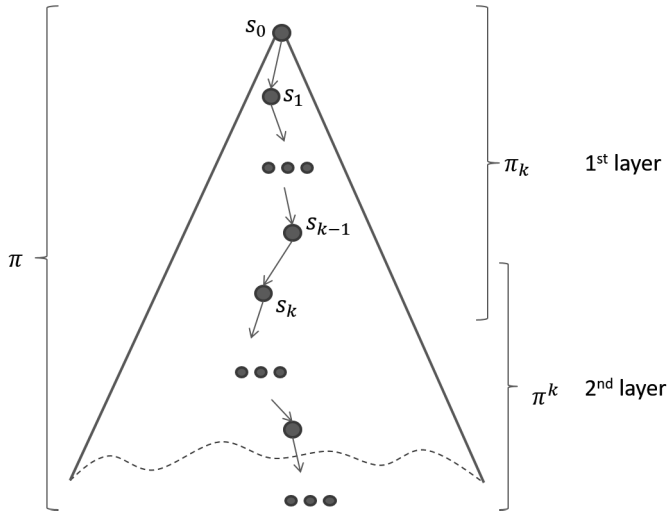

Figure 4.3: 2 layer division of the reachable state space

Let $\boldsymbol{K}$ be any Kripke structure, π be any path of $\boldsymbol{K}$ and φ_1 & φ_2 be any LTL formulas of $\boldsymbol{K}$ in this section unless otherwise stated. π is an arbitrarily chosen path (or computation) of $\boldsymbol{K}$ and then can be regarded as the whole reachable state space (see Fig. 4.3). Let π be split into two paths π_k and π^k, where π_k can be regarded as the first layer of the reachable state space and π^k can be regarded as the second layer of the reachable state space (see Fig. 4.3). We prove four lemmas from which Theorem 4.1 is derived.

The first lemma says that if $\varphi_1 \rightsquigarrow \varphi_2$ holds for the whole reachable state space, then it also holds for the second layer.

Lemma 4.1 *For any natural number k, $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2)$.*

Proof 4.1 *Let us suppose $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$. From the definition of $\rightsquigarrow$, for each $i \geq k$ such that $\boldsymbol{K}, \pi^i \models \varphi_1$, there exists $j \geq i$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$. Thus, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$.*

Note that $(\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2) \not\Rightarrow (\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2)$ because of the following. If there exists $i < k$ such that $\boldsymbol{K}, \pi^i \models \varphi_1$, there does not exist any $j \geq i$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$ and there does not exist any $i' \geq k$ such that $\boldsymbol{K}, \pi^{i'} \models \varphi_1$, then $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$ and $\boldsymbol{K}, \pi \not\models \varphi_1 \rightsquigarrow \varphi_2$.

The second lemma says that if $\varphi_1 \rightsquigarrow \varphi_2$ holds for the whole reachable state space and it does not hold for the first layer, then $\Diamond\varphi_2$ holds for the second layer, where $\varphi_1$and φ_2 are state propositions.

Lemma 4.2 *Let φ_1 & φ_2 be any state propositions of $\boldsymbol{K}$. For any natural number k, $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \wedge (\boldsymbol{K}, \pi_k \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^k \models \Diamond\varphi_2)$.*

Proof 4.2 *From the assumption $\boldsymbol{K}, \pi_k \not\models \varphi_1 \rightsquigarrow \varphi_2$, there exists $i \leq k$ such that $\boldsymbol{K}, \pi_k^i \models \varphi_1$ but does not exist any $j \geq i$ such that $\boldsymbol{K}, \pi_k^j \models \varphi_2$. Because φ_1 is a state proposition, $\boldsymbol{K}, \pi^i \models \varphi_1$ from Proposition 2.1. Because φ_2 is a state proposition, $\boldsymbol{K}, \pi^j \not\models \varphi_2$ for any j such that*

$i \leq j \leq k$ from Proposition 2.1. From the assumption $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ as well, however, there must exist $j' > k$ such that $\boldsymbol{K}, \pi^{j'} \models \varphi_2$. Thus, $\boldsymbol{K}, \pi^k \models \Diamond\varphi_2$.

The third lemma says that if $\Diamond\varphi_2$ holds for the second layer and $\varphi_1 \rightsquigarrow \varphi_2$ holds for the second layer, then $\varphi_1 \rightsquigarrow \varphi_2$ holds for the whole reachable state space.

Lemma 4.3 *For any natural number k, $(\boldsymbol{K}, \pi^k \models \Diamond\varphi_2) \wedge (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2)$.*

Proof 4.3 *We suppose that there exists i such that $\boldsymbol{K}, \pi^i \models \varphi_1$ holds. If $i \geq k$, because of the assumption $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$, there exists $j > i\ (> k)$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$ holds. If $i < k$, because of the assumption $\boldsymbol{K}, \pi^k \models \Diamond\varphi_2$, there exists $j \geq k\ (> i)$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$ holds.*

The fourth lemma says that if $\varphi_1 \rightsquigarrow \varphi_2$ holds for the first layer and $\varphi_1 \rightsquigarrow \varphi_2$ holds for the second layer, then $\varphi_1 \rightsquigarrow \varphi_2$ holds for the whole reachable state space, where φ_1 and φ_2 are state propositions.

Lemma 4.4 *Let φ_1 & φ_2 be any state propositions of $\boldsymbol{K}$. For any natural number k, $(\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2) \wedge (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2)$.*

Proof 4.4 *We suppose that there exists i such that $\boldsymbol{K}, \pi^i \models \varphi_1$ holds. If $i \geq k$, because of the assumption $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$, there exists $j > i\ (> k)$ such that $\boldsymbol{K}, \pi^j \models \varphi_2$ holds. If $i < k$, because φ_1 is a state proposition, $\boldsymbol{K}, (\pi_k)^i \models \varphi_1$ holds from Proposition 2.1. Thus, if $i < k$, because of the assumption $\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2$, there exists $j \geq i$ such that $\boldsymbol{K}, (\pi_k)^j \models \varphi_2$ holds. Because φ_2 is a state proposition, if $j \leq k, \boldsymbol{K}, \pi^j \models \varphi_2$ holds, and if $j > k\ (> i), \boldsymbol{K}, \pi^k \models \varphi_2$ holds from Proposition 2.1.*

We are ready to prove that a leads-to model checking problem $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of three sub-model checking problems $\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2$, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$ and $\boldsymbol{K}, \pi^k \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$ for $\boldsymbol{K}$ and the two paths π_k and π^k of $\boldsymbol{K}$, where k is a natural number.

Theorem 4.1 (Two layer division of $\rightsquigarrow$) *Let φ_1 & φ_2 be any state propositions of $\boldsymbol{K}$. For any natural number k, $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow ((\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge ((\boldsymbol{K}, \pi_k \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^k \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)))$.*

Proof 4.5 *(1) Case "only if" ($\Rightarrow$): It follows from Lemma 4.1 and Lemma 4.2. (2) Case "if" ($\Leftarrow$): If $\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2$, then it follows from Lemma 4.4. Otherwise, namely that if $\boldsymbol{K}, \pi_k \not\models \varphi_1 \rightsquigarrow \varphi_2$, then it follows from Lemma 4.3.*

Theorem 4.1 makes it possible to divide the original model checking problem $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ into the three sub-model checking problems $\boldsymbol{K}, \pi_k \models \varphi_1 \rightsquigarrow \varphi_2$, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2$ and $\boldsymbol{K}, \pi^k \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$.

We need to have two more lemmas to prove Theorem 4.2 whose description uses two abbreviations of statements on ternary satisfaction relations among Kripke structures, their paths and LTL formulas. The first one of the two lemmas says that an eventual model checking problem for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of two sub-model checking problems for $\boldsymbol{K}$ and two paths of $\boldsymbol{K}$, where the two paths are obtained by splitting π into two parts.

Lemma 4.5 *Let φ_1 be any state proposition of $\boldsymbol{K}$. For any natural number k, $(\boldsymbol{K}, \pi \models \Diamond\varphi_1) \Leftrightarrow ((\boldsymbol{K}, \pi_k \models \Diamond\varphi_1) \vee ((\boldsymbol{K}, \pi_k \not\models \Diamond\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^k \models \Diamond\varphi_1)))$.*

Proof 4.6 *(1) Case "only if" ($\Rightarrow$): There must be i such that $\boldsymbol{K}, \pi^i \models \varphi_1$. If $i \leq k$, $\boldsymbol{K}, \pi_k^i \models \varphi_1$ from Proposition 2.1 because φ_1 is a state proposition. Thus, $\boldsymbol{K}, \pi_k \models \Diamond\varphi_1$. Otherwise, $\boldsymbol{K}, \pi_k \not\models \Diamond\varphi_1$. However, $i > k$ and $\boldsymbol{K}, \pi^i \models \varphi_1$. Hence, $\boldsymbol{K}, \pi^k \models \Diamond\varphi_1$. (2) Case "if" ($\Leftarrow$): If $\boldsymbol{K}, \pi_k \models \Diamond\varphi_1$, there must be i such that $i \leq k$ and $\boldsymbol{K}, \pi_k^i \models \varphi_1$. Because φ_1 is a state proposition, $\boldsymbol{K}, \pi^i \models \varphi_1$ from Proposition 2.1 and then $\boldsymbol{K}, \pi \models \Diamond\varphi_1$. If $\boldsymbol{K}, \pi_k \not\models \Diamond\varphi_1$, then there must be j such that $j > k$ and $\boldsymbol{K}, \pi^j \models \varphi_1$. Thus, $\boldsymbol{K}, \pi \models \Diamond\varphi_1$.*

The second one of the two lemmas is on a model checking problem that tackles an eventual property and a leads-to property. Let us call such a model checking problem an eventual & leads-to model checking problem. The lemma says that an eventual & leads-to model checking problem for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of three sub-model checking problems for $\boldsymbol{K}$ and two paths of $\boldsymbol{K}$, where the two paths are obtained by splitting π into two parts.

Lemma 4.6 *Let φ_1 & φ_2 be any state propositions of $\boldsymbol{K}$. For any natural number k, $(\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow ((\boldsymbol{K}, \pi_k \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge ((\boldsymbol{K}, \pi_k \not\models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^k \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)))$.*

Proof 4.7 *It follows from Theorem 4.1 and Lemma 4.5.*

We define Leads2$_L$ and E&Leads2$_L$ that are abbreviations of statements on ternary satisfaction relations among Kripke structures, their paths and LTL formulas. To this end, let π be split into $L+1$ paths $\pi^{(d(0),d(1))}$ $(= \pi_{d(1)})$, $\ldots$, $\pi^{(d(l),d(l+1))}$, $\ldots$, $\pi^{(d(L),d(L+1))}$ $(= \pi^{d(L)})$, where $L \geq 1$, $d(0) = 0$, $d(L+1) = \infty$ and $d(l)$ is a non-zero natural number for $l = 1, \ldots, L$ (see Fig. 4.4). $\pi^{(d(l),d(l+1))}$ can be regarded as the $l+1$st layer of the reachable state space for $l = 0, 1, \ldots, L$.

Definition 4.1 (Leads2$_L$ **and** E&Leads2$_L$) *Let L be any non-zero natural number, k be any natural number and d be any function such that $d(0)$ is 0, $d(x)$ is a non-zero natural number for $x = 1, \ldots, L$ and $d(L+1)$ is ∞.*

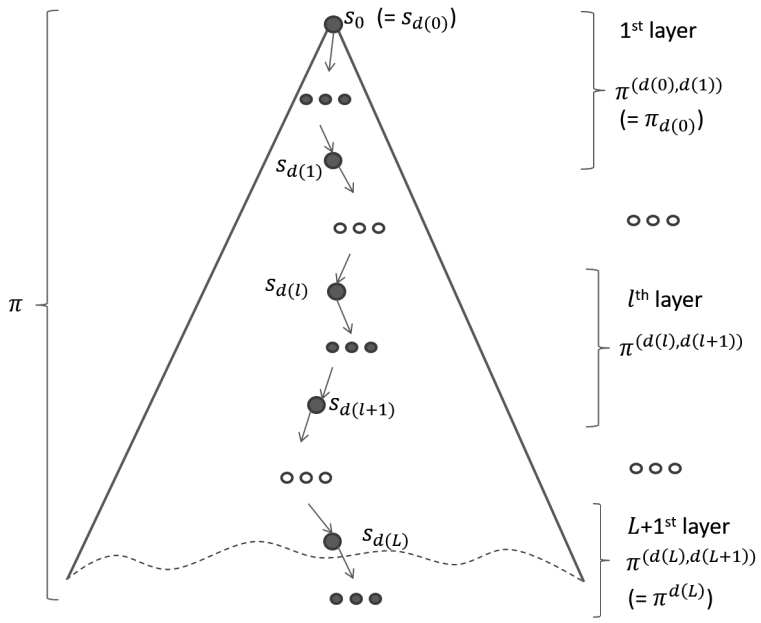

Figure 4.4: $L+1$ layer division of the reachable state space

1. $0 \leq k < L-1$

$$
\begin{aligned}
&\mathrm{Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
&\triangleq \ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow \mathrm{Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k+1)) \wedge \\
&\quad\ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow \mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k+1)) \\
&\mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
&\triangleq \ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow \mathrm{Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k+1)) \wedge \\
&\quad\ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow \mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k+1))
\end{aligned}
$$

2. $k = L-1$

$$
\begin{aligned}
&\mathrm{Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
&\triangleq \ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge \\
&\quad\ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2))) \\
&\mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
&\triangleq \ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge \\
&\quad\ ((\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)))
\end{aligned}
$$

We are ready to prove that a leads-to model checking problem $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of $2L+1$ sub-model checking problems $\boldsymbol{K}, \pi_{d(1)} \models \varphi_1 \rightsquigarrow \varphi_2, \ldots, \boldsymbol{K}, \pi^{(d(l),d(l+1))} \models \varphi_1 \rightsquigarrow \varphi_2$, $\boldsymbol{K}, \pi^{(d(l),d(l+1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2) \ldots, \boldsymbol{K}, \pi^{d(L)} \models \varphi_1 \rightsquigarrow \varphi_2, \boldsymbol{K}, \pi^{d(L)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$ for $\boldsymbol{K}$ and the $L+1$ paths $\pi^{(d(0),d(1))}$ $(= \pi_{d(1)}), \ldots, \pi^{(d(l),d(l+1))}, \ldots, \pi^{(d(L),d(L+1))}$ $(= \pi^{d(L)})$.

Theorem 4.2 ($L+1$ layer division of $\rightsquigarrow$) *Let L be any non-zero natural number. Let $d(0)$ be 0, $d(x)$ be any non-zero natural number for $x = 1, \ldots, L$ and $d(L+1)$ be ∞. Let φ_1 & φ_2 be any state propositions of $\boldsymbol{K}$. Then,*

1. $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$
2. $(\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$

Proof 4.8 *(1) and (2) are proved by simultaneous induction [43] on L.*

- *Base case ($L = 1$): It follows from Theorem 4.1 and Lemma 4.6.*
- *Induction case ($L = l + 1$): We prove the following:*
  - $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$
  - $(\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$

  Let d_{l+1} be d used in $\mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ and $\mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ such that $d_{l+1}(0) = 0$, $d_{l+1}(i)$ is an arbitrary natural number for $i = 1, \ldots, l+1$, and $d_{l+1}(l+2) = \infty$. The induction hypotheses[1] are as follows:

  - $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$
  - $(\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$

  Let d_l be d used in $\mathrm{Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ and $\mathrm{E\&Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ such that $d_l(0) = 0$, $d_l(i)$ is arbitrary natural number for $i = 1, \ldots, l$, and $d_l(l+1) = \infty$. Because $d_{l+1}(i)$ is arbitrary natural number for $i = 1, \ldots, l+1$, we suppose that $d_{l+1}(1) = d_l(1)$ and $d_{l+1}(i+1) = d_l(i)$ for $i = 1, \ldots, l$. Because π is any path of $\boldsymbol{K}$, π can be replaced by $\pi^{d_l(1)}$. If so, we have the following as instances of the induction hypothesis:

  - $(\boldsymbol{K}, \pi^{d_l(1)} \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$
  - $(\boldsymbol{K}, \pi^{d_l(1)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$

  From the definition of $\mathrm{Leads2}_L$ and $\mathrm{E\&Leads2}_L$,

  - $\mathrm{Leads2}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$ *is* $\mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$*;*
  - $\mathrm{E\&Leads2}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$ *is* $\mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$.

[1] Precisely, the induction hypotheses are $(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ and $(\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ for all Kripke structure $\boldsymbol{K}$, all paths π of $\boldsymbol{K}$ and all LTL state propositions φ_1 & φ_2 of $\boldsymbol{K}$. What are used in the proof are the instances of the hypotheses obtained by replacing π with $\pi^{d_l(1)}$.

