Master's Thesis

A Technique to Alleviate the State Space Explosion for Eventual Model Checking, Its Support Tool and Case Studies

Moe Nandi Aung

Supervisor Kazuhiro Ogata

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

August, 2023

Abstract

In our research, we address the state space explosion in model checking, which stands as one of the most significant hurdles in this field. To mitigate this problem, we propose a divide & conquer approach to eventual model checking (DCA2EMC).
As the name implies, our approach primarily focuses on handling eventual properties, which are expressed in linear Temporal Logic (LTL) as $\Diamond\varphi$, where φ represents a state proposition. Eventual properties informally say that something will eventually happen. These can be used to express many important software requirements of a system. For example, halting or termination is one important software requirement and this can be formalized by utilizing the eventual properties. Therefore, eventual properties capture many important system requirements, making them a valuable area of focus.
Our divide & conquer approach involves dividing the original eventual model checking problem into multiple smaller model checking problems, each of which is tackled. We have proved a theorem that demonstrates the equivalence of the multiple smaller model checking problems to the original eventual model checking problem. We have also constructed an algorithm based on the theorem in order to build a support tool capable of performing verification of eventual properties in model checking. Additionally, we have developed a tool that supports the proposed approach. Our support tool is developed in Maude, a high-level language and high-performance system which supports both equational and rewriting logic computation.
To demonstrate the effectiveness of our proposed approach, as well as the efficiency of our tool, we have conducted some case studies and experiments. Some limitations of our approach are also discussed in this thesis as well. According to experimental results, our approach can be used to mitigate the state space explosion to a certain scope. In summary, our approach provides a promising solution for tackling the state space explosion in model checking for eventual properties. Through theoretical analysis and evaluations, we highlight the effectiveness of our proposed approach/tool.

Acknowledgements

I would like to express my deepest gratitude and appreciation to the following individuals who have supported and guided me throughout my thesis journey:

First and foremost, I am grateful to my parents for their unwavering love, encouragement, and support. Their constant belief in me has been a source of strength and motivation.

I extend my heartfelt gratitude to my supervisor, Professor Kazuhiro Ogata, for his guidance, expertise, and invaluable insights. His dedication to teaching and research has been instrumental in shaping my academic growth and research skills. I am truly lucky to have the opportunity to work under his supervision.

I would like to express my sincere appreciation to committee members, Professor Daisuke Ishii, Professor Kunihiko Hiraishi, and my second supervisor Professor Toshiaki Aoki for their valuable feedback, constructive criticism, and support throughout the thesis process. Their expertise and suggestions have significantly enhanced the quality of my work.

I am grateful to Assistant Professor Canh Minh Do for his invaluable support and guidance throughout the entire research project and thesis writing process. I am truly grateful to him for always being there to help and provide explanations whenever I needed assistance with the courses I attended.

I would also like to express my gratitude to Professor Saw Sanda Aye, the President of my undergraduate university, for providing me with the opportunity to connect with JAIST and pursue my Master's degree here.

I am also thankful to my friends and classmates who have provided support, encouragement, and a sense of camaraderie throughout my academic journey. Their friendship has made this experience even more rewarding.

Lastly, I would like to acknowledge the contributions of all the individuals who have played a part, directly or indirectly, in the completion of this thesis. Their support, encouragement, and assistance have been instrumental in my success.

I am truly grateful to everyone who has been a part of this journey and has contributed to my personal and academic growth.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, software and hardware play a crucial role in many areas of our daily life, such as education, finance, healthcare, communication, and more. In these applications, it is important to prevent failures because they are not allowed or acceptable. Therefore, the reliability of these is the first job to do to protect our lives from serious injuries. Model checking, as a formal verification technique, holds great promise due to its ability to automate the verification process once concise formal models are constructed [1]. It has found extensive application in various industries, particularly in the realm of hardware verification. However, the state space explosion [2] problem which refers to the exponential growth in the number of states that must be considered during the verification process, rendering it computationally demanding or even infeasible, still remains in it. Despite the development of techniques such as partial order reduction [3], abstraction [4, 5, 6], order binary decision diagrams (OBDD) [7], and compositional reasoning [8, 9] to address this challenge, the state explosion problem persists and requires further attention. This thesis aims to mitigate the state explosion problem in model checking, with a specific focus on eventual properties. Eventual properties informally say that certain events will eventually happen. These properties play a crucial role in formalizing essential software requirements, such as termination or halting conditions. To mitigate the state explosion in the model checking of eventual properties, we have come up with a divide & conquer approach to eventual model checking (DCA2EMC) [1], which divides the original eventual model checking problem into multiple smaller model checking problems and tackles each smaller one. To show the equivalence of multiple smaller model checking problems to the original eventual model checking problem, we have proved a theorem. Based on the theorem, we have constructed the algorithm in order to build a support tool. We have also developed a tool that supports the proposed approach in conducting model checking experiments. We have conducted some case studies to show the power of our approach. In this chapter, we introduce the motivation, our main contribution, and describe our thesis structure.

1.1 Motivation

We have conducted a case study [10] to formally specify an autonomous vehicle intersection control protocol known as the Lim-Jeong-Park-Lee autonomous vehicle intersection control protocol (LJPL) [11]. The LJPL protocol governs the behavior of autonomous vehicles at intersections to ensure safe and efficient traffic flow. To achieve rigorous verification, we used the Maude language [12] as a formal specification language and the Maude LTL model checker as a model checker to model-check the protocol with its desired properties. One crucial property we examined was the starvation freedom property, which guarantees that each vehicle desiring to pass through the intersection will eventually be able to do so. This property can be formally expressed as an eventual formula in Linear Temporal Logic (LTL). In the model checking of the protocol, we aimed to determine whether it satisfies the starvation freedom property.
When we conducted the model checking experiment with five vehicles taking part in the protocol, we were able to complete it quickly using an ordinary computer with 16 GB of memory. However, when we increased the number of vehicles to 13, we could not finish the experiment in a limited amount of time. We could not get the result and the state space explosion occurred. This experience motivated us to propose an approach to mitigate the state space explosion problem in our research.

1.2 Contributions

We focus on mitigating the state space explosion in model checking with eventual properties that many systems should satisfy. The following are our contributions:

- We propose an approach to mitigate the state space explosion in model checking that is dedicated to eventual properties. The approach is called a divide & conquer approach to eventual model checking (DCA2EMC).
- We prove a theorem that shows the equivalence of the original eventual model checking problem to the smaller model checking problems tackled

by our approach. An algorithm is constructed based on the theorem so as to conduct model checking.

- We develop a tool in Maude to support our approach and conduct some experiments showing that our approach can mitigate the state space explosion to some extent.

1.3 Thesis Structure

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

- Chapter 1 introduces model checking and the notorious state space explosion in model checking and our motivation. Moreover, our contributions and the thesis structure are described.
- Chapter 2 investigates some SAT/SMT-based Bounded Model Checking and its extensions used to mitigate the state space explosion problem in model checking. We also compare our technique with some other techniques as well.
- Chapter 3 provides Kripke structure, Linear Temporal Logic (LTL), and Maude.
- Chapter 4 proposes DCA2EMC so as to mitigate the state space explosion in model checking. This chapter sketches out the approach with an example (QLOCK) and proves a theorem that the original eventual model checking problem is equivalent to the smaller model checking problems tackled by our approach. This chapter also describes an algorithm that is constructed from the theorem to conduct model checking for eventual properties with the approach.
- Chapter 5 describes how to use and implement a support tool in Maude for DCA2EMC. Some experiments are conducted to demonstrate that the approach/tool can mitigate the state space explosion to some extent.
- Chapter 6 describes a case study in which the formal specification and model checking of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol is conducted both with Maude LTL model checker and our the support tool.
- Chapter 7 summarizes the main contributions of the thesis and discusses the limitations of our approach/tool and our future work.

Chapter 2

Related Works

In model checking, one of the most prominent challenges is the state space explosion problem. Numerous researchers have proposed various techniques to ease this problem. Among the techniques to ease the problem are partial order reduction [3], abstraction [4, 5, 6], ordered binary decision diagrams (OBDD) [7], and compositional reasoning [8, 9]. This chapter will describe one effective technique called bounded model checking [13] as this technique is related to our divide & conquer approach to eventual model checking and non-exhaustive model checking algorithms offered by the SPIN [14]. We will also compare these techniques with our approach.

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

One effective technique to ease the problem and is related to DCA2EMC is bounded model checking (BMC) [13] based on solvers of satisfiability (SAT) or satisfiability modulo theories (SMT), called 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. DCA2EMC generates multiple sub-state spaces from the original reachable state space from each initial state, where sub-state spaces obtained from intermediate layers are bounded state spaces, although we add a self-transition for each state placed at the bottom of the sub-state spaces. Therefore, DCA2EMC can be considered an extension of SAT/SMT-based BMC, but we never use any SAT/SMT solvers.

Another extension of SAT/SMT-based BMC is k-induction [15, 16]. 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 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. DCA2EMC does not use any inductions but splits the original state space into multiple sub-state spaces that can be regarded as a specific instance of induction. DCA2EMC exhaustively splits the reachable state space from each initial state into multiple sub-state spaces.

Yet another extension of SAT-BMC is model checking with Craig interpolants [17]. 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-based 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 DCA2EMC.

2.2 Non-exhaustive Model Checking

In traditional exhaustive model checking, each reachable state of the system is stored in memory. If each state requires S bytes of memory and there are M bytes of memory available, then the model checker can store up to M/S states. If the total number of reachable states R exceeds M/S, the model checker will exhaust its available memory before exploring all reachable states. SPIN [14] which is a powerful and widely-use model checker for the formal verification of distributed systems offers techniques called bit state hashing [18] and partial order reduction [19].

The bit state hashing technique allows SPIN to explore a much larger portion of the state space with the same amount of memory. Instead of storing the entire state in memory, bit state hashing stores just a hash of the state, using two bits of memory for each state. The hash functions are used to compute the bit address. SPIN utilizes a large bit-state array or bit vector to prevent state duplication. Initially, all array values are set to zero which indicates that no states have been visited yet. As SPIN explores the state space of the system under verification, it encounters new states. For each new state, SPIN calculates the bit address of the state using one or more hash functions. The calculated hash addresses are used as indices in the bit-state array. For each hash value, the corresponding bit in the bit-state array is set to 1. This marks the state as visited. If multiple hash functions are used, then multiple bits in the array will be set to 1 for each state. When SPIN encounters a state, it checks if the state has been visited before by computing the state's hash address and checking the corresponding bit of that state in the bit-state array. If all bits are 1, then the state has been visited before; otherwise, it's a new state. If a hash collision occurs (i.e., the calculated hash addresses for two different states have the same hash address), SPIN simply ignores the collision and proceeds with the model checking. The model checking process continues until all reachable states have been explored, or until a property violation is found. At the end of the process, if no violations have been found and all reachable states have been explored, the system under verification is considered to satisfy the checked properties. The effectiveness of bit-state hashing in SPIN depends on several parameters: the size of the bit-state array, the number of hash functions used, and the search strategy used to explore the state space. Adjusting these parameters can impact the memory usage and accuracy of the verification process. The primary benefit of bit-state hashing is its memory efficiency, which enables SPIN to explore large state spaces that would not fit into memory. Note that it is non-exhaustive model checking where many states may be overlooked. We may adopt the idea of the bit-state hashing from

SPIN for DCA2EMC to handle large state spaces in order to quickly find a counterexample if any at the cost of non-exhaustive model checking.

Partial order reduction [3] is a method to reduce the state spaces of the concurrent systems to be examined. The main idea behind it is to identify independent or commutative transitions, where the order of their occurrence does not impact the overall behavior of the system. It is used to generate the reduced model that includes all behaviors of the original system needed to verify a specific property. SPIN uses the combination of this method and on-the-fly model checking [19] to reduce the number of reachable states that must be examined to complete verification. Assume S is the set of states, $T \subseteq S \times S$ is the set of transitions over states and $\mathbf{enabled}(s) \subseteq T$ is the set of all transitions that can be taken from state s. $\mathbf{ample}(s)$ is a subset of enabled transitions, $\mathbf{ample}(s) \subseteq \mathbf{enabled}(s)$. The reduced model is constructed by selecting the $\mathbf{ample}(s)$ at each state in the system. SPIN computes an ample set of transitions that can be executed next. Instead of exploring all enabled transitions, SPIN only explores the transitions in the ample set. This significantly reduces the number of interleavings that need to be considered, especially for mutual exclusion protocols. The reduction is based on the observation that the validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search [14]. The effectiveness of partial order reduction depends on the level of concurrency and independence of transitions in the system, as well as the type of properties being checked and the selection of the ample sets. The benefits of partial order reductions include the minimization of state spaces and the enhancement of efficiency in terms of time and memory utilization because it does not need to explore the whole state space.

Our approach DCA2EMC offers a different strategy to tackle the state explosion problem. DCAEMC splits the reachable state space into multiple layers, generating multiple smaller sub-state spaces. It tackles smaller sub-state spaces layer by layer sequentially to verify the eventual property. Therefore, it can reduce the overall complexity, making the best use of the memory, and making the verification process more efficient. All these techniques aims to mitigate the state space explosion problem. DCA2EMC splits the original problem into smaller problems to manage the use of the memory. Partial order reduction in SPIN reduces the number of interleavings of independent transitions that are considered during model checking. Bit state hashing in SPIN allows to explore the larger portion of the state spaces in a non-exhaustive way. Therefore, it is worth trying although each technique has its own strengths and weaknesses.

Chapter 3

Preliminaries

This chapter describes Kripke structures, Linear Temporal Logic, and Maude. The support tool is implemented in Maude with its meta-programming facilities.

3.1 Kripke Structure

A Kripke structure $K \triangleq < S, I, T, A, L >$ consists of the following: a set S of states, a set $I \subseteq S$ of initial states, a left-total binary relation $T \subseteq S \times S$ overs states, a set of atomic propositions, and a labeling function whose types is $S \to 2^A$. For a given state $s \in S$, L(s) is the set of atomic propositions that hold in s. Each element $(s, s') \in T$ is called a (state) transition and may be written $s \to_K s'$ or $s \to s'$.
We define some notations as:

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

where i and j are any natural numbers. Note that $\pi^{(0,j)} = \pi_j$. A path π of K is called a computation of K if and only if $\pi(0) \in I$. We use $P_{(K,s)}$, where $s \in S$, to denote the set of paths that start with s. We use $P^b_{(K,s)}$, where b is a natural number, to denote the set of π_b such that π is an element of $P_{(K,s)}$. We use $P^\infty_{(K,s)}$ to denote $P_{(K,s)}$.

3.2 Linear Temporal Logic (LTL)

Let p be an atomic proposition and then the syntax of an LTL formula φ is defined as:

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

We use $\mathcal{F}$ to denote the set of all LTL formulas. We inductively define $K, \pi \models \varphi$ for $\pi \in \mathcal{P}$ and $\varphi \in \mathcal{F}$ as:

- $K, \pi \models \top$
- $K, \pi \models p$ iff $p \in L(\pi(0))$
- $K, \pi \models \neg\varphi_1$ iff $K, \pi \not\models \varphi_1$
- $K, \pi \models \varphi_1 \vee \varphi_2$ iff $K, \pi \models \varphi_1$ and/or $K, \pi \models \varphi_2$
- $K, \pi \models \bigcirc\, \varphi_1$ iff $K, \pi^1 \models \varphi_1$
- $K, \pi \models \varphi_1\, \mathcal{U}\, \varphi_2$ iff there exists a natural number i such that $K, \pi^i \models \varphi_2$ and for all natural numbers $j < i$, $K, \pi^j \models \varphi_1$

where φ_1 and φ_2 are LTL formulas. Then, $K \models \varphi$ iff $K, \pi \models \varphi$ for all computations $\pi \in \mathcal{C}$ of K. $\bigcirc$ and $\mathcal{U}$ are called the next temporal connective and the until temporal connective, respectively.

The other logical and temporal connectives are defined as:

$\bot \triangleq \neg\top$
$\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2)$
$\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$
$\Diamond\, \varphi \triangleq \top\, \mathcal{U}\, \varphi$
$\Box\, \varphi \triangleq \neg(\Diamond\, \neg\varphi)$

$\Diamond$ is the eventual (or eventually) temporal connective and $\Box$ is the always temporal connective. State propositions are classical propositions, which do not contain temporal connectives at all, and then the first state $\pi(0)$ can judge if state propositions are satisfied by a path π. We call properties that can be formalized as $\Diamond\varphi$ eventual properties, where φ is a state proposition, in this thesis.

State propositions are LTL formulas such that they do not have any temporal connectives.

Proposition 1. Let K be any Kripke structure. If φ is any state proposition, then $(K, \pi \models \varphi) \iff (K, \pi' \models \varphi)$ for any paths π and π' of K such that $\pi(0) = \pi'(0)$.