Because $d_l(0) = d_{l+1}(0) = 0$, $d_l(1) = d_{l+1}(1)$, *and* $d_l(i) = d_{l+1}(i+1)$ *for* $i = 1, \ldots, l$, *and* $d_l(l+1) = d_{l+1}(l+2) = \infty$. *Therefore, the induction hypotheses can be rephrased as follows:*

- $(\boldsymbol{K}, \pi^{d_{l+1}(1)} \models \varphi_1 \rightsquigarrow \varphi_2) \Leftrightarrow \mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$
- $(\boldsymbol{K}, \pi^{d_{l+1}(1)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Leftrightarrow \mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$

From the definition of $\mathrm{Leads2}_L$,
$\mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ *is*

$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow \mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)) \wedge$$
$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow \mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1))$$

which is

$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^{d_{l+1}(1)} \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge$$
$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \varphi_1 \rightsquigarrow \varphi_2) \Rightarrow (\boldsymbol{K}, \pi^{d_{l+1}(1)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)))$$

because of the induction hypothesis instances. From Theorem 4.1, this is equivalent to $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$.

From the definition of $\mathrm{E\&Leads2}_L$,
$\mathrm{E\&Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ *is*

$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow \mathrm{Leads2}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)) \wedge$$
$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow \mathrm{E\&Leads2}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1))$$

which is

$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^{d_{l+1}(1)} \models \varphi_1 \rightsquigarrow \varphi_2)) \wedge$$
$$((\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)) \Rightarrow (\boldsymbol{K}, \pi^{d_{l+1}(1)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)))$$

because of the induction hypothesis instances. From Lemma 4.6, this is equivalent to $\boldsymbol{K}, \pi \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$.

Theorem 4.2 makes it possible to divide the original model checking problem $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \varphi_2$ into the $2L+1$ sub-model checking problems $\boldsymbol{K}, \pi_{d(1)} \models \varphi_1 \rightsquigarrow \varphi_2$, $\ldots$, $\boldsymbol{K}, \pi^{(d(l), d(l+1))} \models \varphi_1 \rightsquigarrow \varphi_2$, $\boldsymbol{K}, \pi^{(d(l), d(l+1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$ $\ldots$, $\boldsymbol{K}, \pi^{d(L)} \models \varphi_1 \rightsquigarrow \varphi_2$, $\boldsymbol{K}, \pi^{d(L)} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$, where $\pi_{d(1)} = \pi^{(d(0), d(1))}$ and $\pi^{d(L)} = \pi^{(d(L), d(L+1))}$.

4.3 A Divide & Conquer Approach to Leads-to Model Checking Algorithm

An algorithm can be constructed based on Theorem 4.1, which is shown as Algorithm 1. For each initial state $s_0 \in \boldsymbol{K}$, unfolding s_0 by using $\boldsymbol{T}$ without sharing any nodes that are states,

Algorithm 1: A divide & conquer approach to leads-to model checking

input : $\boldsymbol{K}$ – a Kripke structure
φ_1, φ_2 – State propositions
L – a non-zero natural number
d – a function such that $d(x)$ is a non-zero natural number for $x = 1, \ldots, L$

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

```
1  LS ← I
2  ELS ← ∅
3  forall l ∈ {1, ..., L} do
4      LS' ← {π(d(l)) | s ∈ LS ∪ ELS, π ∈ P^{d(l)}_{(K,s)}}
5      ELS' ← ∅
6      forall s ∈ LS do
7          forall π ∈ P^{d(l)}_{(K,s)} do
8              if K, π ⊭ φ1 ⇝ φ2 then
9                  LS' ← LS' − {π(d(l))}
10                 ELS' ← ELS' ∪ {π(d(l))}
11     forall s ∈ ELS do
12         forall π ∈ P^{d(l)}_{(K,s)} do
13             if K, π ⊭ (◇φ2) ∧ (φ1 ⇝ φ2) then
14                 LS' ← LS' − {π(d(l))}
15                 ELS' ← ELS' ∪ {π(d(l))}
16     LS ← LS'
17     ELS ← ELS'
18 forall s ∈ LS do
19     forall π ∈ P_{(K,s)} do
20         if K, π ⊭ φ1 ⇝ φ2 then
21             return Failure
22 forall s ∈ ELS do
23     forall π ∈ P_{(K,s)} do
24         if K, π ⊭ (◇φ2) ∧ (φ1 ⇝ φ2) then
25             return Failure
26 return Success
```

an infinite tree whose root is s_0 is made. Such an infinite tree can be divided into $L + 1$ layers as shown in Fig. 4.5, where L is a non-zero natural number. Because $\boldsymbol{R}$ is finite, the number of

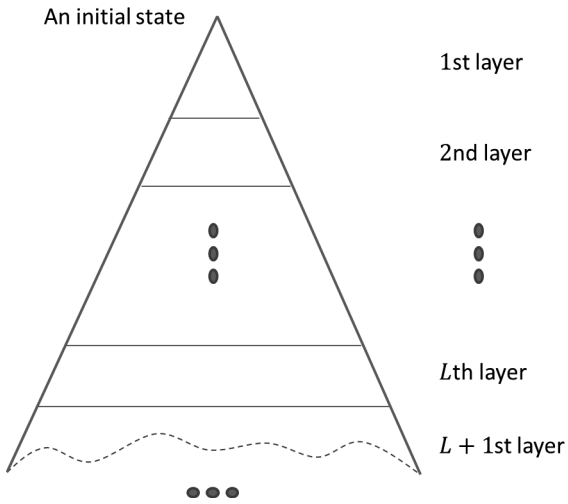

Figure 4.5: An infinite tree of states made from an initial state with $L+1$ layers

different states in each layer is finite. Theorem 4.1 makes it possible to check $\boldsymbol{K} \models \varphi_1 \rightsquigarrow \varphi_2$ in a stratified way in that for each layer $l \in \{1, \ldots, L+1\}$ we can check $\boldsymbol{K}, s, d(l) \models \varphi$ for each $s \in \{\pi(d(l-1)) \mid \pi \in \boldsymbol{P}^{d(l-1)}_{(\boldsymbol{K},s_0)}\}$, where $d(0)$ is 0, $d(x)$ is a non-zero natural number for $x = 1, \ldots, L$, $d(L+1)$ is ∞ and φ is $\varphi_1 \rightsquigarrow \varphi_2$ or $(\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$. The code fragment at lines 6 – 10 checks $\boldsymbol{K}, \pi^{(d(l-1),d(l))} \models \varphi_1 \rightsquigarrow \varphi_2$ for some paths $\pi^{(d(l-1),d(l))}$ in the lth layer of $\boldsymbol{R}$ for $l = 1, \ldots, L$. The code fragment at lines 11 – 15 checks $\boldsymbol{K}, \pi^{(d(l-1),d(l))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$ for the remaining paths $\pi^{(d(l-1),d(l))}$ in the lth layer of $\boldsymbol{R}$ for $l = 1, \ldots, L$. When l is 1, $\boldsymbol{LS}$ consists of all initial states and $\boldsymbol{ELS}$ is $\emptyset$. The code fragment at lines 18 – 21 checks $\boldsymbol{K}, \pi^{(d(L),d(L+1))} \models \varphi_1 \rightsquigarrow \varphi_2$ for some paths $\pi^{(d(L),d(L+1))}$ in the $L+1$st layer of $\boldsymbol{R}$, where $d(L+1)$ is ∞ and then $\pi^{(d(L),d(L+1))}$ is $\pi^{d(L)}$. The code fragment at lines 22 – 25 checks $\boldsymbol{K}, \pi^{(d(L),d(L+1))} \models (\Diamond\varphi_2) \wedge (\varphi_1 \rightsquigarrow \varphi_2)$ for the remaining paths $\pi^{(d(L),d(L+1))}$ in the $L+1$st layer of $\boldsymbol{R}$, where $d(L+1)$ is ∞ and then $\pi^{(d(L),d(L+1))}$ is $\pi^{d(L)}$.

The two assignments at lines 9 & 10 (and those at lines 14 & 15 as well) could be replaced with $\boldsymbol{ELS}' \leftarrow \boldsymbol{ELS}' \cup \{\pi(d(l))\}$ and the condition at line 13 (and that at line 24) could be replaced with $\boldsymbol{K}, \pi \not\models \Diamond\varphi_2$.

The proposed technique can be interpreted as follows. For each initial state, the reachable state space from the initial state is split into multiple layers ($L+1$ layers) and generates multiple smaller sub-state spaces as shown in Fig. 4.6. For each smaller sub-state space, we check if $\varphi_1 \rightsquigarrow \varphi_2$ and/or $\Diamond\varphi_2$ holds. If the number of different states in each sub-state space is much less than the one in the entire reachable state space, then model checking for each sub-state space is feasible even though model checking for the entire reachable state space is not.

Although Algorithm 1 does not construct a counterexample when Failure is returned, it could be constructed. For each $l \in \{0, 1, \ldots, L\}$, $\boldsymbol{LS}_l$ and $\boldsymbol{ELS}_l$ are prepared. As elements of $\boldsymbol{L}S_l$ and $\boldsymbol{ELS}_l$, pairs (s, s') are used, where s is a state in $\boldsymbol{S}$ or a dummy state denoted δ-stt

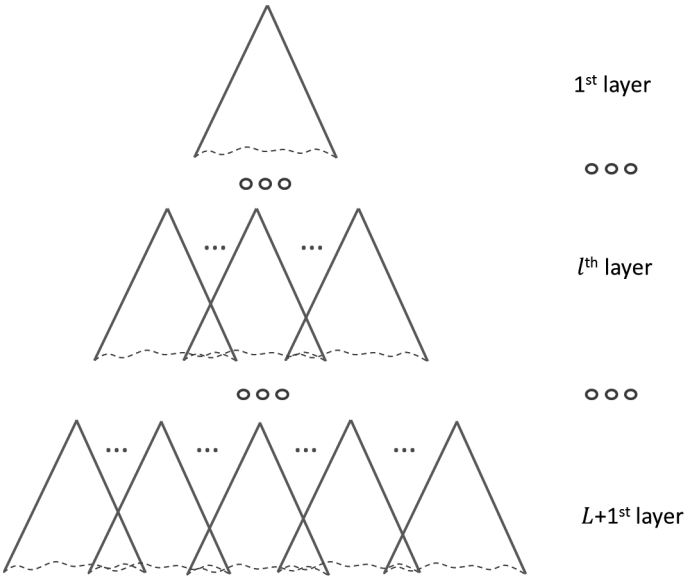

Figure 4.6: Multiple sub-state spaces obtained by splitting the reachable state space into $L+1$ layers

that is different from any state in $\boldsymbol{S}$, s' is a state in $\boldsymbol{S}$ and s' is reachable from s if $s \in \boldsymbol{S}$. The assignment at line 4 should be revised as follows:

$$\boldsymbol{LS}' \leftarrow \{(s, \pi(d(l))) \mid (s_1, s) \in \boldsymbol{LS}_{l-1} \cup \boldsymbol{ELS}_{l-1}, \pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}\}$$

the condition at line 6 should be revised as $(s_1, s) \in \boldsymbol{LS}_{l-1}$, the condition at line 11 should be revised as $(s_1, s) \in \boldsymbol{ELS}_{l-1}$, the condition at line 18 should be revised as $(s_1, s) \in \boldsymbol{LS}_L$, the condition at line 22 should be revised as $(s_1, s) \in \boldsymbol{ELS}_L$, the two assignments at lines 9 & 10 (and lines 14 & 15 as well) should be revised as follows:

$$\boldsymbol{LS}' \leftarrow \boldsymbol{LS}' - \{(s, \pi(d(l)))\}$$
$$\boldsymbol{ELS}' \leftarrow \boldsymbol{ELS}' \cup \{(s, \pi(d(l)))\}$$

and the two assignments at lines 16 & 17 should be revised as follows:

$$\boldsymbol{LS}_l \leftarrow \boldsymbol{LS}'$$
$$\boldsymbol{ELS}_l \leftarrow \boldsymbol{ELS}'$$

$\boldsymbol{LS}_0$ and $\boldsymbol{ELS}_0$ are set to $\{(\delta\text{-stt}, s) \mid s \in \boldsymbol{I}\}$ and $\emptyset$, respectively. We could then construct a counterexample, when Failure is returned, by searching through $\boldsymbol{LS}_L$, $\boldsymbol{ELS}_L$, $\ldots$, $\boldsymbol{LS}_1$, $\boldsymbol{ELS}_1$, $\boldsymbol{LS}_0$ and $\boldsymbol{ELS}_0$.

The number of different states in each layer shown in Fig. 4.5 should be much smaller than the one of different states in all layers in order to make the proposed technique effective. The number of different states in the xth one of the first L layers depends on $d(x)$ and can be much smaller than the one of different states in all layers but the number of different states in the

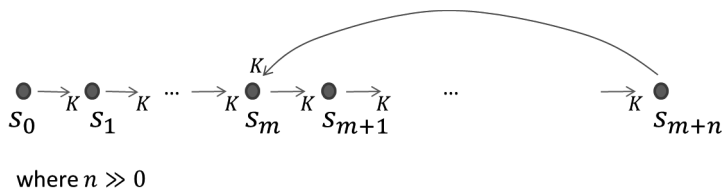

Figure 4.7: A long backward transition

$L+1$st layer may not be much smaller than that. This could happen if there is a long backward transition from a state s_{m+n} to another state s_m as shown in Fig. 4.7 such that $n \gg 0$ or s_{m+n} is far from s_m. Thus, we should write a systems specification such that there is no such a backward transition in it in order to effectively use the proposed technique.

4.4 Summary

We have proposed a new technique to mitigate the state explosion in model checking. The technique is dedicated to leads-to properties. It divides a leads-to model checking problem into multiple smaller model checking problems and tackles each smaller one. We have proved that the multiple smaller model checking problems are equivalent to the original leads-to model checking problem and designed the algorithm based on the theorem.

Chapter 5

A Divide & Conquer Approach to Conditional Stable Model Checking

In this chapter, we describe an extension work of leads-to properties to deal with conditional stable properties expressed as $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1, φ_2 are state propositions. Conditional stable properties informally say that whenever something is true, it will eventually happen that something else will be always true (or will be stable). The properties can be used to express desired properties that self-stabilizing systems [39] should satisfy. The properties can be used to formalize desired properties in self-stabilizing systems, which were first introduced by Dijkstra and became a very important concept in fault tolerance to design robust systems. Hence, it is worth focusing on conditional stable properties. The technique is called a divide & conquer approach to conditional stable model checking. It divides an original conditional stable model checking problem into multiple smaller model checking problems and tackles each smaller one. We prove a theorem that the multiple smaller model checking problems are equivalent to the original conditional stable model checking problem. An algorithm is constructed based on the theorem to support model checking conditional stable properties by our technique.

5.1 Outline of the Technique

We use the first self-stabilizing, unidirectional token ring that was proposed by Dijkstra, which is called K-state Machines (KM) [40] as an example to outline the technique. The ring system KM consists of N machines, numbered from 0 to $N-1$ and a parameter K, which is a natural number, such that $K > N$. Each machine status is represented by a natural number S, satisfying $0 \leq S < K$. The following notations are used for the ith machine:

- L refers to the status of its lefthand neighbor, machine $(i-1)$ mod N.
- S refers to the status of itself, machine i.
- R refers to the status of its righthand neighbor, machine $(i+1)$ mod N.

In the ring system KM, machine 0 is called the bottom machine. For each machine, one *privilege* (token) is defined in form of a Boolean function of its own status and the statuses of its neighbors. When the Boolean function is true, we say that the privilege is present at the machine and the machine can take its move by changing its status if the machine whose privilege is true is selected, or the privilege is selected. The privilege and its corresponding move at each machine use the format as follows:

if *privilege* **then** *corresponding move* **fi**

The privilege and its corresponding move for the bottom machine are as follows:

if $L = S$ **then** $S := (S + 1) \bmod K$ **fi**

and for the other machines as follows:

if $L \neq S$ **then** $S := L$ **fi**

The legitimate state is that it contains exactly one privilege circulating in KM. Regardless of the initial state and the privilege selected each time for the next move of a machine, the ring system is guaranteed to find itself in a legitimate state after a finite number of moves. Note that the number of available privileges in a given state is the number of possible state transitions derived from the state.

Let us specify the ring system KM in Maude. In the case that there are n machines in KM, each state in S_{KM} is formalized as:

`{(k-states:` k`) (pc[0]:` s_0`) ... (pc[`$n-1$`]:` s_{n-1}`) (#pc:` n`)}`

The `#pc` observable component stores the number of machines in KM, and the `pc[`p_i`]` observable component stores the status s_i of the machine i, which is a natural number such that $s_i < k$. The `k-states` observable component stores the natural number k. Initially, s_i is an arbitrary natural number less than k.