Proof. The first state $\pi(0)$ decides if $K, \pi \models \varphi$ holds.

Eventual properties are those that are expressed in the form of $\Diamond\varphi$, where φ is an LTL formula. In this thesis, furthermore, we give the constraint to φ: φ is a state proposition.
Let $K, s \models \varphi$, where $s \in S$ be $K, \pi \models \varphi$ for all $\varphi \in P_{(K,s)}$. Note that $K, s \models \varphi$ for all $s \in I$ is equivalent to $K \models \varphi$. Let $K, s, b \models \varphi$, where $s \in S$ and b is a natural number or ∞, be $K, \pi \models \varphi$ for all $\pi \in P^b_{(K,s)}$. Note that $K, s, \infty \models \varphi$ is $K, s \models \varphi$.
Some logical connective are abused for $K, \pi \models \varphi$ as follows:

- $(K, \pi \models \varphi) \wedge (K', \pi' \models \varphi') \triangleq K, \pi \models \varphi$ and $K', \pi' \models \varphi'$
- $(K, \pi \models \varphi) \vee (K', \pi' \models \varphi') \triangleq K, \pi \models \varphi$ and/or $K', \pi' \models \varphi'$
- $(K, \pi \models \varphi) \Rightarrow (K', \pi' \models \varphi') \triangleq K, \pi \models \varphi$, then $K', \pi' \models \varphi'$
- $(K, \pi \models \varphi) \iff (K', \pi' \models \varphi') \triangleq K, \pi \models \varphi$ if and only if $K', \pi' \models \varphi'$

In this thesis, we provide a comprehensive summary of the notations or symbols utilized in three tables, namely Tables 1-3. Table 3.1 presents the notations or symbols pertaining to paths, Table 3.2 outlines the notations or symbols associated with sets of paths, and Table 3.3 encompasses the notations or symbols relevant to satisfaction relations. The purpose of these tables is to provide a clear and concise reference for understanding the various notations and symbols employed throughout the thesis.

Table 3.1: Descriptions of path notations

Notation	**Description**
π	A path; an infinite sequence $s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$ of states such that $s_i \rightarrow_K s_{i+1}$ for each i; if s_0 is an initial state, it is called a computation
$\pi(i)$	The ith state s_i in π
π^i	The postfix $s_i, s_{i+1}, \ldots$ obtained by deleting the first i states $s_0, s_1, \ldots, s_{i-1}$ from π
π_i	$s_0, s_1, \ldots, s_i, s_i, \ldots$ Constructed by first extracting the prefix $s_0, s_1, \ldots, s_i$, the first $i+1$ states from π
π_∞	$s_0, s_1, \ldots, s_i, s_{i+1}$, the same as π
$\pi^{(i,j)}$	If $i \leq j$, then $s_i, \ldots, s_j, s_j, \ldots$, the same as $(\pi^i)_{(j-i)}$; otherwise, $s_i, s_i, \ldots$, the infinite sequence in which only s_i occurs infinitely many times
$\pi^{(i,\infty)}$	$s_i, s_{i+1}, \ldots$, the same as π_i
π^i_j	The same as $\pi^{(i,j)}$

Table 3.2: Descriptions of path-set notations

Symbol	Description
P_K	The set of all paths of K
$P_{(K,s)}$	The set of all paths π of K such that $\pi(0)$, the 0th state of the path π, is s
$P^b_{(K,s)}$	The set of all paths π_b such that $\pi \in P_{(K,s)}$
$P^{\infty}_{(}K,s)$	The same as $P_{(K,s)}$

Table 3.3: Descriptions of satisfaction relation $\models$ notations(or symbols), where b is a natural number

Symbol	Description
$K, \pi \models \varphi$	An LTL formula φ holds for a path π of K
$K \models \varphi$	An LTL formula φ holds for all computations of K
$K, s \models \varphi$	An LTL formula φ holds for all paths in $P_{(K,s)}$
$K, s, b \models \varphi$	An LTL formula φ holds for all paths in $P^b_{(K,s)}$
$K, s, \infty \models \varphi$	The same as $K, s \models \varphi$

3.3 Maude

We use Maude [12] which is a powerful specification and programming language that is based on rewriting logic. It serves as a direct successor to OBJ3, an order-sorted algebraic specification language. In Maude, terms play a fundamental role as the basic building blocks and represent the data used in specifications and programs. Various entities such as states, components of states, atomic propositions, and LTL formulas are expressed using terms in Maude.
One of the key features of Maude is its ability to describe state transitions through the use of rewrite rules. These rules define how states can be transformed from one form to another, enabling the specification and modeling of dynamic behavior. Furthermore, Maude supports modularity and extensibility through its module system. Specifications and programs can be organized into modules, allowing for hierarchical structuring and reuse of components. The module system enables the development of complex systems by providing mechanisms for composition, parameterization, and encapsulation.
In summary, Maude offers a rich and expressive environment for specifying and verifying systems. Its foundation in rewriting logic, combined with its module system and formal analysis capabilities, makes it well-suited for both formal verification and prototyping of complex software systems.

3.4 Formal Specification and Model Checking

Formal Specification

In the context of expressing states, a state is represented as a braced soup, which is an associative-commutative (AC) collection of name-value pairs. Observable components, denoted as oc1, oc2, oc3, are the individual name-value pairs within a state. Thus, a state can be expressed as a braced soup constructed using the juxtaposition operator: oc1 oc2 oc3. To specify state transitions, we employ rewrite rules within the Maude programming/specification language. Maude is based on rewriting logic and offers flexible means for defining complex systems. It also includes model checking capabilities such as an LTL model checker.

In Maude, rewrite rules can be presented as unconditional rewrite rules and conditional rewrite rules, following the forms in Maude

$$\begin{aligned}
\text{rl } [\textit{Label}] &: \textit{Term-1} \Rightarrow \textit{Term-2}\ [\textit{StatementAttributes}].\\
\text{crl } [\textit{Label}] &: \textit{Term-1} \Rightarrow \textit{Term-2}\\
&\qquad\qquad \text{if}\langle \textit{Condition-1}\rangle \wedge \ldots \wedge \langle \textit{Condition-k}\rangle\\
&\qquad\qquad [\langle \textit{StatementAttributes}\rangle].
\end{aligned}$$

respectively. In the first case, $Term - 1$ and $Term - 2$ are the same kind, which may contain variables. Intuitively, a rule describes a local state transition in a system: anywhere in the distributed state where substitution instance $\sigma(Term - 1)$ of the lefthand side is found, a local transition of that state fragment to the new local state $\sigma(Term - 2)$ can take place. And if many instances of the same or of several rules can be matched in different nonoverlapping parts of the distributed state, then, all of them can fire concurrently. In the second case, the condition can consist of a single statement or can be a conjunction formed with the associative connective $\wedge$. Conditions can also contain rewrite expressions. Furthermore, equations, memberships, and rewrites can be intermixed in any order. As for the functional modules, some of the equations in conditions can be either matching equations or abbreviated Boolean equations.

In our protocol specification, rewrites rules are used to specify concurrent/distributed systems from which some protocols used in this thesis are specified. The specifications of the case study protocols will demonstrate the practical application and utilization of these rewrite rules and conditional rewrite rules. Through the use of these formal specification constructs, we can effectively capture the desired behavior, state transitions, and constraints of the protocol, ensuring its correctness and adherence to the specified requirements.

Model Checking

In Maude, two levels of specification can be distinguished.

- a system specification level, provided by the rewrite theory specified by that system module which defines the behavior of the system, and
- a property specification level, given by some property (or properties) φ that we want to state and prove about our module.

After defining the system specification and a property specification for the system, we can use Maude LTL model checker to model check the system. In order to use Maude LTL model checker, Maude supports **modelCheck** command to model check the system. It automates the model checking process by encapsulating the analysis of the system, the property to be verified (typically expressed in linear Temporal Logic LTL), and the exploration of the state space. The command **modelCheck(init,** φ**)** checks whether the system, starting from the initial state **init** that we defined, satisfies the LTL formula φ, which can be $\Diamond p$ as our research is focused on the eventual property. In summary, the **modelCheck** command conducts model checking experiments with predefined strategies and algorithms tailored for efficient state space exploration and verification of LTL properties.

3.5 Meta-Programming

Maude supports not only equational logic but also 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 [12]. 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 $\overline{\mathcal{R}}$ as terms t, t', and any pair $(\mathcal{R}, t)$ as a term $\langle\overline{\mathcal{R}}, t\rangle$. Because $\mathcal{U}$ is representable in itself, we can achieve a reflective tower

with an arbitrary number of levels of reflection:
$\mathcal{R} \vdash t \rightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \iff \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \bar{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle$...