In this example, let us assume that there are four machines participating in KM, k is 5, and the statuses of four machines are 0, 2, 2, and 0, respectively. Note that $p[0]$ is the bottom machine. The initial state is denoted `init` as:

```
{(k-states: 5) (pc[0]: 0) (pc[1]: 2) (pc[2]: 2) (pc[3]: 0) (#pc: 4)}
```

I_{KM} has one state `init`.

T_{KM} is described in rewrite rules in what follows:

```
crl [bottom] : {(pc[J]: L) (pc[I]: S) (#pc: N) (k-states: K) OCs}
  => {(pc[J]: L) (pc[I]: ((S + 1) rem K )) (#pc: N) (k-states: K) OCs}
  if #enable({(pc[J]: L) (pc[I]: S) (#pc: N) (k-states: K) OCs}) > 1
  /\ I == 0 /\ J := sd(N,1) /\ L == S .
crl [other] : {(pc[J]: L) (pc[I]: S) (#pc: N) OCs}
  => {(pc[J]: L) (pc[I]: L)  (#pc: N) OCs}
  if #enable({(pc[J]: L) (pc[I]: S) (#pc: N) OCs}) > 1
```

```
  /\ I =/= 0 /\ J := ((sd(I,1)) rem N) /\ L =/= S .
crl [fin] : {OCs} => {OCs} if #enabled({OCs}) == 1 .
```

The three rules are named `bottom`, `other`, and `fin`. The first two rules say how to change the statuses of the bottom machine and the other machines if their privileges are true, respectively, while the last rule says that when the system reaches a legitimate state, it just stays there and does nothing. We use `fin` to avoid a long lasso loop in the specification so that the proposed technique can work effectively. The proposed technique cannot handle long lasso loops. It is one piece of our future work to come up with how to handle long lasso loops in the proposed technique. The function `#enable` returns the number of available privileges in a given state. Note that a legitimate state has exactly one privilege. `I`, `J`, `S`, `L`, `N`, and `K` are Maude variables whose types (or sorts) are natural numbers and `OCs` is a Maude variable whose sort is observable component soups. `sd`, which stands for symmetric difference, takes two natural numbers x and y and returns $|x - y|$. Given a state formalized as:

```
{(k-states: 5) (pc[0]: 0) (pc[p1]: 2) (pc[p2]: 2) (pc[p3]: 0) (#pc: 4)}
```

Each of the two rules `bottom` and `other` can be applied to the term expressing the state. Rule `bottom` can be applied to the term at one position and rule `other` can be applied to the term at two positions. Rule `bottom` can change it as:

```
{(k-states: 5) (pc[0]: 1) (pc[p1]: 2) (pc[p2]: 2) pc[p2]: 0) (#pc: 4)}
```

Fig. 5.1 shows the reachable state space made from S_{KM}, I_{KM}, and T_{KM} in which `(#pc: 4)` is not included for simplicity. The total number of states is 17.

We consider `illegal` and `legal` as atomic propositions in this example. Therefore, P_{KM} has `illegal` and `legal`. L_{KM} is defined as follows:

```
eq {OCs} |= illegal = #enabled({OCs}) > 1 .
eq {OCs} |= legal = #enabled({OCs}) == 1 .
eq {OCs} |= PROP = false [owise] .
```

`OCs` and `PROP` are Maude variables whose sorts are observable component soups and atomic propositions. The definition says that $L_{\mathrm{KM}}(s)$ consists of `illegal` if a state has more than one privilege; $L_{\mathrm{KM}}(s)$ consists of `legal` if a state s has one privilege; and L_{KM} is empty otherwise.

$K_{\mathrm{KM}} \models$ `illegal` $\rightsquigarrow \Box$ `legal` can be checked by reducing the following:

```
modelCheck(init, illegal |-> []legal)
```

`_|->_` and `[]_` are Maude operators that denote $\rightsquigarrow$ and $\Box$, respectively. It concludes that KM enjoys the conditional stable property in the case that there are four machines with our initial configuration.

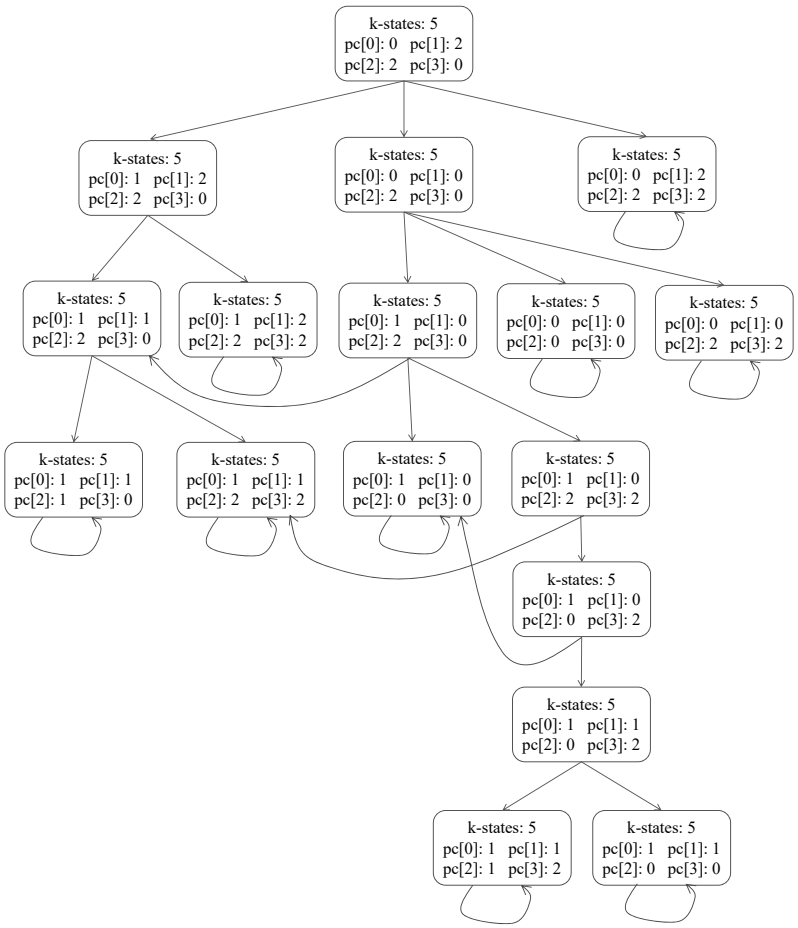

Figure 5.1: Reachable state space of KM with 4 machines

It is unnecessary to rely on the technique in order to model-check the stable conditional property, but we employ it to rough out the technique. The reachable state space depicted in Fig. 5.1 is divided into three layers as depicted in Fig. 5.2. Fig. 5.2 (a), (b), and (c) exhibit the first, second, and third layers. 15 sub-state spaces are made. There are some lasso loops, but none of them is long. The greatest number of states that belong to each sub-state space is nine, while 17 is the number of states in the entire reachable state one. In summary, the number of states in each sub-state space is less than the one in the entire reachable state space; this is the core idea the state space explosion can be eased.

For layers 1 and 2, it is necessary to change the Maude specification of KM. We change the state as:

`{(k-states:` k`) (pc[0]:` s_0`) ... (pc[`$n-1$`]:` s_{n-1}`) (#pc:` n`)(depth:` d`)}`

We have added one observable component called `depth` to manage the information of depth. The rules are changed as:

```
crl [bottom] :
  {(pc[J]: L) (pc[I]: S) (#pc: N) (k-states: K) (depth: D) OCs}
```

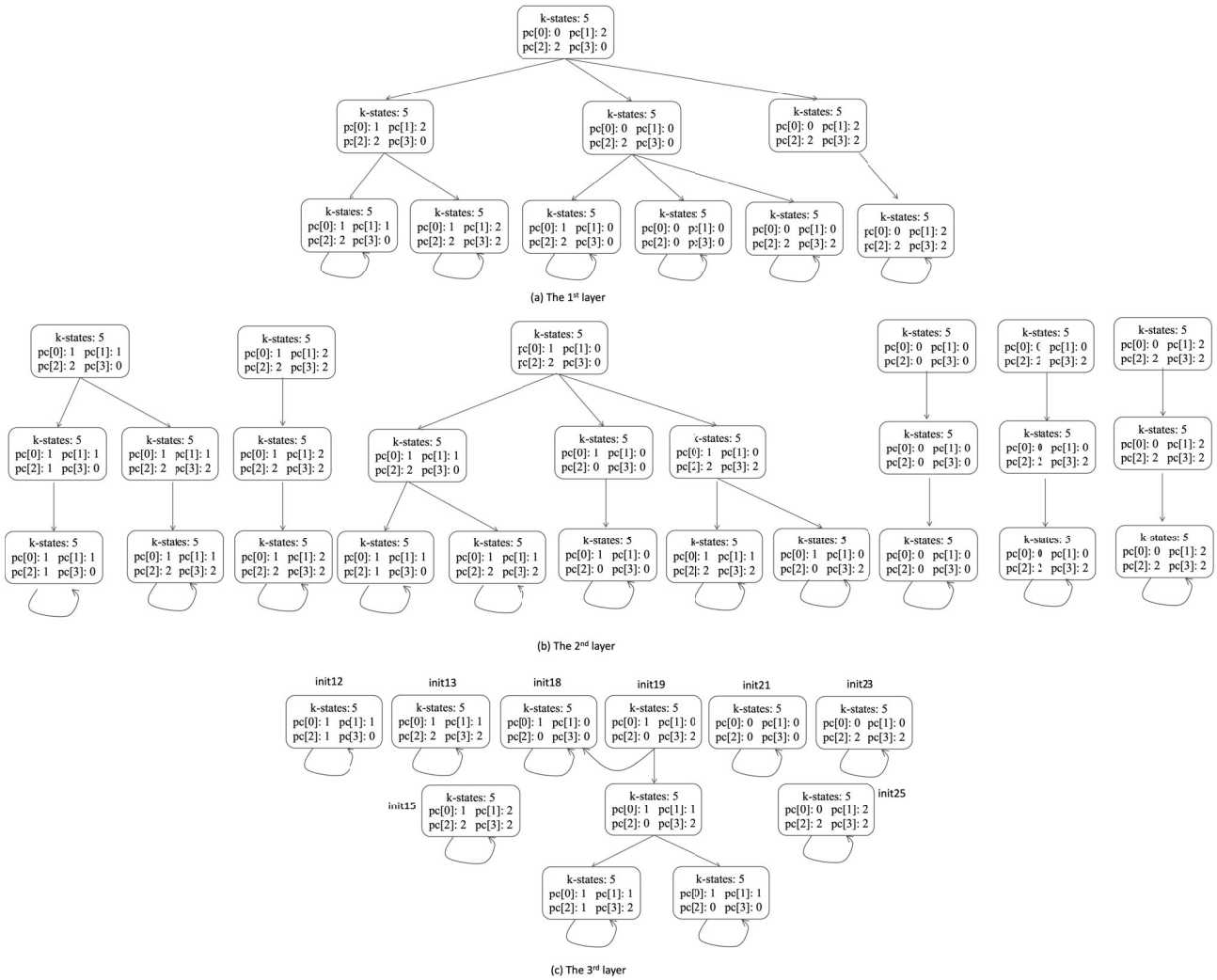

Figure 5.2: Three layers of KM reachable state space

```
  => {(pc[J]: L) (pc[I]: ((S + 1) rem K )) (#pc: N) (k-states: K)
      (depth: (D + 1)) OCs}
  if #enable({(pc[J]: L) (pc[I]: S) (#pc: N) (k-states: K) OCs}) > 1
  /\ D < Bound /\ I == 0 /\ J := sd(N,1) /\ L == S .
crl [other] :
  {(pc[J]: L) (pc[I]: S) (#pc: N) (depth: D) OCs}
  => {(pc[J]: L) (pc[I]: L)  (#pc: N) (depth: (D + 1)) OCs}
  if #enable({(pc[J]: L) (pc[I]: S) (#pc: N) OCs}) > 1
  /\ I =/= 0 /\ D < Bound /\ J := ((sd(I,1)) rem N) /\ L =/= S .
crl [fin] : {(depth: D) OCs} => {(depth: (D + 1)) OCs}
  if #enabled({OCs}) == 1 /\ D < Bound.
crl [stutter] : {(depth: D) OCs}  => {(depth: D) OCs} if D >= Bound .
```

`D` is a Maude variable whose sort is natural numbers. `Bound` is a Maude constant whose sort is natural numbers. We use 2 as `Bound` for the example used.

Let `init` denote the following:

```
{(k-states: 5) (pc[0]: 0) (pc[1]: 2) (pc[2]: 2) (pc[3]: 0) (#pc: 4) (depth: 0)}
```

We are supposed to first gather all non-cx states and all cx ones placed at depth 2 from `init`. The union of non-cx and cx states is all states placed at depth 2 reachable from `init`. How to gather such states is described in the next section. We follow Algorithm 2; if a path satisfies $\Box\neg illegal$, the last state (that has the self-transition) of the path is regarded as a non-cx state; otherwise, the last state is a cx state. A non-cx state can become a cx state if it is found later as a cx state. There are only six cx states in the first layer (see Fig. 5.2 (a)):

```
{(pc[0]: 1) (pc[1]: 1) (pc[2]: 2) (pc[3]: 0) (#pc: 4) (depth: 2) (k-states: 5)}
{(pc[0]: 1) (pc[1]: 2) (pc[2]: 2) (pc[3]: 2) (#pc: 4) (depth: 2) (k-states: 5)}
{(pc[0]: 1) (pc[1]: 0) (pc[2]: 2) (pc[3]: 0) (#pc: 4) (depth: 2) (k-states: 5)}
{(pc[0]: 0) (pc[1]: 0) (pc[2]: 0) (pc[3]: 0) (#pc: 4) (depth: 2) (k-states: 5)}
{(pc[0]: 0) (pc[1]: 0) (pc[2]: 2) (pc[3]: 2) (#pc: 4) (depth: 2) (k-states: 5)}
{(pc[0]: 0) (pc[1]: 2) (pc[2]: 2) (pc[3]: 2) (#pc: 4) (depth: 2) (k-states: 5)}
```

These states are denoted `init4`, `init5`, `init6`, `init7`, `init8`, and `init9` that are utilized as the initial states for the second layer. `Bound` should be `4` for layer 2. For the six initial states, we generate all states positioned at depth 4 reachable from those states for the second layer because they are cx states. How to generate such states is described in the next section.

Fig. 5.2 (b) shows eight states positioned at depth 4 (from `init`) in the second layer. Two states are positioned at depth 4 reachable from `init4`. Four states are positioned at depth 4 reachable from `init6`, where two states of them are also reachable from `init4`. One state is positioned at depth 4 reachable from each `init5`, `init7`, `init8`, and `init9`. The eight states are denoted `init12`, `init13`, `init18`, `init19`, `init5'`, `init7'`, `init8'`, and `init9'`. Note that `init5'`, `init7'`, `init8'`, and `init9'` are the same as `init5`, `init7`, `init8`, and `init9` except the `depth` observable component. Hence, there are actually four new states `init12`, `init13`, `init18`, and `init19` as follows:

```
{(pc[0]: 1) (pc[1]: 1) (pc[2]: 1) (pc[3]: 0) (#pc: 4) (depth: 4) (k-states: 5)}
{(pc[0]: 1) (pc[1]: 1) (pc[2]: 2) (pc[3]: 2) (#pc: 4) (depth: 4) (k-states: 5)}
{(pc[0]: 1) (pc[1]: 0) (pc[2]: 0) (pc[3]: 0) (#pc: 4) (depth: 4) (k-states: 5)}
{(pc[0]: 1) (pc[1]: 0) (pc[2]: 0) (pc[3]: 2) (#pc: 4) (depth: 4) (k-states: 5)}
```

We use the eight states `init12`, `init13`, `init18`, `init19`, `init5'`, `init7'`, `init8'`, and `init9'` from which `(depth: 4)` is removed as the initial states in the final layer. The initial Maude specification is used for the final layer. The eight model-checking experiments are carried out as:

```
modelCheck(init12, <>[] legal)
modelCheck(init13, <>[] legal)
modelCheck(init18, <>[] legal)
modelCheck(init19, <>[] legal)
modelCheck(init5', <>[] legal)
modelCheck(init7', <>[] legal)
modelCheck(init8', <>[] legal)
modelCheck(init9', <>[] legal)
```

Because no cx is found, KM satisfies the conditional stable property in the case there are four machines with our initial configuration by using our technique.

5.2 Multiple Layer Division of Conditional Stable Model Checking

Firstly, we prove the Theorem 5.1 that a conditional stable model checking problem for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of three sub-model checking problems for $\boldsymbol{K}$ and two paths of $\boldsymbol{K}$, where the two paths are obtained by splitting π into two parts. We then prove in Theorem 5.1 that a leads-to model checking problem is equivalent to a model checking problem that is composed of $2L+1$ sub-model checking problems for $\boldsymbol{K}$ and $L+1$ paths of $\boldsymbol{K}$, where $L \geq 1$ and the $L+1$ paths are obtained by splitting π into $L+1$ parts.