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

- in the module **META–TERM**, Maude terms are 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 treats Maude's specifications as terms and allows for advanced manipulation and analysis. A meta-program is a program that takes the Maude specification as input and performs some useful computations, such as adding or revising more specifications or analyzing the specifications using the meta functions. One key application of meta-programming is in the development of support tools. These tools leverage the reflective capabilities of Maude to perform automated analysis, verification, or model checking on specifications. By treating specifications and properties as data, the tools can apply various analysis techniques and algorithms to verify desired properties. Our support tool takes a Maude specification and the desired property as data in the form of terms. It utilizes the proposed technique to perform model checking and determine whether the specification satisfies the desired property. The meta-programming capabilities of Maude provide the flexibility and expressiveness to perform the analysis effectively and efficiently.

Chapter 4

A Divide & Conquer Approach to Eventual Model Checking

This chapter presents the theoretical concept of our approach called "A divide & conquer approach to eventual model checking" (DCA2EMC), followed by the proof and a detailed explanation of the algorithm. To illustrate our approach, we utilize a simple mutual exclusion protocol called Qlock. The basic idea of the proposed technique is that an eventual model checking problem is divided into multiple smaller model checking problems and each smaller model checking problem is tackled so as to tackle the original eventual model checking experiment. We will prove a theorem that tackling each smaller model checking problem is equivalent to tackling the original eventual model checking problem.

4.1 Outline of the Technique

The technique splits each infinite state sequence $s_0^1, ..., s_{n_1}^1, ..., s_0^i, ..., s_{n_i}^i, ..., s_0^m,$ ... (generated from a Kripke structure K) into multiple m sub-sequences, for each $s_0^i, ...s_{n_i}^i$ (for $i = 1, ..., m-1$) of all the sub-sequence except for the final one we add the final state $s_{n_i}^i$ infinitely many times to the end, generating the infinite sequence $s_0^i, ..., s_{n_i}^i, s_{n_i}^i, ...$ and conduct model checking experiment with the eventual properties which can be expressed in LTL as $\Diamond\varphi$ where φ

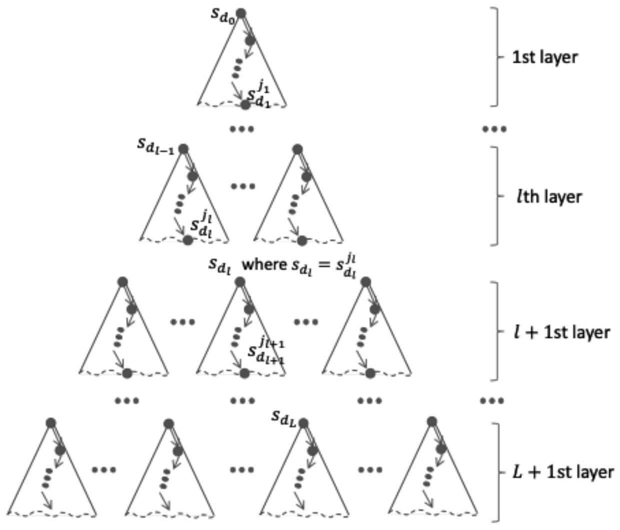

Figure 4.1: A Divide & Conquer Approach to Eventual Model Checking

is a state proposition for each infinite sequence in each layer. $s^i_{n_i}$ is a state located at the bottom of layer i or at the beginning of the next layer i. If $(K, s^i_0, ..., s^i_{n_i}, s^i_{n_i}, ... \nvDash \Diamond\varphi)$, then the last state of the path $s^i_{n_i}$ is collected as a `counterexample state (cx)` at layer i. 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 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 space explosion. Our technique makes it possible to use any existing Linear Temporal Logic(LTL) model checking algorithm and then any existing LTL model checker. We will prove a theorem that the original eventual model checking problem for the original infinite sequence is equivalent to the multiple model checking problems for the multiple infinite sequence. We design an algorithm according to our theorem to build a support tool for our technique. We conduct experiments using some case studies to show the effectiveness of our technique. First, we use a simple mutual exclusion protocol to outline our technique in this chapter.
A mutual exclusion protocol named ‘Qlock’, which serves as an abstract version of Dijkstra’s binary semaphore is used to design our proposed technique. The primary goal of this protocol is to guarantee that a process can successfully enter the critical section, perform essential tasks within it, exit

the critical section, and ultimately reach the final section. This guarantee is expressed as an eventual property which can be expressed as $\Diamond$, emphasizing the protocol's ability to enable a process's uninterrupted progression through the specified sections.

Qlock for each process p can be described as :

```
"start-section"
ss : enq(q, p);
ws : while until top(q) = p;
"critical-section"
cs : deq(q);
"final-section"
fs : ...
```

q is an atomic queue of process IDs shared by all processes. `enq`, `top`, and `deq` are the atomic operators of atomic queues. q is initially empty and each process p is located at `ss`. When p would like to enter the `cs`, it adds its ID into q and goes to `ws`. It waits at `ws` while the top of q is not p. Whenever the top of q becomes p, p enters the `cs`. When it exits cs, it removes the ID of p from q and goes to `fs`. We suppose that each process enters the `cs` at most once. In this case study, the property of interest is the eventual arrival of a process at the `fs` state. This property can be formalized as an eventual property, denoting the guarantee that, given sufficient time, each process will reach the `fs` state.

4.1.1 Formal Specification and Model Checking of *Qlock*

In the context of Qlock, when there are 'n' processes participating, each state in S_{Qlock} can be formalized as follows:

$$(q : q) \quad (\text{loc}[p_1] : l_1) \quad \ldots \quad (\text{loc}[p_n] : l_n) \quad (\#\text{ps} : x)$$

The state consists of a tuple containing the values of different components: The value q represents the atomic queue and is stored in the q observable component. Initially, q is an empty queue denoted as 'empq'. For each process p_i, the place is saved in the l_i label of the loc[p_i] observable component. Initially, l_i is set to ss. The number of processes that have not yet reached the fs state is saved in the x of the #ps observable component. Initially, x is set to n to represent the total number of processes.

Consider the scenario where two processes, denoted as p1 and p2, participate in the *Qlock* mutual exclusion protocol.

```
    {(q: empq) (loc[p1]: ss) (loc[p2]: ss) (#ps:2)}
```

I_{Qlock} has one state init. The transition between state T_{Qlock} can be described in Maude as rewrite rules:

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

The rewrite rules in the *Qlock* protocol are named `start`, `wait`, `exit`, and `fin`. Maude variables `Q`, `I`, `N`, and `OCs` represent queues of process IDs, process IDs, natural numbers, and observable component soups, respectively. The `dec` operation is used to decrement a non-zero natural number by one, returning 0 if applied to 0. The constructor `_|_` represents non-empty queues of process IDs. The rule `start` states that if the location of the process `I` is in `ss`, then its ID is put into `Q` at the end and moves to `ws`. The rule `wait` states that if the location of the process `I` is in `ws` and the ID of the top of the queue is equal to the ID of the process `I`, then the process `I` enters the `cs`. The rule `exit` states that if the location of the process `I` is in `cs`, then the queue removes the ID of the process `I` and decrements the total number of processes `ps`, and the location of the process `I` is moved to `fs`. The rule `fin` says that if the natural number `N` stored in `#ps` is 0, a self-transition occurs. The rule `fin` is used to make the transitions total.

Consider a state formalized as

```
    {(q: p1 | p2) (loc[p1]: ws) (loc[p2]: ws) (#ps: 2)}.
```

The `wait` rule can be applied to modify the state to

```
    {(q: p1 | p2) (loc[p1]: cs) (loc[p2]: ws) (#ps: 2)}.
```

Figure 4.2 illustrates the reachable state space formed by S_{Qlock}, I_{Qlock}, and T_{Qlock}. In total, there are 16 reachable states.