Let $\boldsymbol{K}$ be any Kripke structure, π be any path of $\boldsymbol{K}$ and φ_1 & φ_2 be any LTL formulas of $\boldsymbol{K}$ in this section unless otherwise stated. π is an arbitrarily chosen path (or computation) of $\boldsymbol{K}$ and then can be regarded as the whole reachable state space (see Fig. 4.3). Let π be split into two paths π_k and π^k, where π_k can be regarded as the first layer of the reachable state space and π^k can be regarded as the second layer of the reachable state space (see Fig. 4.3). We prove four lemmas from which Theorem 5.1 is derived.

The first lemma says that if $\Box\varphi$ holds for the whole reachable state space, then the first and second layer also hold $\Box\varphi$, where φ is a state proposition and it is vice versa.

Lemma 5.1 *Let φ be any state proposition of $\boldsymbol{K}$. Let k be any natural number. Then, $(\boldsymbol{K}, \pi \models \Box\varphi) \Leftrightarrow (\boldsymbol{K}, \pi_k \models \Box\varphi) \wedge (\boldsymbol{K}, \pi^k \models \Box\varphi)$.*

Proof 5.1 *Because φ is a state proposition, whether it holds only depends on the first state of a given path. If $(\boldsymbol{K}, \pi \models \Box\varphi)$, then φ holds for $\pi(i)$ for all i, and vice versa. If $\boldsymbol{K}, \pi_k \models \Box\varphi$ and $\boldsymbol{K}, \pi^k \models \Box\varphi$, then φ holds for $\pi(i)$ for $i = 0, \ldots, k$ and φ holds for $\pi(i)$ for $i = k, \ldots$, respectively, and therefore φ holds for $\pi(i)$ for all i, and vice versa.*

The second lemma says that if $\Diamond\Box\varphi_2$ holds for the second layer, then the whole reachable state space holds $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions.

Lemma 5.2 *Let φ_1, φ_2 be any state propositions of $\boldsymbol{K}$. Let k be any natural number. Then,* $(\boldsymbol{K}, \pi^k \models \Diamond\Box\varphi_2) \Rightarrow (\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2)$.

Proof 5.2 *From the assumption, there exists i ($\geq k$) such that $\boldsymbol{K}, \pi^i \models \Box\varphi_2$. Thus, $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$.* $\Box$

The third lemma says that if $\Box\neg\varphi_1$ holds for the first layer and $\varphi_1 \rightsquigarrow \Box\varphi_2$ holds for the second layer, then the whole reachable state space holds $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions.

Lemma 5.3 *Let φ_1, φ_2 be any state propositions of $\boldsymbol{K}$. Let k be any natural number. Then,* $(\boldsymbol{K}, \pi_k \models \Box\neg\varphi_1) \wedge (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Rightarrow (\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2)$.

Proof 5.3 *The case is split into two cases: (1) $\boldsymbol{K}, \pi^k \models \Box\neg\varphi_1$ and (2) $\boldsymbol{K}, \pi^k \not\models \Box\neg\varphi_1$. In (1), $\boldsymbol{K}, \pi \models \Box\neg\varphi_1$ from the first conjunct of the assumption and Lemma 5.1. Hence, $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$. In (2), from the second conjunct of the assumption, there exists i ($\geq k$) such that $\boldsymbol{K}, \pi^i \models \Box\varphi_2$. Thus, $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$.* $\Box$

The fourth lemma says that if $\varphi_1 \rightsquigarrow \Box\varphi_2$ holds for the whole reachable state space, then $\varphi_1 \rightsquigarrow \Box\varphi_2$ holds for the first layer and $\varphi_1 \rightsquigarrow \Box\varphi_2$ holds for the second layer, where φ_1 and φ_2 are state propositions and it is vice versa.

Lemma 5.4 (Two layer division of $\varphi_1 \rightsquigarrow \Box\varphi_2$) *Let φ_1, φ_2 be any state propositions of $\boldsymbol{K}$. Let k be any natural number. Then,*

$$\begin{aligned}
&(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2) \\
\Leftrightarrow\ & [(\boldsymbol{K}, \pi_k \models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \Box\varphi_2)] \wedge \\
& [(\boldsymbol{K}, \pi_k \not\models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^k \models \Diamond\Box\varphi_2)]
\end{aligned}$$

Proof 5.4 *(1) Case "only if" ($\Rightarrow$): The case is split into two cases: (1.1) $\boldsymbol{K}, \pi \models \Box\neg\varphi_1$ and (1.2) $\boldsymbol{K}, \pi \not\models \Box\neg\varphi_1$. In (1.1), $\boldsymbol{K}, \pi^k \models \Box\neg\varphi_1$ from Lemma 5.1. Therefore, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \Box\varphi_2$. In (1.2), there exists i such that $\boldsymbol{K}, \pi^i \models \varphi_1$. Thus, from the assumption, there exists j ($\geq i$) such that $\boldsymbol{K}, \pi^j \models \Box\varphi_2$. Hence, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \Box\varphi_2$ and $\boldsymbol{K}, \pi^k \models \Diamond\Box\varphi_2$.*

(2) Case "if" ($\Leftarrow$): The case is split into two cases: (2.1) $\boldsymbol{K}, \pi_k \models \Box\neg\varphi_1$ and (2.2)$\boldsymbol{K}, \pi_k \not\models \Box\neg\varphi_1$. In (2.1), $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$ from Lemma 5.3. In (2.2), $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$ from Lemma 5.2. $\Box$

We define $\mathrm{CStable}_L$ that is abbreviations of statements on ternary satisfaction relations among Kripke structures, their paths and LTL formulas. To this end, let π be split into $L+1$ paths $\pi^{(d(0),d(1))}$ ($= \pi_{d(1)}$), ..., $\pi^{(d(l),d(l+1))}$, ..., $\pi^{(d(L),d(L+1))}$ ($= \pi^{d(L)}$), where $L \geq 1$, $d(0) = 0$, $d(L+1) = \infty$ and $d(l)$ is a non-zero natural number for $l = 1, \ldots, L$ (see Fig. 4.4). $\pi^{(d(l),d(l+1))}$ can be regarded as the $l+1$st layer of the reachable state space for $l = 0, 1, \ldots, L$.

Definition 5.1 (CStable$_L$) *Let L be any non-zero natural number, k be any natural number and d be any function such that $d(0)$ is 0, $d(x)$ is a natural number for $x = 1, \ldots, L$ and $d(L+1)$ is ∞.*

1. $0 \leq k < L - 1$

$$\begin{aligned}
&\mathrm{CStable}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
\triangleq\ & [(\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models \Box\neg\varphi_1) \Rightarrow \mathrm{CStable}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k+1)] \wedge \\
& [(\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{(d(L),d(L+1))} \models \Diamond\Box\varphi_2)]
\end{aligned}$$

2. $k = L - 1$

$$\begin{aligned}
&\mathrm{CStable}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, k) \\
\triangleq\ & [(\boldsymbol{K}, \pi^{(d(k),d(k+1))} \models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models \varphi_1 \rightsquigarrow \Box\varphi_2)] \wedge \\
& [(\boldsymbol{K}, \pi^{(d(k),d(k+1))} \not\models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{(d(k+1),d(k+2))} \models \Diamond\Box\varphi_2)]
\end{aligned}$$

We are ready to prove that a conditional stable model checking problem $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$ for a Kripke structure $\boldsymbol{K}$ and a path π of $\boldsymbol{K}$ is equivalent to a model checking problem that is composed of three sub-model checking problems $\boldsymbol{K}, \pi_k \models \Box\neg\varphi_1$, $\boldsymbol{K}, \pi^k \models \varphi_1 \rightsquigarrow \Box\varphi_2$ and $\boldsymbol{K}, \pi^k \models \Diamond\Box\varphi_2$ for $\boldsymbol{K}$ and the two paths π_k and π^k of $\boldsymbol{K}$, where k is a natural number.

Theorem 5.1 ($L+1$ **layer division of** $\varphi_1 \rightsquigarrow \Box\varphi_2$) *Let L be any non-zero natural number. Let $d(0)$ be 0, $d(x)$ be any natural number for $x = 1, \ldots, L$ and $d(L+1)$ be ∞. Let φ_1, φ_2 be any state propositions of $\boldsymbol{K}$. Then,*

$$(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Leftrightarrow \mathrm{CStable}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$$

Proof 5.5 *By induction on L.*

- *Base case ($L = 1$): It follows from Lemma 5.4.*
- *Induction case ($L = l + 1$): We prove the following:*

$$(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Leftrightarrow \mathrm{CStable}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$$

Let d_{l+1} be d used in $\mathrm{CStable}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ such that $d_{l+1}(0) = 0$, $d_{l+1}(i)$ is an arbitrary natural number for $i = 1, \ldots, l+1$ and $d_{l+1}(l+2) = \infty$. The induction hypothesis is as follows:

$$(\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Leftrightarrow \mathrm{CStable}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$$

Let d_l be d used in $\mathrm{CStable}_l(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ such that $d_l(0) = 0$, $d_l(i)$ is an arbitrary natural number for $i = 1, \ldots, l$ and $d_l(l+1) = \infty$. Because $d_{l+1}(i)$ is an arbitrary natural number for $i = 1, \ldots, l+1$, we suppose that $d_{l+1}(1) = d_l(1)$ and $d_{l+1}(i+1) = d_l(i)$ for

Algorithm 2: A divide & conquer approach to conditional stable model checking

input : $\boldsymbol{K}$ – a Kripke structure
φ_1, φ_2 – State propositions
L – a non-zero natural number
d – a function such that $d(x)$ is a non-zero natural number for $x = 1, \ldots, L$

output: Success ($\boldsymbol{K} \models \varphi_1 \rightsquigarrow \Box\varphi_2$) or Failure ($\boldsymbol{K} \not\models \varphi_1 \rightsquigarrow \Box\varphi_2$)

```
NCxS ← I
CxS ← ∅
forall l ∈ {1, ..., L} do
    NCxS' ← {π(d(l)) | s ∈ NCxS, π ∈ P^{d(l)}_{(K,s)}}
    CxS' ← {π(d(l)) | s ∈ CxS, π ∈ P^{d(l)}_{(K,s)}}
    forall s ∈ NCxS do
        forall π ∈ P^{d(l)}_{(K,s)} do
            if K, π ⊭ □¬φ1 then
                NCxS' ← NCxS' − {π(d(l))}
                CxS' ← CxS' ∪ {π(d(l))}
    NCxS ← NCxS'
    CxS ← CxS'
forall s ∈ NCxS do
    forall π ∈ P_{(K,s)} do
        if K, π ⊭ φ1 ⇝ □φ2 then
            return Failure
forall s ∈ CxS do
    forall π ∈ P_{(K,s)} do
        if K, π ⊭ ◇□φ2 then
            return Failure
return Success
```

$i = 1, \ldots, l$. Because π is any path of $\boldsymbol{K}$, π can be replaced with $\pi^{d_l(1)}$. If so, we have the following as an instance of the induction hypothesis:

$$(\boldsymbol{K}, \pi^{d_l(1)} \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Leftrightarrow \mathrm{CStable}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$$

From Definition 5.1, $\mathrm{CStable}_l(\boldsymbol{K}, \pi^{d_l(1)}, \varphi_1, \varphi_2, 0)$ is $\mathrm{CStable}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$ because $d_l(0) = d_{l+1}(0) = 0$, $d_l(1) = d_{l+1}(1)$ and $d_l(i) = d_{l+1}(i+1)$ for $i = 1, \ldots, l$ and $d_l(l+1) = d_{l+1}(l+2) = \infty$. Therefore, the induction hypothesis instance can be rephrased

as follows:

$$(\boldsymbol{K}, \pi^{d_{l+1}(1)} \models \varphi_1 \rightsquigarrow \Box\varphi_2) \Leftrightarrow \mathrm{CStable}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)$$

From Definition 5.1, $\mathrm{CStable}_{l+1}(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 0)$ *is*

$$\begin{aligned}&[(\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \Box\neg\varphi_1) \Rightarrow \mathrm{CStable}_L(\boldsymbol{K}, \pi, \varphi_1, \varphi_2, 1)] \wedge \\ &[(\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{(d_{l+1}(L), d_{l+1}(L+1))} \models \Diamond\Box\varphi_2)]\end{aligned}$$

which is

$$\begin{aligned}&[(\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{d_{l+1}(1)} \models \varphi_1 \rightsquigarrow \Box\varphi_2)] \wedge \\ &[(\boldsymbol{K}, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \Box\neg\varphi_1) \Rightarrow (\boldsymbol{K}, \pi^{(d_{l+1}(L), d_{l+1}(L+1))} \models \Diamond\Box\varphi_2)]\end{aligned}$$

because of the induction hypothesis instance. From Lemma 4.5, this is equivalent to $\boldsymbol{K}, \pi \models \varphi_1 \rightsquigarrow \Box\varphi_2$. □

5.3 A Divide & Conquer Approach to Conditional Stable Model Checking Algorithm

An algorithm can be constructed based on Theorem 5.1, which is shown as Algorithm 2. $\boldsymbol{CxS}$ consists of all cx states located at the bottom of the Lth layer while $\boldsymbol{NCxS}$ consists of all the non-cx states located at the bottom of the Lth layer. $\boldsymbol{NCxS} \cup \boldsymbol{CxS}$ is the set of all states located at the bottom of the Lth layer (or the top of the $L+1$st layer). Initially, $\boldsymbol{NCxS}$ consists of only the initial states in $\boldsymbol{I}$ at line 1 and $\boldsymbol{CxS}$ is an empty set. For the first **forall** loop in the code fragment at lines 3–12, we need to collect non-cx and cx states located at each layer of intermediate layers stored in $\boldsymbol{NCxS}$ and $\boldsymbol{CxS}$ for the next layer, respectively.

We use $\boldsymbol{NCxS'}$ and $\boldsymbol{CxS'}$ to temporarily store non-cx and cx states while collecting states. For each layer l of intermediate layers, $\boldsymbol{NCxS'}$ and $\boldsymbol{CxS'}$ first store all states located at depth $d(l)$ reachable from each state in $\boldsymbol{NCxS}$ and $\boldsymbol{CxS}$ at lines 4 and 5, respectively. For each non-cx state in $\boldsymbol{NCxS}$, we check whether each path π that starts with the state up to depth $d(l)$ satisfies $\Box\neg\varphi_1$ at lines 6–9. If not, the last state of π is removed from $\boldsymbol{NCxS'}$ and added to $\boldsymbol{CxS'}$ at lines 9 and 10, respectively. Lastly, $\boldsymbol{NCxS'}$ and $\boldsymbol{CxS'}$ are assigned to $\boldsymbol{NCxS}$ and $\boldsymbol{CxS}$ at lines 11 and 12 for the next layer.

The code fragment at lines 13 – 16 checks $\varphi_1 \rightsquigarrow \Box\varphi_2$ for each path that starts with each state in $\boldsymbol{NCxS}$. The code fragment at lines 17 – 20 checks $\Diamond\Box\varphi_2$ for each path that starts with each state in $\boldsymbol{CxS}$. If there are any counterexamples found at the final layer, Failure is returned.

Although Algorithm 2 does not construct a counterexample when Failure is returned, it could be constructed. For each $l \in \{0, 1, \ldots, L\}$, $\boldsymbol{NCxS_l}$ and $\boldsymbol{CxS_l}$ are arranged. As elements of $\boldsymbol{NCxS_l}$ and $\boldsymbol{CxS_l}$, pairs (s, s') are used, where s is a state in $\boldsymbol{S}$ or a dummy state denoted δ-stt that is different from any state in $\boldsymbol{S}$, s' is a state in $\boldsymbol{S}$, and s' is reachable from s if $s \in \boldsymbol{S}$. The two assignments at lines 4 and 5 are to be revised as follows:

$\boldsymbol{NCxS'} \leftarrow \{(s, \pi(d(l)))|\ (s_1, s) \in \boldsymbol{NCxS_{l-1}}, \pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}\}$
$\boldsymbol{CxS'} \leftarrow \{(s, \pi(d(l)))|\ (s_1, s) \in \boldsymbol{CxS_{l-1}}, \pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}\}$

The condition at line 6 is to be revised as $(s_1, s) \in \boldsymbol{NCxS_{l-1}}$, the condition at line 13 is to be revised as $(s_1, s) \in \boldsymbol{NCxS_L}$, and the condition at line 17 is to be revised as $(s_1, s) \in \boldsymbol{CxS_L}$. The two assignments at lines 9 and 10 are to be revised as follows:

$\boldsymbol{NCxS'} \leftarrow \boldsymbol{NCxS'} - \{(s, \pi(d(l)))\}$
$\boldsymbol{CxS'} \leftarrow \boldsymbol{CxS'} \cup \{(s, \pi(d(l)))\}$

and the two assignments at lines 11 and 12 are to be revised as follows:

$\boldsymbol{NCxS_l} \leftarrow \boldsymbol{NCxS'}$
$\boldsymbol{CxS_l} \leftarrow \boldsymbol{CxS'}$

$\boldsymbol{NCxS_0}$ and $\boldsymbol{CxS_0}$ are initially $\{(\delta\text{-stt}, s) \mid s \in \boldsymbol{I}\}$ and $\emptyset$. A cx could be made, when Failure is returned, by searching through $\boldsymbol{NCxS}_L$, $\boldsymbol{CxS}_L$, ..., $\boldsymbol{NCxS}_1$, $\boldsymbol{CxS}_1$, $\boldsymbol{NCxS}_0$, and $\boldsymbol{CxS}_0$.

5.4 Summary

We have described how to model check conditional stable properties expressed as $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1, φ_2 are state propositions, in a stratified way. The technique aims at mitigating the state space explosion in model checking. We have proved a theorem that the proposed technique is correct and designed an algorithm based on the theorem. The technique is an extension of the technique used for leads-to and eventual model checking [4, 2] in order to deal with desired properties of self-stabilizing systems that can be expressed as conditional stable properties. The support tool with some experiments showing the power of the technique is described in detail in Chapter 7.

Chapter 6

A Support Tool for the Divide & Conquer Approach to Leads-to Model Checking

The chapter describes a support tool for the $L+1$-layer divide & conquer approach to leads-to model checking and conducts some experiments with the tool showing that the tool/technique can alleviate the state space explosion problem to some extent.

6.1 How the Tool Works in a Nutshell

Given the original specification of TAS including the leads-to property concerned, the number of layers and each layer depth, the model checking experiment used to outline the $L+1$-layer divide & conquer approach to leads-to model checking in Sect. 4.1 is conducted with the support tool as follows:

```
Maude> in specs/tas
...
Maude> in full-maude
...

    Full Maude 2.7.1 June 30th 2016

Maude> in solver
...

    L+1 Layers Divide & Conquer Approach to Leads-To Model Checking Available Now

Done reading in file: "solver-loop.maude"

Maude> (initialize[TAS-CHECK, init, l2-prop, e-prop, OComp, Soup{OComp}])
Initialization: success
```

```
    originModule: TAS-CHECK
    reviseModule: TAS-CHECK-REVISE
    initialState: init
    leadsToFormula: l2-prop
    eventuallyFormula: e-prop
    eleSort: OComp
    soupSort: Soup`{OComp`}

Maude> (check 2 2)
Analyzer:
    currentDepth: 4
    depthList: 2 2
    numberOfNodeSet: 4
    numberOfStates: 2
    numberOfCxStates: 1

Check: success

Maude>
```

where some outputs emitted by Maude are omitted, where ... is written.

First the module in which TAS is specified, Full Maude and the support tool are loaded into Maude in this order. Then, the tool is initialized with the command of the tool:

```
(initialize[TAS-CHECK, init, l2-prop, e-prop, OComp, Soup{OComp}])
```

where `TAS-CHECK` is the module in which TAS and its properties to be checked are specified, `init` is an initial state of TAS, `l2-prop` is the leads-to property (or formula) `inWs1 |-> inCs1`, `e-prop` is the eventually property (or formula) `<> inCs1`, `OComp` is the sort of observable components and `Soup{OComp}` is the sort of observable component soups. In the initialization of the tool, a revised specification of TAS is made by adding the `depth` observable component to the original specification as described in Sect. 4.1. The revised specification is referred as `TAS-CHECK-REVISE`.

Next we conduct the 3-layer divide & conquer approach to leads-to model checking for TAS by using the command `check 2 2`, where each of the first two layer depths is 2 and the final layer is ∞. The tool returns `success`, meaning TAS satisfies the leads-to property under verification.

6.2 Tool Implementation in Maude

The support tool is developed in Maude by using meta-programming on top of Full Maude [46]. Full Maude maintains a database of modules that we call DB. Our tool maintains an extended

database that contains our configurations (our way to express states) and data gathered at each layer. The extended database is called DB-EXT. DB-EXT stores the original specification under model checking, such as `TAS-CHECK`, the revised version of the specification, such as `TAS-CHECK-REVISE`, the initial state, the leads-to property, the eventually property, the sort of observable components, and the sort of observable component soups. We also save in DB-EXT the current depth that is initially 0, the list of depth information fed by users that is initially `nil`, the result that is initially `nil` and the node set (denoted NodeSet, one of the most crucial data structure used in the tool) that initially contains the initial state only.

The result stored in DB-EXT is `nil`, `success` or in the following form:

`formula:` $formula$`, term:` $state$`, trace:` log`, cx:` cx

where $formula$ is either the leads-to property or the eventually property, $state$ is a state that is the initial state of a sub-state space under model checking, log is the log to $state$ from the actual initial state and cx is a (local) counterexample found while doing the model checking experiment for the sub-state space. While tackling a sub-state space in intermediate layers, the result stored in DB-EXT is `nil`. The result stored in DB-EXT is `success` if and only if all sub-state spaces have been tackled and no counterexample has been found. When the tool tackles a sub-state space whose start state is s_0 in the final layer for a formula f and finds a (local) counterexample cx, the result stored in DB-EXT becomes as follows:

`formula:` f`, term:` s_0`, trace:` log`, cx:` cx

where log is a sequence of states from the actual initial state to s_0. We can construct a (global) counterexample from log and cx.

NodeSet is constructed from `empty` and `_,_`, where `empty` denotes the empty NodeSet and is an identity of `_,_` that is associative and commutative. If NodeSet is not `empty`, it is in the following form:

$node_1$ `,` $\ldots$, $node_i$ `,` $node_{i+1}$ `,` $\ldots$ `,` $node_n$

Each $node_i$ is in the following form:

`<` `<` $states$ `:` $cxstates$ `>` `:` $list$ `>`

where $states$ is a set of states located at some depth, $cxstates$ is a set of counterexample states located at the same depth such that $cxstates \subseteq states$ and $list$ is a list of state & natural number-pairs. Initially, NodeSet only consists of one element, where $states$ only consists of the initial state to which the `depth` observable component is added, where `0` is stored, $cxstates$ is the empty set and $list$ is the empty list. In general, if we consider a sub-state space (as shown in Fig. 6.1) that starts with s_1^{l+1} located at depth $d_1+\ldots+d_l$ and is in the $l+1$st layer whose depth is d_{l+1}, where $s_{n_1}^{l+1}, \ldots, s_{n_k}^{l+1}, s_{n_{k+1}}^{l+1}, \ldots, s_{n_m}^{l+1}$ are all states located at depth $d_1 + \ldots + d_l + d_{l+1}$ reachable from s_1^{l+1} and among them $s_{n_{k+1}}^{l+1}, \ldots, s_{n_m}^{l+1}$ are counterexample states, then the $node$ is as follows:

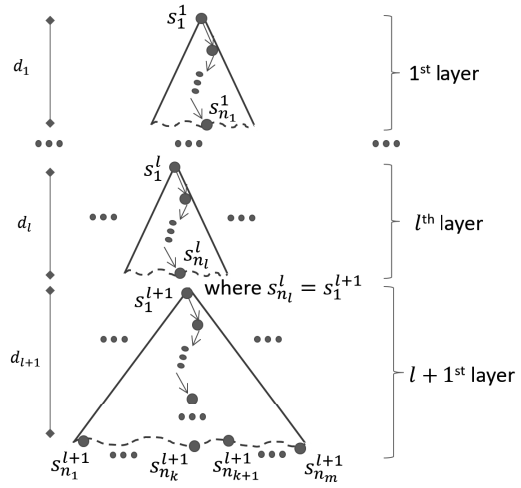

Figure 6.1: Some information maintained by the tool

$$< < s_{n_1}^{l+1} | \ldots | s_{n_k}^{l+1} | s_{n_{k+1}}^{l+1} | \ldots | s_{n_m}^{l+1} : s_{n_{k+1}}^{l+1} | \ldots | s_{n_m}^{l+1} >$$
$$: < s_1^{l+1} : d_{l+1} > < s_1^l : d_l > \ \ldots < s_1^1 : d_1 > >$$

where the operator `_|_` is used as the constructor of non-empty sets and the juxtaposition operator `__` is used as the constructor of non-empty lists. In the *node*, *list* is as follows:

$$< s_1^{l+1} : d_{l+1} > < s_1^l : d_l > \ \ldots < s_1^1 : d_1 >$$

Each $< s_1^i : d_i >$ is the pair of s_1^i and d_i, where s_1^i is a state located at the top of the ith layer and d_i is the depth of the ith layer for $i = 1, \ldots, l, l+1$. Each s_1^{i+1} is reachable from s_1^i for $i = 1, \ldots, l$.

The command `(check 2 2)` conducts the 3-layer divide & conquer approach to leads-to model checking, where each of the first two layer depths is 2. One primary function used by the command is `layerCheck` that is defined as follows:

```
eq layerCheck(M, NS, N, Cs)
= $layerCheck(M, NS, N, empty, < emptyTermSet : emptyTermSet >, Cs) .
```

where `M` is the revised specification, `NS` is the node set that are treated as the initial states of a layer (say the lth layer), `N` is the depth of the layer from the original initial state (say $d_0 + \ldots + d_l$ if it is the lth layer), and `Cs` is the sort of our configuration (`ConfigExtSet`). `layerCheck` handles each layer (say the lth layer) except for the last one and returns the node set of the next layer (say the $l+1$st layer).

The function `$layerCheck` is defined as follows:

```
eq $layerCheck(M,empty,N,NS,CACHE,Cs) = NS .
ceq $layerCheck(M,(ND , NS1),N,NS2,CACHE,Cs)
= $layerCheck(M,NS1,N,union(NS2,ns(NS&CACHE)),cache(NS&CACHE),Cs)
if NS&CACHE := $$layerCheck(M,ND,N,empty,CACHE,Cs) .
```

The first three parameters and the sixth one are the same as the four parameters of `layerCheck`, respectively. The fourth parameter maintains the result (the node set) and the fifth parameter maintains a cache that is used to avoid duplicate nodes in the result. `$layerCheck` returns a node set.

The function `$$layerCheck` is defined as follows:

```
eq $$layerCheck(M,< < emptyTermSet : emptyTermSet > : LL >,N,NS,CACHE,Cs)
= < NS : CACHE > .
ceq $$layerCheck(M,< < T | TS1 : TS2 > : LL >,N,NS,CACHE,Cs)
= $$layerCheck(M,< < TS1 : TS2 > : LL >,N,
   union(NS,filterValidNode(
     < < (STATES except states(CACHE)) : (CxSTATES except cxStates(CACHE)) >
       : (< T : N >  LL) >)),
   < (STATES | states(CACHE)) : (CxSTATES | cxStates(CACHE)) >,Cs)
if STATES := genTermSet(M, T, N, Cs) /\
   CxSTATES := allCounterExample(M,T,leadsToFormula(Cs),eventuallyProp(Cs)) .
ceq $$layerCheck(M,< < emptyTermSet : T | TS2 > : LL >,N,NS,CACHE,Cs)
= $$layerCheck(M,< < emptyTermSet : TS2 > : LL >,N,
   union(NS,filterValidNode(
     < < emptyTermSet : (CxSTATES except cxStates(CACHE)) > : (< T : N > LL) >)),
   < states(CACHE) : (CxSTATES | cxStates(CACHE)) >,Cs)
if CxSTATES := allCounterExample(M,T,eventuallyFormula(Cs),eventuallyProp(Cs)) .
```

When all states and all counterexample states have been handled, the first equation is used, returning the pair `< NS : CACHE >` of a node set `NS` and a cache `CACHE`. The second equation deals with *states* in `<` *states* `:` *cxstates* `>` `:` *list* `>` used as the second parameter of `$$layerCheck`, while the third equation deals with *cxstates*. The function `genTermSet` collects all states (say $s^{l+1}_{n_1}, \ldots, s^{l+1}_{n_k}, s^{l+1}_{n_{k+1}}, \ldots, s^{l+1}_{n_m}$ as shown in Fig. 6.1) reachable from the state `T` (say s^{l+1}_1) in the specification `M`, while the function `allCounterExample` collects all counterexample states (say $s^{l+1}_{n_{k+1}}, \ldots, s^{l+1}_{n_m}$ as shown in Fig. 6.1) reachable from the state `T` (say s^{l+1}_1) in the specification `M`.

The function `genTermSet` is defined as follows:

```
eq genTermSet(M,empty,D,Cs) = emptyTermSet .
eq genTermSet(M,T,0,Cs) = emptyTermSet .
eq genTermSet(M,T,D,Cs)
= $genTermSet(metaSearch(M,T,'`{_`}['__[getVarTerm(soupSort(Cs)),
    'depth:_[upTerm(D)]]],nil,'*,unbounded,0),M,T,D,0,Cs) .
eq $genTermSet(failure,M,T,D,N,Cs) = emptyTermSet .
eq $genTermSet(RT,M,T,D,N,Cs)
= getTerm(RT) | $genTermSet(metaSearch(M,T,'`{_`}['__[getVarTerm(soupSort(Cs)),
    'depth:_[upTerm(D)]]],nil,'*,unbounded, N + 1),M,T,D,N + 1,Cs) .
```

The function `metaSearch` is used to collect all states located at the depth `D` reachable from the state `T` in the specification `M`.

The function `allCounterExample` is defined as follows:

```
eq allCounterExample(M,T,F,P) = $allCounterExample(M,T,F,P) .
ceq $allCounterExample(M,T,F,P)
= if (Cx :: Constant)
  then emptyTermSet
  else getCounterExampleState(Cx) |
    $allCounterExample(addEqs(buildEqs(getCounterExampleState(Cx),P),M),T,F,P) fi
if Cx := getCounterExample(M,'modelCheck[T,F]) .
eq $allCounterExample(M,T,F,P) = emptyTermSet [owise] .
```

The function `getCounterExample` extracts a counterexample state by conducting the model checking experiment `'modelCheck[T,F]`. The function `$allCounterExample` finds all counterexamples one by one by adding equations, such as the one

```
eq {(locked: false) (cnt: 2) (depth: 2) (pc[p1]: ws) (pc[p2]: ws)} |= inCs1 = true .
```

we did in Sect. 4.1 with `addEqs(...)`.

To show what is displayed by the tool when a model checking experiment finds a counterexample, we intentionally add the following rewrite rule to the TAS specification:

```
rl [flaw] : {(cnt: 0) (locked: false) (pc[I]: fs) OCs}
  => {(cnt: 0) (locked: true) (pc[I]: ws) OCs} .
```

For the flawed version of TAS, our tool displays the following:

```
Check: failure
Formula: l2-prop.Formula
Cx: counterexample(
    {{cnt: 2 locked: false (pc[p1]: ss)pc[p2]: ss},'start}
    {{cnt: 2 locked: false(pc[p1]: ws)pc[p2]: ss},'start}
    {{cnt: 2 locked: false(pc[p1]: ws)pc[p2]: ws},'wait}
    {{cnt: 2 locked: true(pc[p1]: cs)pc[p2]: ws},'exit}
    {{cnt: 1 locked: false(pc[p1]: fs)pc[p2]: ws},'wait}
    {{cnt: 1 locked: true(pc[p1]: fs)pc[p2]: cs},'exit}
    {{cnt: 0 locked: false(pc[p1]: fs)pc[p2]: fs},'flaw},
    {{cnt: 0 locked: true(pc[p1]: ws)pc[p2]: fs},'fin})
```

The tool displays a counterexample as the Maude LTL model checker does. Once the flawed version of TAS reaches the state

```
{(cnt: 0) (locked: true) (pc[p1]: ws) (pc[p2]: fs)}
```

the state never changes because only the self transition can be taken in the state. Hence, the process `p1` located at `ws` never goes to `cs`.

6.3 Case Studies

We conducted some model checking experiments with our support tool. The experiments were conducted with a docker container running Ubuntu 20.04.3 LTS as a virtual machine that had 2 GB memory and ran on a host machine (an iMac) with 4 GHz processor and 32 GB memory. Four mutual exclusion protocols were employed as examples: (1) Qlock, (2) Anderson, (3) MCS, and (4) TAS. Qlock is an abstract version of the Dijkstra binary semaphore. Anderson is an array-based mutual exclusion protocol invented by Anderson [47]. MCS invented by Mellor-Crummey & Scott is a list-based queueing mutual exclusion protocol whose variants have been used in Java virtual machines [48].

Qlock

Qlock in Algol-like pseudo-code for each process p is as follows:

$$
\begin{array}{ll}
 & \text{“Start Section”} \\
\text{ss} & : \text{enqueue}(\mathit{queue}, p); \\
\text{ws} & : \textbf{repeat until } \text{top}(\mathit{queue}) = p; \\
 & \text{“Critical Section”} \\
\text{cs} & : \text{dequeue}(\mathit{queue}); \\
 & \text{“Final Section”} \\
\text{fs} & : \ldots
\end{array}
$$

queue is a queue of process IDs that is shared by all processes. enqueue, top and dequeue are the standard operators of queues. Initially, *queue* is empty and each process p is located at ss. If p wants to enter the critical section, it puts its ID into *queue* at the end and moves to ws where it waits until the top of *queue* becomes p. If so, it moves to cs. When it leaves cs, it deletes the top element from *queue* and moves to fs.