Let's consider an atomic proposition represented by `inFs1`. As a result, P_{Qlock} is said to possess `inFs1`. We define L_{Qlock} in the following manner:

```
    eq {(loc[p1]: fs) OCs} |= inFs1 = true .
    eq {OCs} |= PROP = false [owise] .
```

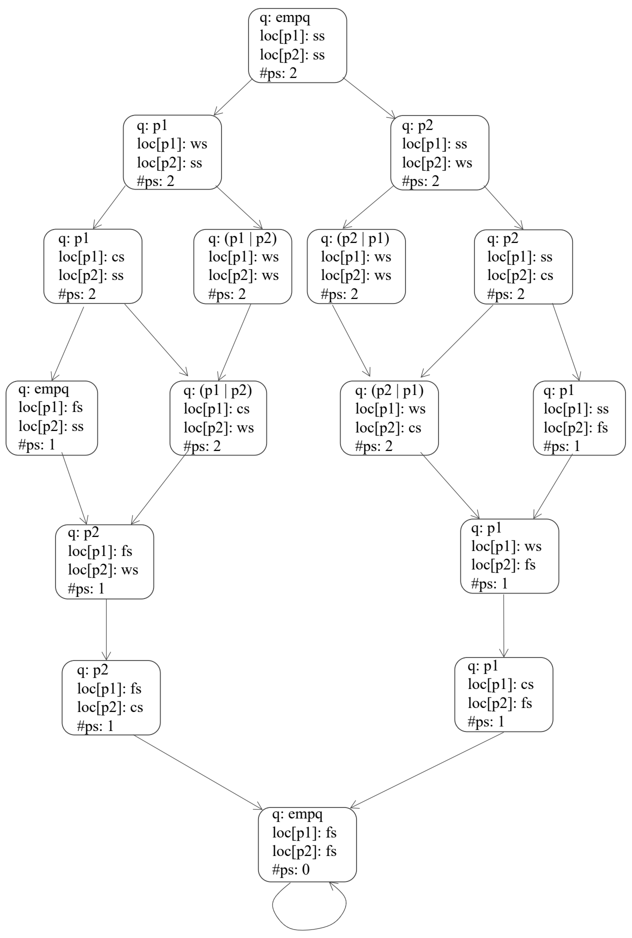

Figure 4.2: The reachable state space of Qlock

OCs and PROP are the observable components of soups and atomic propositions. According to the definitions, $L_{Qlock}(s)$ consists of `inFs1` if a state s has process p1 placed at fs. Otherwise, $L_{Qlock}(s)$ is empty.

```
modelCheck(init, <> inFs1)
```

¡¿_ is the Maude operator that expresses $\Diamond$. We can use the above command to check whether a process can go to the critical section, do some important tasks in the section, leave the section, and finally reach the final section.

During model checking, when we examine this property using a model that involves two processes, no counterexample (cx) is found. As a result, we can conclude that Qlock satisfies the given property. It quickly completes to model check $\Diamond$inFs1 for Qlock when there are five processes, finding no counterexample. It is, however, impossible to model check the same property for Qlock where there are 9 processes because of the state space explosion problem. In the next chapter, we will describe how we use our approach/tool to handle this scenario.

4.1.2 Utilizing Our Technique for Model Checking of *Qlock*

Although the model consisting of two processes can be analyzed using the Maude LTL model checker, we use it to outline how DCA2EMC works in detail. The reachable state space depicted in Figure 4.2 is divided into three layers, represented by Figure 4.3, Figure 4.4, and Figure 4.5, respectively. These figures correspond to the first, second, and third layers, respectively. We create six sub-state spaces, each containing a maximum of seven states, while the original reachable state space consists of 16 states. The main concept behind DCA2EMC is to create sub-state spaces with a smaller number of states compared to the original reachable state space.
According to our DCA2EMC approach, it is necessary to revise the formal specification of *Qlock*. Each state should be changed as follows:

$$(q : q) \quad (\mathrm{loc}[p_1] : l_1) \quad \ldots \quad (\mathrm{loc}[p_n] : l_n) \quad (\#\mathrm{ps} : x) \quad \mathrm{depth} : d)$$

We have added one observable component called depth to manage the depth information. Therefore, the transition rules are changed as follows:

```
crl [start] : {(q: Q) (loc[I]: ss) (depth: D) OCs}
    => {(q: (Q | I)) (loc[I]: ws) (depth: (D + 1) OCs}
    if D < Bound .
crl [wait]  : {(q: (I | Q)) (loc[I]: ws) (depth: D) OCs}
    => {(q: (I | Q)) (loc[I]: cs) (depth: (D + 1) OCs}
    if D < Bound .
crl [exit] : {(q: Q) (loc[I]: cs) (#ps: N) (depth: D) OCs}
    => {(q: deq(Q)) (loc[I]: fs) (#ps: dec(N))
    (depth: (D + 1) OCs} if D < Bound .
crl [fin] : {(#ps: 0) (depth: N) OCs}
    => {(#ps: 0) (depth: (D + 1)) OCs} if D < Bound
crl [stutter] : {(depth: D) OCs}
    => {(depth: D) OCs} if D >= Bound .
```

Let D be a Maude variable, where the sort of D is natural numbers (N). Let Bound be a Maude constant, where the sort of Bound is also natural numbers (N). For the given example, the value of Bound is 2.

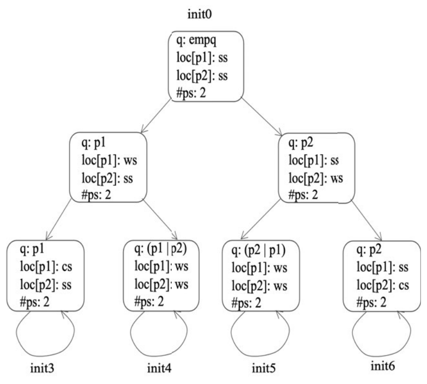

Figure 4.3: Layer 1

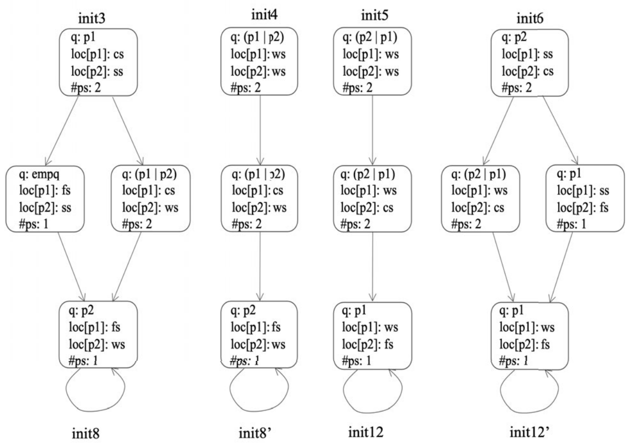

Figure 4.4: Layer 2

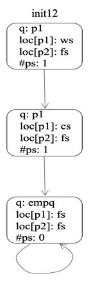

Figure 4.5: Layer 3

Let "init" denote the following state:

```
init = {(q: empq),(#ps: 2),(depth: 0),
       (loc[p1]: ss), (loc[p2]: ss)}
```

We are required to collect all "cx" states placed at depth 2 that are reachable from "init". To achieve this, we comply with Algorithm 1. According to the algorithm, if a path does not satisfy ◊inFs1, then the last state (that has the self-transition) of the path is considered a "cx" state. Otherwise, the state does not need to be taken into account.
To identify "cx" states in layer 1, we can begin by reducing the command.

```
modelCheck(init, <> inFs1)
```

The Maude LTL model checker finds the following as a cx state:

```
{(q : (p1|p2))(#ps : 2)(depth : 2)(loc[p1] : ws)(loc[p2] : ws)}
```

To ensure that Qlock ignores the first cx state, we introduce the following equation:

```
eq {(q: (p1 | p2)) (#ps: 2) (depth: 2)
(loc[p1]: ws) (loc[p2]: ws)} |= inFs1 = true .
```

We perform model checking experiments until no more cx states are discovered. After conducting the model checking experiments, it is determined that four cx states, were found at depth 2.

```
{(q : p1)(#ps : 2)(depth : 2)(loc[p1] : cs)(loc[p2] : ss)}
{(q : (p2|p1))(#ps : 2)(depth : 2)(loc[p1] : ws)(loc[p2] : ws)}
{(q : p2)(#ps : 2)(depth : 2)(loc[p1] : ss)(loc[p2] : cs)}
```

These four cx states become the initial states for layer 2, represented as init3, init4, init5, and init6, respectively. The bound for layer 2 is set to 4. Similar to layer 1, we conduct model checking experiments for layer 2 by checking ◊inFs1 for each state. To perform the model checking experiments, the following commands are used:

```
modelCheck(init3, <> inFs1)
modelCheck(init4, <> inFs1)
modelCheck(init5, <> inFs1)
modelCheck(init6, <> inFs1)
```

The first two commands in the model checking experiments for layer 2 do not find any cx state. However, the last two commands, each find one cx state. There are four states placed at depth 4, reachable from each of the four initial states (init3, init4, init5, init6). These states are denoted as init8, init8', init12, and init12', respectively. However, it is specified that init8 and init12 are the same as init8' and init12', respectively. Therefore, only two different states are placed at depth 4.

```
{(q: p2) (#ps: 1) (depth: 4)(loc[p1]: fs) (loc[p2]: ws)}
{(q : p1)(#ps : 1)(depth : 4)(loc[p1] : ws)(loc[p2] : fs)}
```

Therefore, there is only one cx state placed at depth 4. As init12 and init12' are the same, the cx state is represented by init12, where the depth information is removed. We use this as an initial state for the final layer. After model checking this experiment, there is no cx state. Therefore, we can conclude that *Qlock* satisfies the eventual property $\Diamond$inFs1 when two processes are used. With this *Qlock* specification, our tool which will be described in the next chapter automates the process of conducting model checking experiments. It utilizes the specified eventual property and layer configuration to analyze the Qlock system and verify its intended eventual property.

4.2 A Divide & Conquer Approach to Eventual Model Checking

We will first describe a two-layer divide-and-conquer approach to eventual model checking. The basic idea is to split the original model checking problem for a Kripke structure K and a path π from the Kripke structure K into two parts: π_k and π^k which can be visualized as Figure 4.6.

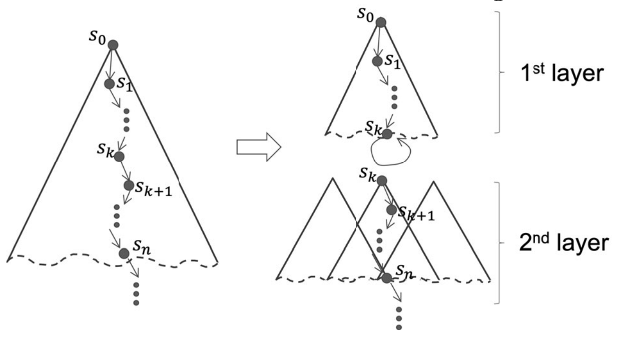

Figure 4.6: Two-layer division of the reachable state space

When there is one initial state, the first layer has one sub state space and the second layer has N sub-stae spaces, where N is the number of states located at a depth k. If each sub-state space has a much fewer number of different states than the original state space, then the model checking experiment for each sub-state space would be feasible even though the model checking experiment for the original state space is not. We anticipate that if $(K, \pi_k \models \Diamond\varphi)$ holds, then $(K, \pi \models \Diamond\varphi)$ holds and if $(K, \pi_k \not\models \Diamond\varphi)$ does not hold, then $(K, \pi^k \models \Diamond\varphi)$ should hold to make $(K, \pi \models \Diamond\varphi)$ hold. Therefore, we will prove this.

Lemma 4.1. *(Two-layer division of $\Diamond$). Let φ be any state proposition of* K. *For any natural number* k*, $(K, \pi \models \Diamond\varphi) \iff ((K, \pi_k \models \Diamond\varphi) \vee ((K, \pi_k \not\models \Diamond\varphi) \Rightarrow (K, \pi^k \models \Diamond\varphi)))$. (We could use $(K, \pi_k \models \Diamond\varphi) \vee (K, \pi^k \models \Diamond\varphi)$ instead of $(K, \pi_k \models \Diamond\varphi) \vee ((K, \pi_k \not\models \Diamond\varphi) \Rightarrow (K, \pi^k \models \Diamond\varphi))$ because they are equivalent).*

Proof. (1) Case "only if" ($\Rightarrow$): There must be i such that $K, \pi^i \models \varphi$. If $i \leq k$, $K, \pi_k^i \models \varphi$ because φ is a state proposition. Thus, $K, \pi_k \models \Diamond\varphi$. Otherwise, $K, \pi_k \not\models \Diamond\varphi$. However, $i{>}k$ and $k, \pi^i \models \varphi$. Hence, $K, \pi^k \models \Diamond\varphi$. (2)Case "if($\Leftarrow$): If $K, \pi_k \models \Diamond\varphi$, there must be i such that $i{<}k$ and $K, \pi_k^i \models \varphi$. As φ is a state proposition, $K, \pi^i \models \varphi$ and then $K, \pi \models \Diamond\varphi$. If $K, \pi_k \not\models \Diamond\varphi$, then there must be j such that $j{>}k$ and $K, \pi^j \models \varphi$. Thus, $K, \pi \models \Diamond\varphi$.

lemma 4.1 makes it possible to divide the original model checking problem $K, \pi \models \Diamond\varphi$ into two layers model checking problems, $K, \pi_k \models \Diamond\varphi$ and $K, \pi^k \models \Diamond\varphi$. We only need to tackle $K, \pi^k \models \Diamond\varphi$ unless $K, \pi_k \models \Diamond\varphi$ holds.

Definition 4.1. *(Eventually$_L$). 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 ∞. $d_i = d(0) + \ldots + d(i)$.$d_i$ is the depth of states located at the bottom at layer i for $i = 0, 1, \ldots, L$; $d_{(L+1)} = \infty$. K_i is the Kripke structure obtained from K by deleting all transitions from each state at the depth $d_{(i+1)}$ and adding a self-transition to each state at depth d_i for $i = 1, \ldots, L$.*

1. $0 \leq k{<}L-1$
Eventually$_L(K_{k+1}, \pi, \varphi, k)$
$\triangleq (K_{k+1}, \pi^{(d_k, d_{(k+1)})} \models \Diamond\varphi) \vee [(K_{k+1}, \pi^{(d_k, d_{(k+1)})} \not\models \Diamond\varphi)$
$\Rightarrow$ Eventually$_L(K_{k+1}, \pi, \varphi, k+1)]$.
2. $k = L-1$
Eventually$_L(K, \pi, \varphi, k)$
$\triangleq (K_{k+1}, \pi^{(d_k, d_{(k+1)})} \models \Diamond\varphi) \vee [(K_{k+1}, \pi^{(d_k, d_{(k+1)})} \not\models \Diamond\varphi)$
$\Rightarrow (K, \pi^{(d_{(k+1)}, d_{(k+2)})} \models \Diamond\varphi)]$.

We are going to focus on the $L+1$ layer division of the eventual model checking problem. We will prove the equivalence of the original eventual model checking problem to the multiple smaller eventual model checking problems.

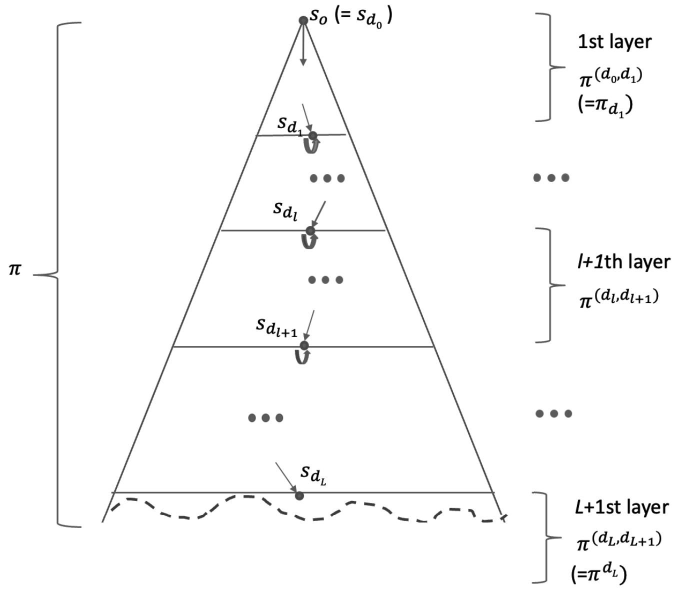

Figure 4.7: An L+1-layer Divide & Conquer Approach to Eventual Model Checking

Theorem 4.1. *(L+1 layer division of $\Diamond$). 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 be any state proposition of K. Then,*

$$K, \pi \models \Diamond\varphi \iff \text{Eventually}_L(K, \pi, \varphi, 0)$$

Proof. By induction on L.