Anderson

Anderson in Algol-like pseudo-code for each process i is as follows:

$$
\begin{array}{ll}
 & \text{“Start Section”} \\
\text{ss} & : \mathit{place}[i] := \text{fetch\&incmod}(\mathit{next}, N); \\
\text{ws} & : \textbf{repeat until } \mathit{array}[\mathit{place}[i]]; \\
 & \text{“Critical Section”} \\
\text{cs} & : \mathit{array}[\mathit{place}[i]], \mathit{array}[(\mathit{place}[i] + 1)\%N] \\
 & \quad := \text{false}, \text{true}; \\
 & \text{“Final Section”} \\
\text{fs} & : \ldots
\end{array}
$$

where N is the number of processes participating in Anderson. *next* is a natural number variable and *array* is a Boolean array whose size is N. Both *next* and *array* are shared with all processes. $place[i]$ is a natural number variable that is local to process i. fetch&incmod is an atomic instruction that takes a natural number variable x and a non-zero natural number n and atomically does the following: x is set to $(x+1)\%n$ and the old value of x is returned. For two variables x, y and two expressions e_1, e_2, $x, y := e_1, e_2$; is a concurrent assignment that computes e_1 and e_2 and sets x and y to their values, respectively. Initially, each process i is located at ss, *next* is 0, $array[0]$ is true, each $array[j]$ for $j = 1, \ldots, N-1$ is false and each $place[i]$ is 0. If process i wants to enter the critical section, it atomically sets $place[i]$ to *next* and increments *next* modulo N with fetch&incmod, moving to ws where it waits until $array[place[i]]$ becomes true. When so, it moves to cs. When it leaves cs, it sets $array[place[i]]$ and $array[(place[i]+1)\%N]$ to false and true, respectively, moving to fs.

MCS

MCS [48] is also a shared-memory mutual exclusion protocol for multiple processes. Its variants have been used in Java virtual machines and therefore the inventors (John M. Mellor-Crummery and Michael L. Scott) were honored with 2006 Edsger W. Dijkstra Prize in Distributed Computing[1]. MCS [48] for each process p can be described as:

```
      "Start Section"
ss  : ...
l1  : nxt[p] := nop;
l2  : prd[p] := fetch−store(glock, p);
l3  : if prd[p] ≠ nop {
l4  :    lock[p] := true;
l5  :    nxt[prd[p]] := p;
l6  :    repeat while lock[p]; }
      "Critical Section"
cs  : ...
l7  : if nxt[p] = nop {
l8  :    if comp−swap(glock, p, nop)
l9  :       goto fs;
l10 :    repeat while nxt[p] = nop; }
l11 : lock[nxt[p]] := false;
      "Final Section"
fs  : ...
```

glock is a global variable whose type is Bool and initially assigned *nop* that means no process. $nxt[p]$, $prd[p]$, and $lock[p]$ are local variables of each process p whose types are process IDs

[1] https://www.podc.org/dijkstra/2006-dijkstra-prize

Table 6.1: Model checking running performance by Maude and the tool 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

(including *nop*), process IDs (including *nop*), and Bool, respectively. Their initial values are *nop*, *nop*, and false. Although they are local to p, they can be used by any other processes because a shared-memory machine is used. MCS uses the two atomic operators fetch−store and comp−swap. fetch−store$(glock, p)$ assigns *glock* p and returns the old value of *glock*, while comp−swap$(glock, p, nop)$ assigns *glock* *nop* and returns true if *glock* equals p; it just returns false otherwise. Whenever p would like to go to cs, it moves to l1 from ss. It carries out the two assignments at l1 and l2. It then checks if $prd[p] \neq nop$. If so, it goes to l4 from l3; otherwise, it directly goes to cs from l3. It then carries out the two assignments at l4 and l5. It waits at l6 while $lock[p]$ is true. Whenever $lock[p]$ becomes false, it goes to cs from l6. Whenever it would like to exit cs, it goes to l7 from cs. It checks if $nxt[p] = nop$. If so, it goes to l8; otherwise, it goes to l11 from l7. It carries out comp−swap$(glock, p, nop)$ at l8. If comp−swap$(glock, p, nop)$ returns true, it goes to l9 and then fs; otherwise, it goes to l10 from l8. It waits at l10 while $nxt[p] = nop$. Whenever $next[p] \neq nop$, it goes to l11. It carries out the assignment at l11, going to fs. We suppose that each process goes to cs at most once.

We suppose that each process enters the critical section once for all case studies. Let us also consider two atomic propositions `inWs1` and `inCs1`. For Qlock, Anderson, and TAS, `inWs1` and `inCs1` hold in a state if and only if the state matches `{(pc[p1]: ws) OCs}`, namely that process `p1` is at `ws`, and `{(pc[p1]: cs) OCs}`, namely that process `p1` is at `cs`, respectively. For MCS, `inWs1` holds in a state if and only if the state matches `{(pc[p1]: l6) OCs}`, namely that process `p1` is at `l6`. `inCs1` holds in a state if and only if the state matches `{(pc[p1]: cs) OCs}`, namely that process `p1` is at `cs`.

We model checked `inWs1` $\rightsquigarrow$ `inCs1` for Qlock with 8 and 9 processes, for Anderson with 7 and 8 processes, for MCS with 4 and 5 processes, and for TAS with 11 and 12 processes with our support tool and the Maude LTL model checker. The experimental data are shown in Table 6.1. In the third column, NA stands for Non-Available meaning that the model checking experiment was not able to be conducted in that 2GB memory was not enough for the experiment. When

Table 6.2: Model checking running performance by the tool with different layers

Protocol	**#Processes**	**Maude LTL Model Checker**	**Layers**	$L+1$**-DCA2L2MC**
Qlock	10 processes	NA	3	45h 37m
			2 2	6h 31m
			2 2 1	6h 43m
			2 2 2	13h 10m
Anderson	9 processes	NA	3	12h 0m
			2 2	1h 44m
			2 2 1	1h 48m
			2 2 2	3h 28m

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). Thus, Our tool as well as our technique mitigate the state space explosion problem in model checking to some extent.

We conducted some more model checking experiments in which a computer that had a 2.9GHz processor and 32GB memory. In the experiments, we model checked `inWs1` $\rightsquigarrow$ `inCs1` for Qlock with 10 processes and for Anderson with 9 processes. The experimental results are shown in Table 6.2. $d_1\ d_2\ \ldots\ d_L$ in the layers column means that we use $L+1$ layers and the depth of the ith layer is d_i. nil in the layers column means that we used the Maude LTL model checker and did not use our support tool. NA stands for Non-Available meaning that the model checking experiment was not able to be conducted in that 32GB memory was not enough for the experiment. Therefore, our tool alleviates the notorious state space explosion problem to some extent if not perfectly conquering the problem.

6.4 Comparison of the Tool and Maude Model Checker

The model checking experiments were conducted for TAS and MCS with our tool and the Maude LTL model checker. The number of processes was changed from 2 to 12 for TAS, while it was changed from 2 to 5 for MCS. The experimental data are shown in Table 6.3. The experimental data for TAS are plotted as a graph shown in Figure 6.2, while MCS are plotted

as a graph shown in Figure 6.3. The experiments were conducted with a docker container running Ubuntu 20.04.3 LTS as a virtual machine that had 2 GB memory and ran on a host machine (MacBook Air) with 1.6 GHz processor and 16 GB memory.

Table 6.3: Model checking running performance by Maude and the tool with different processes (2GB memory)

Proto	**#Procs**	**Maude**	**Layers**	**#Sub-state Spaces**	**Time (1)**	**Time (2)**	$L+1$-**DCA2L2MC**
TAS	2	0.001s	3 3	5	0.11s	0.71s	0.817s
	3	0.001s		11	0.46s	0.35s	0.81s
	4	0.004s		21	0.90s	1.62s	2.52s
	5	0.022s		36	1.43s	4.81s	6.14s
	6	0.070s		57	2.59s	8.73s	11.32s
	7	0.220s		85	13.47s	13.63s	26.73s
	8	0.798s		121	31.67s	1m 11s	1m 43s
	9	3.249s		166	1m 12s	4m 32s	5m 44s
	10	19.769s		221	2m 42s	43m 45s	46m 27s
	11	2m 28s		287	9m 04s	3h 32m 29s	3h 41m 33s
	12	NA		365	15m 19s	1d 3h 59m 44s	1d 4h 15m03s
MCS	2	0.003s	4 4 4 4	28	0.55s	0.333s	0.883s
	3	0.098s		232	1.38s	2.58s	3.96s
	4	2.042s		1273	36.60s	14.15s	50.75s
	5	NA		5126	7m 39s	1h 12m 43s	1h 20m 21s

Time (1) - Time-taken for generating all sub-state spaces for non-final layers

Time (2) - Time-taken for model checking experiments of all sub-state spaces in the final layer

Based on the graph shown in Figure 6.2 for TAS, when the number of processes is between 2 and 9, the time taken with our tool is not very different from the time take with the Maude LTL model checker; when the number of processes is each of 10 and 11, the former time is greater than the latter time; when when the number of processes is 12, our tool successfully completed the model checking experiment, while the Maude LTL model checker does not due to the state space explosion problem.

For each sub-state space in each non-final layer, we update the formal systems specification on the fly so that we can generate all counterexamples for $\varphi_1 \rightsquigarrow \varphi_2$ or $\Diamond\varphi_2$. This is one overhead introduced by our tool. We thought that the overhead would largely affect the model checking running performance. Looking at the time taken for tackling all sub-state spaces in all non-final layers shown in Table 6.3, when the number of processes is each of 10, 11 and 12, the time taken for tackling the final layer is much greater than the time taken for tackling all non-final layer. This means that the overhead introduced by updating the formal systems specification on the fly is not that large. The reason why it took much time to tackle the final layer would

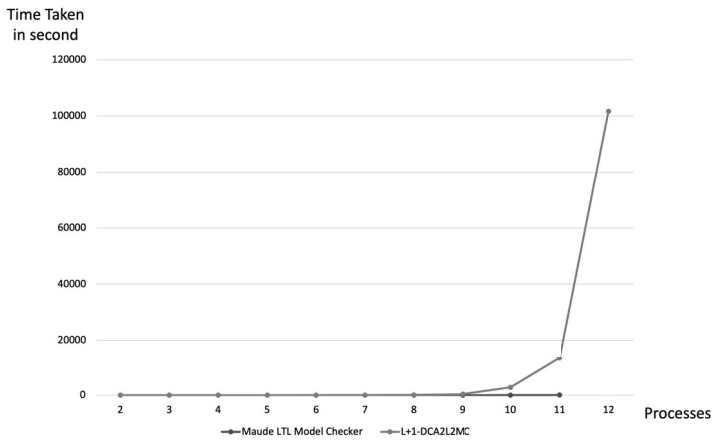

Figure 6.2: Graph plotted for running performance by Maude and the tool for TAS with different processes

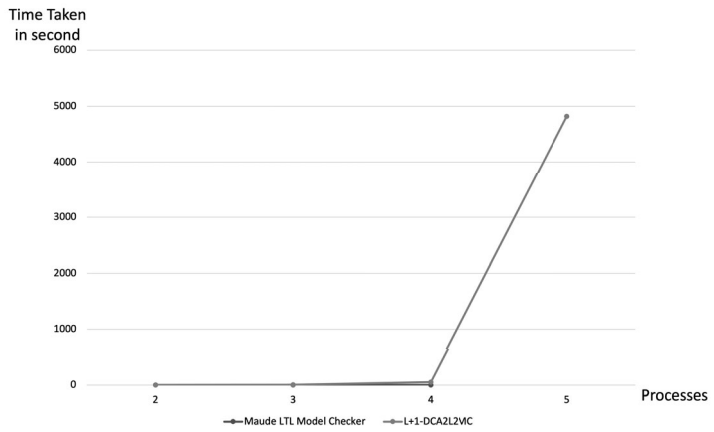

Figure 6.3: Graph plotted for running performance by Maude and the tool for MCS with different processes

be that there are many states duplicated and shared by many sub-state spaces in the final layer for TAS when the number of processes is each of 10, 11 and 12. Each of the processes waiting at ws in TAS has an equal opportunity to enter cs, meaning that TAS has many symmetries from a process point of view. This is why there are many states duplicated and shared by many sub-state spaces in the final layer for TAS.

Based on the graph shown in Figure 6.3 for MCS, when the number of processes is between 2 and 4, the time taken with our tool is not very different from the time take with the Maude LTL model checker; when the number of processes is 5, our tool successfully completed the model checking experiment, while the Maude LTL model checker does not due to the state space explosion problem.

When the number of processes is between 2 and 4, the overhead introduced by our tool does not affect very much the model checking running performance based on the graph shown in

Figure 6.3. Because MCS has a virtual queue to control the order in which the processes waiting at ws will enter cs, MCS does not have many symmetries from a process point of view unlike TAS. This is why we would suppose that there are not so many states duplicated and shared by many sub-state spaces in the final layer for MCS. Although the time taken for tackling the final layer is much greater than the time taken for tackling all non-final layer when there are 5 processes, therefore, we do not think that this is because there are many states duplicated and shared by many sub-state spaces in the final layer.

The technique proposed splits the reachable state space from each initial state into multiple layers, generating multiple sub-state spaces and tackling each sub-state space, and uses an existing model checker, concretely the Maude LTL model checker. Thus, it does not change the computation complexity of the algorithms used by the Maude LTL model checker. At least, it does not essentially improve the computation complexity. If each sub-state space is much smaller than the original reachable state space, the technique/tool may alleviate the state space explosion problem, even though the Maude LTL model checker cannot complete the model checking experiment for the original model checking problem. The tool updates a formal systems specification under model checking on the fly so as to find all counterexamples for each sub-state space in each non-final layer, which would be the biggest overhead introduced by the tool. However, based on the experimental data, the overhead does not affect the model checking running performance very much. For most experiments, especially when the number of processes is large, the time taken for tackling the final layer is (much) greater than the time taken for tackling all non-final layers. For TAS, this would be because there are many states duplicated and shared by many sub-state spaces in the final layer, while for MCS, we do not know the main reason, which should be investigated furthermore and one piece of our future work.

6.5 Summary

This chapter has described the support tool for the $L + 1$-layer divide & conquer approach to leads-to model checking and comparison of the tool and Maude model checker.

The tool has been implemented in Maude with its meta-programming facilities on the top of Full Maude. This chapter has also reported on the case studies in which it is demonstrated that the tool can alleviate the notorious state space explosion problem in model checking to some extent.

Looking at the experimental data as shown in Table 6.2, we find that the number of layers and each layer depth greatly affect the model checking performance. We are supposed to make formal systems specifications so that each of the multiple sub-state spaces to which the original reachable state space is divided has a much smaller number of states than the number of states in the original reachable state space so that we can effectively use the support tool.

For example, we should avoid any long backward state transitions that may hinder some sub-state spaces located at the last layer from having a much smaller number of states than the number of states in the original reachable state space. The formal systems specifications of the four mutual exclusion protocols used in this chapter do not have any long backward state transitions. As one piece of our future work, we need to come up with a way to find a good layer configuration with the tool for a system under verification.

Chapter 7

A Support Tool for the Divide & Conquer Approach to Conditional Stable Model Checking

This chapter describes a support tool for the $L+1$-layer divide & conquer approach to conditional stable model checking ($L+1$-DCA2CSMC) and conducts some experiments with the tool showing that the support tool/technique can alleviate the state space explosion to some extent.

7.1 How the Tool Works in Nutshell

Given the initial KM specification in which the conditional stable property and the layer configuration (a positive natural number list) are written, the model checking experiment outlined in Sect. 5.1 can be be carried out with the support tool as:

```
Maude> in specs/self-stabilization/k-states
...
Maude> in full-maude
...
    Full Maude 3.1 Oct 12 2020

Maude> in solver
...
    L+1 Layers Divide & Conquer Approach to
       Conditional Stable Model Checking

Maude> (initialize[KM-CHECK, init, cstable, OComp, Soup{OComp}])
Initialization: success
    origin-module: KM-CHECK
```

```
    revise-module: KM-CHECK-REVISE
    initial-state: init
    formula-to-check: cstable
    element-sort: OComp
    soup-sort: Soup`{OComp`}

Maude> (check 2 2)
Analyzer:
    current-depth: 4
    depth-list: 2 2
    #node-set: 3
    #states: 0
    #cx-states: 3

Checker: success

Maude>
```

Note that all outputs made by Maude are not written and ... is shown instead.