- Base case ($L = 1$): It follows from Lemma1.
- Induction case($L = l + 1$)):
  We prove the following:

$$(K, \pi \models \Diamond\varphi) \iff \text{Eventually}_{l+1}(K_{l+1}, \pi, \varphi, 0)$$

Let d_{l+1} be d used in $\text{Eventually}_{l+1}(K_{l+1}, \pi, \varphi, 0)$ 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} and $d_{l+1}(l+2) = \infty$. The induction hypothesis is as follows:

$(K, \pi \models \Diamond\varphi) \iff$ Eventually$_l(K_l, \pi, \varphi, 0)$
Let d_l be d used in Eventually$_l(K_l, \pi, \varphi, 0)$ such that $d_l(0) = 0$, $d_l(i)$ is an arbitrary natural number for $i = 1, \ldots, l$ and $d_{l+1} = \infty$. As $d_{l+1}(i)$ 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 $i = 1, \ldots, l$. As π is any path of K, π can be replaced with $\pi^{d_l(1)}$.
If so, we have the following as an instance of the induction hypothesis:
$(K, \pi^{d_l(1)} \models \Diamond\varphi) \iff$ Eventually$_l(K_l, \pi^{d_l(1)}, \varphi, 0)$
From Definition 4.1, Eventually$_l(K_l, \pi^{d_l(1)}, \varphi, 0)$ is
Eventually$_{l+1}(K_{l+1}, \pi, \varphi, 1)$ 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:
$(K, \pi^{d_{l+1}(1)} \models \Diamond\varphi \iff$ Eventually$_{l+1}(K_{l+1}, \pi, \varphi, 1)$
From Definition 4.1, Eventually$_{l+1}(K_{l+1}, \pi, \varphi, 0)$ is
$(K, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \Diamond\varphi) \vee [(K, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \Diamond\varphi)] \iff$
Eventually$_{l+1}(K_{l+1}, \pi, \varphi, 1)$ which is
$(K, \pi^{(d_{l+1}(0), d_{l+1}(1))} \models \Diamond\varphi) \vee [(K, \pi^{(d_{l+1}(0), d_{l+1}(1))} \not\models \Diamond\varphi)] \Rightarrow (K, \pi^{d_{l+1}(1)} \models \Diamond\varphi)$
because of the induction hypothesis instance.
From Lemma 4.1, this is equivalent to $K, \pi \models \Diamond\varphi$.

Theorem 4.1 makes it possible to divide the original model checking problem $K, \pi \models \Diamond\varphi$ into $L + 1$ model checking problems: $K, \pi^{(d(0),d(1))} \models \Diamond\varphi$, $K, \pi^{(d(1),d(2))} \models \Diamond\varphi, \ldots$, $K, \pi^{(d(i-1),d(i))} \models \Diamond\varphi$, $K, \pi^{(d(i),d(i+1))} \models \Diamond\varphi, \ldots$, $K, \pi^{(d(L),d(L+1))} \models \Diamond\varphi$. We only need to tackle $K, \pi^{(d(i),d(i+1))} \models \Diamond\varphi$ if all of $K, \pi^{(d(0),d(1))} \models \Diamond\varphi$, ..., $K, \pi^{(d(i-1),d(i))} \models \Diamond\varphi$ do not hold. We prove that an eventual model checking problem for a Kripke structure K and a path π of K is equivalent to $L + 1$ eventual model problems for K and $L + 1$ paths of K, where $L \geq 1$ and the $L+1$ paths are obtained by splitting π into $L+1$ parts. The $L + 1$ parts are $\pi^{(d(0),d(1))}(= \pi_{d(0)}), \ldots, \pi^{(d(l),d(l+1))}, \ldots, \pi^{(d(L),d(L+1))}(= \pi^{d(L)})$.

4.3 Algorithm of DCA2EMC

This section describes an algorithm that carries out the proposed technique. The algorithm takes as inputs a Kripke structure K, a state proposition φ, a non-zero natural number L, and a function d such that $d(x)$ is a natural number for $x = 1, ..., L$, where $d(x)$ is the depth of layer x; and returns as an output success if $K \models \Diamond\varphi$ holds and failure otherwise.

Algorithm 1: Model checking eventual properties in a stratified way.

```
input : $\boldsymbol{K}$—a Kripke structure
        $\varphi$—a state proposition
        $L$—a non-zero natural number
        $d$—a function such that $d(x)$ is a natural number for
        $x = 1, \ldots, L$, where $d(x)$ is the depth of layer $x$
output: Success ($\boldsymbol{K} \models \Diamond\varphi$) or Failure ($\boldsymbol{K} \not\models \Diamond\varphi$)
1  $\boldsymbol{ES} \leftarrow \boldsymbol{I}$
2  forall $l \in \{1, \ldots, L+1\}$ do
3      if $ES = \emptyset$ then
4          return Success
5      $\boldsymbol{ES}' \leftarrow \emptyset$
6      forall $s \in \boldsymbol{ES}$ do
7          forall $\pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}$ do
8              if $\boldsymbol{K}, \pi \not\models \Diamond\varphi$ then
9                  $\boldsymbol{ES}' \leftarrow \boldsymbol{ES}' \cup \{\pi(d(l))\}$
10     $\boldsymbol{ES} \leftarrow \boldsymbol{ES}'$
11 if $ES = \emptyset$ then
12     return Success
13 else
14     return Failure
```

An algorithm can be constructed based on Theorem 4.1, which is shown as Algorithm 1. For each initial state $s_0 \in K$, unfolding s_0 by using T such that each node except for s_0 has exactly one incoming edge, an infinite tree whose root is s_0 is made. The infinite tree may have multiple copies of some states. Such an infinite tree can be divided into $L + 1$ layers, as shown in Figure 3, where L is a non-zero natural number. Although there does not actually exist layer 0, it is convenient to just suppose that we have layer 0. Therefore, let us suppose that there is virtually layer 0 and so is located at the bottom of layer 0. Let n_l be the number of states located at the bottom of layer $l = 0, 1, ..., L$, and then there are n_l sub-state spaces in layer $l+1$. In this way, the reachable state space from s_0 is divided into multiple smaller sub-state spaces. As R is finite, the number of different states in each layer and in each sub-state space is finite. Theorem 1 makes it possible to check $K \models \Diamond\varphi$ in a stratified way in that for each layer $l \in \{1, ..., L+1\}$ we can check $K, s, d(l) \models \Diamond\varphi$ for each $s \in \{\pi(d(l-1)) | \pi \in P^{d(l-1)}_{(K,s_0)}\}$, where $d(0)$ is 0, $d(x)$ is a non-zero natural number for $x = 1, ..., L$, and $d(L+1)$ is ∞.

ES and ES' are variables to which sets of states are set. Each iteration of the outermost loop in Algorithm 1, which conducts the model checking experiment in layer $l = 1, ..., L + 1$, ES is the set of states located at the bottom of layer $l = 0, 1, ...L$, and ES' is the empty set before the model checking experiments conducted in the $l + 1$st iteration. If $K, \pi \not\models \Diamond\varphi$ for $\pi \in P^{d(l)}_{(K,s)}$, then $\pi(d(l))$ is added to ES'. ES' is set to ES at the end of each iteration. If $(K, s) \in ES$ is empty at the beginning of an iteration, Success is returned, meaning that $K \models \Diamond\varphi$ holds. After the outermost loop, we check whether ES is empty. If so, Success is returned; otherwise, Failure is returned.

Although Algorithm 1 does not construct a counterexample when failure is returned, it could be constructed. For each $l \in \{0, 1, ..., L\}$, ES_l is prepared. As elements of ES_l, pairs (s, s') are used, where s is a state in S or a dummy state denoted δ-stt that is different from any state in S, s' is a state in S, and s' is reachable from s if $s \in S$. The assignment at line 5 should be revised as follows:
$ES_l \leftarrow \emptyset$.
The assignment at line 9 should be revised as follows:
$ES_l \leftarrow ES_l \cup \{(s, \pi(d(l)))\}$.
The assignment at line 10 should be revised as follows:
$ES \leftarrow \{s | (s, s') \in ES_l\}$.
ES_0 is set to $\{(\delta\text{-stt}, s) | s \in I\}$. We could then construct a counterexample when failure is returned by searching through $ES_L, ..., ES_1$, and ES_0.
In the next chapter, we will describe the implementation of our support tool which can automate the whole model checking process following this algorithm and can construct the counterexample if any.

4.4 Summary

In this section, we have described the theoretical concept of the technique $L+1$ divide and conquer approach to eventual model checking in detail with the proof. The proof have showed that the original eventual original checking problem is equivalent to the multiple smaller model checking problems. We have constructed the algorithm and explained every step of the algorithm. We used the *QLOCK* protocol in which two processes are used to show how our approach works in detail. We used three layers for this protocol and the depth for the first layer and second layer was defined as 2. The final layer does not need to define the bound value for our approach/tool. We left to compare Maude LTL model checker and DCA2EMC in terms of memory usage and performance in the next chapter.

Chapter 5

A Support Tool for the Proposed Technique

This chapter presents a detailed step-by-step application of our support tool and its implementation within the Maude environment. The effectiveness of our support tool is demonstrated through several case studies, comparing its performance with that of the Maude LTL model checker.