We first load the formal specification of KM, Full Maude, and our tool in this order into the Maude system. The formal specification has one module called `KM-CHECK` in which KM and the conditional stable property are described. Our tool is then initialized as follows:

```
(initialize[KM-CHECK, init, cstable, OComp, Soup{OComp}])
```

The initial state of KM is denoted `init`. `illegal |-> [] inCs1`, the conditional stable property concerned, is denoted `cstable`. `OComp` and `Soup{OComp}` are the sorts of observable components and their soups, respectively. The revised specification (denoted `KM-CHECK-REVISE`) of KM is made when initializing the tool.

Next we conduct the 3-layer divide & conquer approach to leads-to model checking for KM by using the command `check 2 2`, where each of the first two layer depths is 2 and the final layer is ∞. The tool returns `success`, meaning KM satisfies the conditional stable property under verification.

7.2 Tool Implementation in Maude

The support tool is developed in Maude by using meta-programming on top of Full Maude [46]. A database (called DB) of modules is managed by Full Maude. We manage an extended database called DB−EX. DB−EX saves our way to formalize states (our configurations), data collected at each layer, the initial specification, such as `KM-CHECK`, the revised specification, such as `KM-CHECK-REVISE`, the initial state, the conditional stable property, the eventual property

obtained against the conditional stable property, `OComp`, and `Soup{OComp}`. DB−EX stores the following as well: the current depth whose initial value is 0, a layer configuration (a list of natural numbers, the depths of layers) whose initial value is `nil`, the result whose initial value is `nil`, and the node set (denoted NodeSet) that initially has the initial state alone. NodeSet is one of the most important data structures utilized by the tool.

The result saved in DB−EX is `nil`, `success`, or in the form as follows:

`formula:` fm`, term:` stt`, trace:` log`, cx:` cx

fm is a conditional stable property or an eventual one, stt is a state, the initial one of a sub-state space being tackled for model checking, log is the log that starts with the real initial state and ends with stt, and cx is a local cx discovered while tackling the sub-state space. When conducting a model checking experiment for a sub-state space in middle layers, `nil` is the result saved in DB−EX. The result is `success` iff all model checking experiments are done for all sub-state spaces and no cx states were discovered. If a model checking experiment is conducted for a sub-state space in the final layer whose initial state is s_0 and a local cx cx is discovered, then the result saved in DB−EX is the following:

`formula:` f`, term:` s_0`, trace:` log`, cx:` cx

A global cx can be built from log and cx.

`empty` and `_,_` are the constructors of NodeSet. `empty` is the empty NodeSet. `_,_` is given `assoc`, `comm`, and `id: empty` as its operator attributes. A non-empty NodeSet is expressed as $node_1$, ..., $node_i$, $node_{i+1}$, ... , $node_n$. Each $node_i$ is in the form as $< < stts : cx{-}stts > : lst >$, where $stts$ is a set of non-cx states placed at depth d, $cx{-}stts$ is a set of cx ones placed at d such that $stts \cup cx{-}stts$ is the set of all states placed at d, and lst is a list of pairs of states and natural numbers. NodeSet initially has one element. In the element, $stts$ only has the initial state to which `(depth: 0)` is added, $cx{-}stts$ is empty, and lst is `nil`. When we generally think about a sub-state space (as depicted in Fig. 6.1) that starts with s_1^{l+1} placed at depth $d_1+\ldots+d_l$ and is in layer $l+1$ whose depth is d_{l+1}, where $s_{n_1}^{l+1}, \ldots, s_{n_k}^{l+1}, s_{n_{k+1}}^{l+1}, \ldots, s_{n_m}^{l+1}$ are all states placed at depth $d_1+\ldots+d_l+d_{l+1}$ reachable from s_1^{l+1} and among them $s_{n_{k+1}}^{l+1}, \ldots, s_{n_m}^{l+1}$ are cx ones, $node$ is as:

$$< < s_{n_1}^{l+1} | \ldots | s_{n_k}^{l+1} | : s_{n_{k+1}}^{l+1} | \ldots | s_{n_m}^{l+1} > : < s_1^{l+1} : d_{l+1} > < s_1^{l} : d_l > \ \ldots < s_1^{1} : d_1 > >$$

`_|_` is the constructor of non-empty sets and `__` is the constructor of non-empty lists. lst in the $node$ is as $< s_1^{l+1} : d_{l+1} > < s_1^{l} : d_l > \ldots < s_1^{1} : d_1 >$, where $< s_1^{i} : d_i >$ is the pair of s_1^i and d_i, s_1^i is a state placed at the top of layer i and d_i is the depth of layer i for $i = 1, \ldots, l, l+1$. s_1^{i+1} is reachable from s_1^i for $i = 1, \ldots, l$.

`(check 2 2)` is the command that carries out a conditional stable model checking experiment where three layers are used and each depth of the first two layers is two. A key function utilized by `(check ...)` is `layer-check` defined as:

```
eq layer-check(M, NS, D, N, Cs) = $layer-check(M, NS, D, N, empty,
   < empty-term-set : empty-term-set >, Cs) .
```

M is the specification revised, NS is the node set that is dealt with as the initial states of a layer (e.g. layer l), D is the depth of the layer from the real initial state (e.g. $d_0 + \ldots + d_l$ when it is layer l), N is the depth of layer l (d_l), and Cs is the sort of our configuration (Config-Ex-Set). layer-check deals with each layer (e.g. layer l) except the final layer and returns the node set of the next layer (e.g. layer $l+1$).

We define $layer-check as:

```
eq $layer-check(M,empty,D,N,NS,CACHE,Cs)
= filter-valid-node(filter-by-cx-states(NS,cx-states(CACHE))) .
ceq $layer-check(M,(ND,NS1),D,N,NS2,CACHE,Cs)
= $layer-Check(M,NS1,D,N, union(NS2,ns(NS&CACHE)),cache(NS&CACHE),Cs)
  if NS&CACHE := layer-check-for-stable(M,ND,D,N,empty,CACHE,Cs).
```

The first four and seventh parameters are the same as the five ones of layer-check. The fifth and sixth ones manage the result (the node set) and a cache utilized to evade the same states, respectively, while building nodes in the result. A node set is returned by $layer-check.

We define layer-check-for-stable as:

```
eq layer-check-for-stable(M,< < empty-term-set : empty-term-set > : LL >,
   D,N,NS,CACHE,Cs) = < NS : CACHE > .
ceq layer-check-for-stable(M,< < T | TS1 : TS2 > : LL >,D,N,NS,CACHE,Cs)
= layer-check-for-stable(M, < < TS1 : TS2 > : LL >,D,N,
  union(NS,filter-valid-node(< < (STATES except states(CACHE))
    : (CxSTATES except cx-states(CACHE)) > : (< T : D > LL) >)),
  < (STATES | states(CACHE)) : (CxSTATES | cx-states(CACHE)) >,Cs)
  if < STATES : CxSTATES > := gen-states-for-stable(M,T,D,N,Cs) .
ceq layer-check-for-stable(M, < < empty-term-set : T | TS2 > : LL >,
    D,N,NS,CACHE,Cs)
= layer-check-for-stable(M, < < empty-term-set : TS2 > : LL >,D,N,
    union(NS,filter-valid-node(
    < < empty-term-set : (CxSTATES except cx-states(CACHE)) >
       : (< T : D > LL) >)),
    < states(CACHE) : (CxSTATES | cx-states(CACHE)) >,Cs)
  if CxSTATES := gen-term-set(M, T, D, Cs) .
```

If all non-cx states and all cx ones have been tackled, the first equation is employed to return the pair < NS : CACHE > of a node set NS and a cache CACHE. The second one handles *stts* in < < *stts* : *cx−stts* > : *lst* > utilized as the second parameter of layer-check-for-stable, while the third one treats *cx−stts*. Function gen-term-set gathers all states (e.g. $s^{l+1}_{n_1}, \ldots, s^{l+1}_{n_k}, s^{l+1}_{n_{k+1}}$,

$\ldots, s^{l+1}_{n_m}$ as depicted in Fig. 6.1) reachable from state `T` (e.g. s^{l+1}_1) in the formal specification `M`, while function `gen-states-for-stable` gathers non-cx states (e.g. $s^{l+1}_{n_1}, \ldots, s^{l+1}_{n_k}$) and cx ones (e.g. $s^{l+1}_{n_{k+1}}, \ldots, s^{l+1}_{n_m}$ as depicted in Fig. 6.1) reachable from state `T` (e.g. s^{l+1}_1) in the formal specification `M`.

We define `gen-term-set` as:

```
eq gen-term-set(M,empty,D,Cs) = empty-term-set .
eq gen-term-set(M,T,0,Cs) = empty-term-set .
eq gen-term-set(M,T,D,Cs) = $gen-term-set(metaSearch(M,T,
    '`{_`}['__[get-var-term(soup-sort(Cs)),
    'depth:_[upTerm(D)]]],nil,'*,unbounded,0),M,T,D,0,Cs) .
eq $gen-term-set(failure,M,T,D,N,Cs) = empty-term-set .
eq $gen-term-set(RT,M,T,D,N,Cs) = getTerm(RT) |
   $gen-term-set(metaSearch(M,T,
      '`{_`}['__[get-var-term(soup-sort(Cs)),
      'depth:_[upTerm(D)]]],nil,'*, unbounded, N + 1),M,T,D,N + 1,Cs) .
```

We employ `metaSearch` to gather all states placed at depth `D` reachable from state `T` in the formal specification `M`.

We define `gen-states-for-stable` as:

```
eq gen-states-for-stable(M,T,D,N,Cs) = gen-states-for-stable*(M,Cs,
   gen-state-seqs(M,T,N), < empty-term-set : empty-term-set >) .
eq gen-states-for-stable*(M,Cs,emp, < TS1 : TS2 >) = < TS1 : TS2 > .
eq gen-states-for-stable*(M,Cs, (TL || STL), < TS1 : TS2 >)
= gen-states-for-stable*(M,Cs,STL,
  gen-states-for-stable**(M,Cs,TL,< TS1 : TS2 >)) .
eq gen-states-for-stable**(M,Cs,empty, < TS1 : TS2 >) = < TS1 : TS2 > .
eq gen-states-for-stable**(M,Cs,(T,TL), < TS1 : TS2 >)
= if check-state-seq(M, Cs, (T, TL)) then
  else < TS1 except T : TS2 | T >
    (if T in TS2 then < TS1 : TS2 > else < T | TS1 : TS2 > fi) fi .
eq check-state-seq(M,Cs,empty) = false .
ceq check-state-seq(M,Cs,(T,TL))
= if B then true else check-state-seq(M,Cs,TL) fi
  if B := downTerm(getTerm(metaReduce(M,'_|=_[T, mid-formula(Cs)])),b-error) .
```

Function `gen-state-seqs` used in the first equation is employed to gather all state sequences starting with state `T` (in the formal specification `M`) whose depth is `N`. Function `gen-states-for-stable*` used in the second equation discovers non-cx states and cx ones by inspecting each in those state sequences. If `illegal`, which is derived from `mid-formula(Cs)`, holds in a state in a given state sequence, then the last one (that has the self-transition) in the sequence is considered

as a cx one, added to the set `TS2` of states and deleted from the set `TS1` of states in function `gen-states-for-stable**` in the fifth equation. Otherwise, the last state is considered as a non-cx one and added to `TS1` if `TS2` does not consist of it. Function `check-state-seq` in the two last equations examines each state sequence by using function `metaReduce`. If `illegal` holds at any state `T` in the state sequence in the formal specification `M`, true is returned; otherwise false.

We intend to insert the following rule to the formal specification of KM in order to demonstrate what is shown by our tool when a cx is discovered:

```
rl [flaw] : {(pc[0]: 1) (pc[1]: 1) (pc[2]: 0) (pc[3]: 2) OCs}
=> {(pc[0]: 1) (pc[1]: 1) (pc[2]: 0) (pc[3]: 2) OCs} .
```

The following is shown:

```
Check: failure
Cx: counterexample(
  {{#pc: 4 k-states: 5 (pc[0]: 0)(pc[1]: 2)(pc[2]: 2)pc[3]: 0},'other}
  {{#pc: 4 k-states: 5(pc[0]: 0)(pc[1]: 0)(pc[2]: 2)pc[3]: 0},'bottom}
  {{#pc: 4 k-states: 5(pc[0]: 1)(pc[1]: 0)(pc[2]: 2)pc[3]: 0},'other}
  {{#pc: 4 k-states: 5 (pc[0]: 1)(pc[1]: 0)(pc[2]: 2)pc[3]: 2},'other}
  {{#pc: 4 k-states: 5 (pc[0]: 1)(pc[1]: 0)(pc[2]: 0)pc[3]: 2},'other},
  {{#pc: 4 k-states: 5(pc[0]: 1)(pc[1]: 1)(pc[2]: 0)pc[3]: 2},'flaw})
```

Once we get to the following state:

```
{(k-states: 5) (pc[0]: 1) (pc[1]: 1) (pc[2]: 0) (pc[3]: 2) (#pc: 4)}
```

we will stay there forever because the self-transition can be taken infinitely many times. We cannot, therefore, get to a legitimate state from this one.

7.3 Case Studies

The case studies were conducted with a docker container running Ubuntu 20.04.3 LTS as a virtual machine that had 2 GB memory and ran on a host machine (an iMac) with 4 GHz processor and 32 GB memory. Two mutual exclusion protocols were employed as examples: (1) Qlock and (2) Anderson. Let us assume that each process comes into the critical section for each protocol at most once. Both protocols are revised so that they can be self-stabilizing. We add an observable component `abnorm` that stores a Boolean value which denotes whether the current state is abnormal. `abnorm` is set to true whenever we detect that at least two processes locate at the critical section. `abnorm` can be set back to false if the state has been recovered in which no process locates at the critical section detected.

Table 7.1: Conditional stable model checking running performance by Maude and the tool with 2GB memory restriction

Protocol	**#Processes**	**Maude LTL Model Checker**	**Layers**	$L+1$**-DCA2CSMC**
Qlock	10 processes	23s	2 2	1m 3s
	11 processes	NA		15m 31s
Anderson	5 processes	10s	2 2	44s
	6 processes	NA		38m 8s

For Qlock case study, if `abnorm` is false, p works as described above. If `abnorm` is true, when p would like to exist cs, it gets rid of all elements from q and goes to fs. The initial state of Qlock is set to an illegitimate state in which processes p2, p3, and p5 are placed at cs, q only consists of p3, the other processes are placed at ss and `abnorm` is set to false.

For Anderson case study, if `abnorm` is false, p works as described above. If `abnorm` is true, when p would like to exist cs, it gets rid of all elements from q and goes to fs. The initial state of Anderson is set to an illegitimate state in which processes p2, p3, and p5 are placed at cs, q only consists of p3, the other processes are placed at ss and `abnorm` is set to false.

We take three atomic propositions `inCs1`, `inCs5`, and `inAbnorm` that mean whether processes p1 and p5 are placed at cs or not, and whether the current state is an abnormal state or not, respectively. Model-checking experiments were conducted for `inAbnorm` $\rightsquigarrow \Box\neg$ `(inCs1 /\ inCs5)` wrt Qlock with 10 and 11 processes, and wrt Anderson with 5 and 6 processes by our tool and Maude LTL model checker. The experimental data are exhibited in Table 7.1. $d_1\ d_2\ \ldots\ d_L$ in the *layer* column says that $L+1$ layers are employed and ith layer depth is d_i. `nil` in the *layer* column says that Maude LTL model checker is employed. `N/A` means that the model checking experiment made it impossible to be carried out in that it did not suffice to employ 2 GB memory for the model checking. Thus, the state space explosion can be eased by the tool to a certain scope.

We conduct experiments for KM with the virtual machine above, but only 1GB memory is used. The experimental data are shown in Table 7.2. For KM with 10 processes, 11 processes and 12 processes, both our tool and LTL Maude model checker could complete the model checking experiments. For KM with 13 processes, both our tool and LTL Maude model checker could not complete the model checking experiments because 1GB memory was not sufficient for model checking experiments, leading to the state space explosion. We can see that the verification time of Maude LTL model checker is much smaller than that of our tool. That is because many states are likely to be shared by the sub-state spaces at the final layer. Therefore, our tool may need to explore again many sub-state spaces in those sub-state spaces at the final layer, making the running performance of our tool degrade. For KM with 13 processes, our tool also could not mitigate the state space explosion because our tool needs to store extra information in DB−EX and the size of each sub-state space at the final layer is likely

Table 7.2: Conditional stable model checking running performance by Maude and the tool for KM with different processes (2GB memory)

Protocol	**#Processes**	**Maude LTL Model Checker**	**Layers**	$L+1$**-DCA2CSMC**
KM	10 processes	36m	2 2	1d 4h 58m
	11 processes	1h 9m		2d 23h 54m
	12 processes	7h 58m		22d 14h 1m
	13 processes	NA		NA

to be still big, making the memory consumption high. We need to optimize how to store information in DB−EX from which less memory is required. Besides, we need to find a better layer configuration for KM with 13 processes in which the size of each sub-state space at the final layer is small enough. These are two pieces of our future work.

7.4 Summary

We have described our tool for model checking conditional stable properties in a layered way. We have implemented it in Maude with its meta-programming equipment on Full Maude. We have also reported on the experimental data demonstrating that our tool eases the ill-famed state space explosion to a certain scope.

Our technique [6] is an extension of the related techniques leads-to and eventual model checking [4, 2] in order to deal with desired properties of self-stabilizing systems that can be expressed as conditional stable properties. We are going to implement a parallel version of our tool to utilize the best use of parallel computing because model checking experiments for multiple sub-state spaces are basically independent in the technique. Hence, it is one future direction of ours to build a parallel version of our tool. Moreover, we should carry out more experiments to demonstrate the benefits obtained from our tool.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis focuses on mitigating the state space explosion in model checking, which is the most annoying problem. We have proposed two techniques as well as their support tools. Some experiments have been conducted to demonstrate the usefulness of our techniques/tools.

We have described a divide & conquer approach to leads-to model checking ($L+1$-DCA2L2 MC) so as to mitigate the state space explosion, which is dedicated to leads-to properties. Many systems requirements can be expressed as the properties, such as the lockout freedom property. Hence, it is worth focusing on the properties. The technique divides the original reachable state space into multiple layers, generating multiple sub-state spaces, and checking multiple model checking experiments. We have proved that the multiple smaller model checking problems are equivalent to the original leads-to model checking problem and designed an algorithm based on the theorem from which we can conduct model checking experiments with the technique. Manually conducting model checking experiments with $L+1$-DCA2L2MC is time-consuming and error-prone because many states are considered at each layer. We have then developed a tool that supports the technique. The support tool is implemented in Maude by using meta-programming facilities on the top of Full Maude. We have conducted experiments with the support tool for four mutual exclusions: Qlock, Anderson, MCS, and TAS. The experimental data says that our technique/tool could complete the model checking experiments with a limited memory used, namely 2GB memory, while Maude LTL model checker could not do so because of the state space explosion problem. Moreover, when 32GB memory was used to conduct model checking experiments for Qlock and Anderson, our technique/tool could complete the model checking experiments, while Maude LTL model checker could not complete the model checking experiments once again. That has demonstrated that our tool/technique could mitigate the state space explosion in model checking to some extent.

We have extended the technique used in $L+1$-DCA2L2MC in order to handle conditional stable properties. We then have proposed a divide & conquer approach to conditional stable

model checking (L + 1-DCA2CS2MC) so as to mitigate the state space explosion in model checking, which is dedicated to the conditional stable properties. The properties can be used to express desired properties that self-stabilizing systems should satisfy. Hence, it is meaningful to concentrate on the properties. The original reachable state space made from each initial state is divided into multiple layers, generating multiple sub-state spaces, by the technique. Multiple smaller model checking experiments are then carried out. We have proved a theorem that says that the original conditional stable model checking problem is equivalent to the smaller model checking problems tackled by the technique. An algorithm has been constructed so as to support conducting model checking experiments with the technique. It is inevitable to develop a tool that supports the technique in order to facilitate conducting experiments. The support tool has been implemented in Maude by using meta-programming facilities on the top of Full Maude. We have conducted experiments for three self-stabilization protocols: Qlock, Anderson, and KM. Note that Qlock and Anderson are revised to become self-stabilizing. With a limited amount of memory, namely 2GB memory, our tool could complete the model checking experiments for Qlock and Anderson, while Maude LTL model checker could not complete the model checking experiments because of the state space explosion. For KM, we used only 1GB memory and both our tool and Maude LTL model checker could not complete the model checking experiments in which 13 processes participated in KM. However, our technique/tool still could mitigate the state explosion in model checking to some extent for Qlock and Anderson.

We have conducted model checking experiments with our techniques/tools and compared them with Maude LTL model checker because of the following reason. Maude LTL model checker employs the same model checking algorithm as SPIN [34], one of the most popular model checkers for software systems. There is a report saying that SPIN and Maude LTL model checker are comparable in terms of both running time and memory consumption [16]. This means that whenever Maude LTL model checker encounters the state space explosion problem, making it impossible to carry out model checking experiments, so do SPIN and almost all other model checkers. Therefore, it is meaningful to compare our tool with Maude LTL model checker.

It is very important to verify the correctness of hardware and software systems so as to make them reliable. Model checking is one of the most advanced techniques to verify that a finite-state system satisfies its desired properties. However, the most challenge in model checking is the state space explosion. This thesis has given an effort to mitigate the state space explosion problem to some extent by describing two techniques L+1-DCA2L2MC and L+1-DCA2CSMC as well as their support tools for leads-to and conditional stable model checking, which are two important properties. We expect that the thesis is beneficial to researchers as well as engineers who are interested in using model checking to verify hardware and software systems.

Because LTL formulae for what are called leads-to properties and conditional stable properties in this thesis are quite similar, it seems possible to integrate the two techniques proposed in the thesis. But, it is not that straightforward to do so. For $\varphi_1 \rightsquigarrow \varphi_2$, we make two sets of

states $\boldsymbol{AllS}_L$ and $\boldsymbol{CxS}_L$ for layer $L+1$, where $\boldsymbol{AllS}_L$ is the set of all states located at depth d_L and $\boldsymbol{CxS}_L$ is the set of all counterexample states (of $\varphi_1 \rightsquigarrow \varphi_2$) located at depth d_L, and check $\varphi_1 \rightsquigarrow \varphi_2$ for all states in $\boldsymbol{AllS}_L$ and $\Diamond\varphi_2$ for all states in $\boldsymbol{CxS}_L$. This cannot be used for $\varphi_1 \rightsquigarrow \Box\varphi_2$. To show it, let us try to use this for $\varphi_1 \rightsquigarrow \Box\varphi_2$. Let s be in $\boldsymbol{AllS}_L$ except for $\boldsymbol{CxS}_L$. Let us suppose that there exists a state in which φ_1 is true in a computation up to s, which implies that φ_2 is true in s. It does not suffice to check $\boldsymbol{K}, s \models \varphi_1 \rightsquigarrow \Box\varphi_2$. Let us suppose that there is no state in any paths that start with s such that φ_1 is true and φ_2 is true. If so, $\boldsymbol{K}, s \models \varphi_1 \rightsquigarrow \Box\varphi_2$. From the assumption, however, there exists an earlier state leading to s such that φ_1 is true. Thus, φ_2 needs to keep being true from a state on, but φ_2 does not from the assumption. For $\varphi_1 \rightsquigarrow \Box\varphi_2$, we make two sets of states $\boldsymbol{NcxS}_L$ and $\boldsymbol{CxS}_L$ for layer $L+1$, where $\boldsymbol{CxS}_L$ is the set of all counterexamples (of $\Box\varphi_1$) located at depth d_L and $\boldsymbol{NcxS}_L$ is the set of all states (except for those in $\boldsymbol{CxS}_L$) located at depth d_L, and check $\varphi_1 \rightsquigarrow \Box\varphi_2$ for all states in $\boldsymbol{NcxS}_L$ and $\Diamond\Box\varphi_2$ for all states in $\boldsymbol{CxS}_L$. This cannot be used for $\varphi_1 \rightsquigarrow \varphi_2$. To show it, let us try to use this for $\varphi_1 \rightsquigarrow \varphi_2$. Let s be in $\boldsymbol{CxS}_L$. By definition, s is not in $\boldsymbol{NcxS}_L$. Based on the approach, if $\boldsymbol{K}, s \models \Diamond\varphi_2$ does not hold, we conclude that $\boldsymbol{K}, s \models \varphi_1 \rightsquigarrow \varphi_2$ does not either. But, there may be a state such that φ_2 is true later than a state such that φ_1 is true in each path leading to s, which does not be checked by the approach. So, the algorithm used for $\varphi_1 \rightsquigarrow \varphi_2$ cannot be used for $\varphi_1 \rightsquigarrow \Box\varphi_2$, and the algorithm used for $\varphi_1 \rightsquigarrow \Box\varphi_2$ cannot be used for $\varphi_1 \rightsquigarrow \varphi_2$.

8.2 Future Work

Although $L+1$-DCA2L2MC and $L+1$-DCA2CSMC have been demonstrated as one promising way to mitigate the state space explosion in model checking for leads-to and conditional stable properties, as usual, there are several lines of our future work that we should take into consideration.

The technique used in $L+1$-DCA2L2MC and $L+1$-DCA2CSMC can be used not only for leads-to and conditional stable properties but also for other classes of LTL properties. Thus, one piece of our future work is to extend the technique in order to handle other classes of LTL properties by which many systems requirements can be expressed. For example, we have proposed a divide & conquer approach to until and until stable properties in model checking [50] so as to verify some desired properties that real-time and hybrid systems should satisfy. We plan to build a support tool for until and until stable properties with the technique and conduct some experiments to demonstrate the usefulness of the technique as well as the support tool being built.

Layer configuration affects the running performance of our tools when conducting model checking experiments. Therefore, we need to come up with a way to find a good layer configuration for our tools/techniques. One possible way is the following. We may need to consider

the number of states located at each layer and the verification time to conduct model checking experiments for some sub-state spaces at the final layer. The more layers are used, the more model checking experiments at the final layer need to be conducted and the smaller the size of each sub-state space at the final layer is. Note that when the number of states at each layer is considerably large, then the time taken to generate states is also time-consuming. Our tools currently support displaying the number of states at each layer and the progress of model checking experiments at the final layer (e.g. how many sub-state spaces have been tackled and not tackled yet). If the verification time to conduct model checking experiments for some sub-state spaces at the final layer is too much, it is not likely to be a good layer configuration because the sub-state space is not smaller enough, which may lead to the state space explosion. We should find a layer configuration such that the verification time to conduct model checking for each sub-state space at the final layer is not too much and the number of states at the final layer is not too large. In our techniques, if we use three layers in which each layer depth of the first two layers is d, it is equivalent to using two layers in which the first layer depth is $2 \times d$. Note that the depth of the final layer is unbounded. Hence, one strategy is to use a small depth for each layer and then stack up more layers one by one, which may avoid the state space explosion while generating states at intermediate layers, and carefully check which layer configuration is good with a small deviation. Let us suppose that we use 2 as each layer depth. We may start conducting model checking experiments with our tools with the layer configuration 2. The depth of states located at the beginning of the final layer is currently 2. If the verification time to conduct model checking experiments for some sub-state spaces is too much and the number of states located at the beginning of the final layer is still small, we can use one more layer and the layer configuration is currently 2 2. We can iterative that process until we find a layer configuration such that the verification time to conduct model checking experiments for some sub-state spaces at the final layer is not too much and the number of states located at the beginning of the final layer is not too large. However, we do not need to carry out the process in an iterative way. We can run several experiments independently with different layer configurations at the same time, such as 2, 2 2, 2 2 2, etc. Observing such experiments over a period of time may allow us to find a good layer configuration to conduct model checking experiments with our techniques/tools.

For KM experiments, our tools for $L+1$-DCA2L2MC and $L+1$-DCA2CSMC, respectively, need to spend much amount of time to complete the model checking experiments compared with Maude LTL model checker. Because many states may be likely to be shared by many sub-state spaces at the final layer. Our tools may need to explore again many shared sub-state spaces from sub-state spaces at the final layer, making the running performance of our tools degrade. Processes and machines used in KM, respectively, seem to have a symmetry in which we may change the context between processes/machines and do not affect the behavior of KM. Such a symmetry may make sub-state spaces in the final layer have many shared states, which should be avoided to improve the running performance. Symmetric reduction is one possible

technique to remove replicated structures produced in such symmetric systems by which the state space is reduced and many replicated sub-state spaces are removed. We are going to integrate symmetric reduction and our techniques from which many states are less likely to be shared by many sub-state spaces at the final layer. We expect that can improve the running performance of our tools for KM case studies.

Recently, high-performance computing technologies, such as multicore processors and clusters have emerged. Hence, speeding up model checking with parallelization is one promising approach. $L+1$-DCA2L2MC and $L+1$-DCA2CSMC divide the reachable state space into multiple layers, generating multiple sub-state spaces, and checking multiple smaller model checking experiments. Checking each smaller model checking experiment is independent. Especially, it is completely independent in the final layer, which opens an opportunity to parallelize the tools. Hence, we would improve the running performance of our tools for $L + 1$-DCA2L2MC and $L + 1$-DCA2CSMC by parallelization as one piece of our future work.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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, 1996, doi:10.1007/3-540-61474-5_93.

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

[9] 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.

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

[11] 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.

[12] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer Aided Verification - 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, 2004, doi:10.1007/978-3-540-27813-9_45.

[13] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *Computer Aided Verification - 21st International Conference, CAV 2009*, volume 5643 of *LNCS*, pages 709–714, 2009, doi:10.1007/978-3-642-02658-4_59.

[14] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *10th IFIP WG 10.5 CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, 1999, doi:10.1007/3-540-48153-2_6.

[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] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software*, pages 230–234, 2003, doi:10.1007/3-540-44829-2_16.

[17] 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.

[18] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007, doi:0.1007/s10009-007-0044-z.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. Scheduling constraint based abstraction refinement for weak memory models. In *the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 645–655, 2018, doi:10.1145/3238147.3238223.

[24] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq 2.0: Combining lazy sequentialization with abstract interpretation - (competition contribution). In *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017*, volume 10206 of *Lecture Notes in Computer Science*, pages 375–379, 2017, doi:10.1007/978-3-662-54580-5_26.

[25] 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.

[26] 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.

[27] Kyungmin Bae and José Meseguer. Predicate abstraction of rewrite theories. In *International Conference on Rewriting Techniques and Applications - International Conference on Typed Lambda Calculi and Applications, RTA-TLCA 2014*, volume 8560 of *Lecture Notes in Computer Science*, pages 61–76, 2014, doi:10.1007/978-3-319-08918-8_5.

[28] 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.

[29] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 1954*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, 2000, doi:10.1007/3-540-40922-X_8.

[30] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Computer Aided Verification - 15th International Conference, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, 2003, doi:10.1007/978-3-540-45069-6_2.

[31] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification - 15th International Conference, CAV 2003*, volume 2725 of *LNCS*, pages 1–13, 2003, doi:10.1007/978-3-540-45069-6_1.

[32] 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.

[33] 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.

[34] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997, doi:10.1109/32.588521.

[35] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998, doi:10.1007/978-0-387-34892-6_19.

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

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

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

[39] Shlomi Dolev. *Self-stabilization*. MIT Press, 2000.

[40] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 1974, doi:10.1145/361179.361202.

[41] 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.

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

[43] Lászró Lovász, József Pelikán, and Katalin Vesztergombi. *Discrete Mathematics: Elementary and Beyond*. Undergraduate Texts in Mathematics. Springer, Berlin, Heidelberg, 2003, doi:10.1007/b97469.

[44] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to leads-to model checking. *The Computer Journal*, page 12 pages, 2021, doi:10.1093/comjnl/bxaa183.

[45] Moe Nandi Aung, Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide and conquer approach to eventual model checking. *Mathematics*, 9(4):16 pages, 2021, doi:10.3390/math9040368.

[46] 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.

[47] 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.

[48] 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.

[49] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to conditional stable model checking. In *18th International Colloquium on Theoretical Aspects of Computing, ICTAC 2021*, volume 12819 of *Lecture Notes in Computer Science*, pages 105–111, 2021, doi:10.1007/978-3-030-85315-0_7.

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

First-author Publications

[1] Yati Phyo and Kazuhiro Ogata. Formal specification and model checking of the Walter-Welch-Vaidya mutual exclusion protocol for ad hoc mobile networks. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018*, pages 89–98, 2018, doi:10.1109/APSEC.2018.00023.

[2] Yati Phyo and Kazuhiro Ogata. Analysis of some variants of the Anderson array-based queuing mutual exclusion protocol with model checking and graphical animations. In *5th International Conference on Dependable Systems and Their Applications, DSA 2018*, pages 126–135, 2018, doi:10.1109/DSA.2018.00030.

[3] 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.

[4] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to leads-to model checking. *The Computer Journal*, 2021, doi:10.1093/comjnl/bxaa183.

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

[6] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to conditional stable model checking. In *18th International Colloquium on Theoretical Aspects of Computing, ICTAC 2021*, pages 105–111, 2021, doi:10.1007/978-3-030-85315-0_7.

The Other Co-author Publications

[1] Moe Nandi Aung, Yati Phyo, and Kazuhiro Ogata. Formal specification and model checking of the Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol (S). In Angelo Perkusich, editor, *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*, pages 159–208, 2019, doi:10.18293/SEKE2019-021.

[2] Moe Nandi Aung, Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A divide & conquer approach to eventual model checking. *Mathematics*, 9(4), 2021, doi:10.3390/math9040368.

[3] 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.

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