5.1 How to Use the Support Tool

The process for employing our support tool in the system can be delineated through the following steps:

1.
   ```
   Maude> in specs/qlock
   ...
   ```
   This command signifies the initial loading of the *Qlock* specification into the Maude environment.

2.
   ```
   Maude> in full-maude
   ...
   Full Maude 3.1 Oct 12 2020
   ```
   This command signifies the loading of the full Maude.

3.
   ```
   Maude> in solver
   ...
   L+1 Layers Divide & Conquer Approach to Eventual
   Model Checking Available Now
   Done reading in file:  "solver-loop.maude"
   ```
   This command signifies the loading of our support tool solver.

4.
   ```
   Maude> (initialize[QLOCK-CHECK, init, e-prop, OComp,
   Soup{OComp}]).
   Initializer:  success
   origin-module:  QLOCK-CHECK
   revise-module:  QLOCK-CHECK-REVISED
   initial-state:  init
   formula:  e-prop
   ele-sort:  OComp
   soup-sort:  Soup‘OComp‘
   ```
   This command initializes the *Qlock* specification along with the initial state, eventual property, observable component, and soup of observable components. Notably, during the initialization process, the original *Qlock* specification undergoes revision.

5.
   ```
   Maude> (check 2 2)
   Analyzer:
   current-depth:  4
   depth-list:  2 2
   #node-set:  1
   #cx-states:  1
   Checker:  success
   ```
   This command performs an experiment of eventual model checking, employing a total of three layers. Specifically, each of the first two layers has a depth of two. The support tool returns success for *Qlock* with two processes. It concludes that *Qlock* with two processes satisfies the eventual property concerned.

5.2 Tool Implementation in Maude

In our support tool, we utilize Full Maude as the underlying framework, which incorporates a database called DB of modules that is managed by Full Maude. Additionally, we employ an extended database known as DB-EX to store and manage our data. DB-EX plays a crucial role in storing various components, including formalized states (configurations), collected data at each layer, initial and revised specifications, the initial state of the system, the desired eventual property, a component called OComp, a specific collection named SoupOComp, and essential data structures like the current depth, layer configurations, results obtained during system execution, and a node set denoted as NodeSet. With the aid of DB-EX, our tool gains the capability to effectively handle and process the diverse information required for its operations. 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:

{ `term:` *stt*`,` `trace:` *log*`,` `cx:` *cx* }

stt is a state, the initial state 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` is 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 counterexample *cx* is discovered, then the result saved in `DB-EX` is the following:

{ `term:` s_0`,` `trace:` *log*`,` `cx:` *cx* }

A global `cx` can be constructed 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:

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

Each $node_i$ is in the form as:

`<` *cx−stts* `:` *lst* `>`

cx−stts is a set of `cx` states placed at d and *lst* is a list of pairs of states and natural numbers. NodeSet initially has one element. In the element, *cx−stts* only has the initial state to which `(depth: 0)` is added and *lst* is `nil`. When we generally think about a sub-state space (as depicted in Fig. 5.1) 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

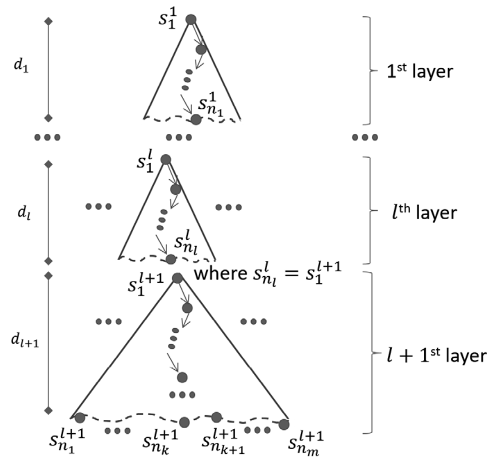

Figure 5.1: Some information managed by the tool

$s_{n_{k+1}}^{l+1}, \ldots, s_{n_m}^{l+1}$ are `cx` states, *node* is as:

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

`_|_` 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 an eventual 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,N,Cs)
= $layer-check(M,NS,N,empty,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), `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) apart from 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,N,NS,CACHE,Cs) = NS .
ceq $layer-check(M,(ND, NS1),N,NS2,CACHE,Cs)
= $layer-check(M,NS1,N, union(NS2,ns(NS&CACHE)),cache(NS&CACHE),Cs)
```

```
  if NS&CACHE := layer-check-for-eventual(M,ND,N,empty,CACHE,Cs) .
```

The first three and sixth parameters are the same as the four ones of `layer-check`, respectively. The fourth parameter manages the result (the node set) and the fifth parameter manages a cache that is used to avoid duplicate states when constructing nodes in the result at each layer. `$layerCheck` returns a node set.
We define `layer-check-for-eventual` as:

```
eq layer-check-for-eventual(M,< empty-term-set : LL >,N,NS,CACHE,Cs)
= < NS : CACHE > .

ceq layer-check-for-eventual(M, < T | TS2 : LL >,N,NS,CACHE,Cs)
= layer-check-for-eventual(M,< TS2 : LL >,N,
union(NS,filter-valid-node(< (CxSTATES except CACHE) :
(< T : N > LL) >)),(CxSTATES | CACHE),Cs)
if CxSTATES := gen-cx-states(M,T,eventual-formula(Cs),eventual-prop
(Cs)) .
```

If all `cx` states 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 $cx{-}stts$ in $<$ $cx{-}stts$: lst $>$ utilized as the second parameter of `layer-check-for-eventual`. Function `gen-cx-states` gathers all `cx` states (e.g. $s^{l+1}_{n_{k+1}}, \ldots, s^{l+1}_{n_m}$ as depicted in Fig. 5.1) reachable from state `T` (e.g. s^{l+1}_1) in the formal specification `M`.
We define `gen-cx-states` as:

```
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]) .
```

We use 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 `cx` state; otherwise, we obtain the last state in the loop of the counterexample `Cx` as a `cx` state by calling function `get-cx-state`, construct an equation to add 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 gather the next one. One of the equations added to the formal specification `M` is as follows:

```
eq {(q: (p1 | p2)) (#ps: 2) (depth: 2) (loc[p1]: ws)
   (loc[p2]: ws)} |= inFs1 = true .
```

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

```
rl [flaw] : {(#ps: 1) (q: (I | Q)) (loc[I]: cs) OCs}
         => {(#ps: 0) (q: (I | Q)) (loc[I]: cs) OCs}.
```

The following is shown:

```
Checker: failure
Cx: counterexample({{#ps: 2 q: empq(loc[p1]: ss)loc[p2]: ss},'start}
    {{#ps: 2 q: p2(loc[p1]: ss)loc[p2]: ws},'wait}
    {{#ps: 2 q: p2(loc[p1]: ss)loc[p2]: cs},'start}
    {{#ps: 2 q:(p2 | p1)(loc[p1]:ws)loc[p2]: cs},'exit}
    {{#ps: 1 q: p1 (loc[p1]: ws)loc[p2]: fs},'wait}
    {{#ps: 1 q: p1(loc[p1]: cs)loc[p2]: fs},'flaw},
    {{#ps: 0 q: p1(loc[p1]: cs)loc[p2]: fs},'fin})
```

Once we get to the following state:

```
{(#ps: 0) (q: p1) (loc[p1]: cs) (loc[p2]: fs)}
```

We will stay there forever because a self-transition can be taken infinitely many times. The process `p1` placed at cs cannot, therefore, go to fs.

5.3 Case Studies and Experimental Results

We conducted some experiments with two kinds of memory usage. The first one is that we used a limited amount of memory by utilizing a Docker container, which ran on a virtual machine operating Ubuntu 20.04.3 LTS. The virtual machine was allocated 2 GB of memory and hosted on an iMac with a 4 GHz processor and 32 GB of memory. The second one is that we employed a MacPro that carries a 2.5 GHz microprocessor with 28 cores and 768 GB memory of RAM. Four mutual exclusion protocols were chosen as examples: (1) Qlock, (2) Anderson, (3) MCS, and (4) TAS. Anderson is a mutual exclusion protocol that utilizes an atomic array shared by all processes [20]. MCS, invented by Mellor-Crummey & Scott, is a list-based queueing mutual exclusion protocol, with variants commonly employed in Java virtual machines [21]. TAS is a simple mutual exclusion protocol that utilizes the atomic operator test&set. For all case studies, we assume that processes enter the critical section at most once. The property $\Diamond$inFs1 was checked consistently across all case studies.

5.3.1 Case Studies

Anderson

The pseudo-code of Anderson is the following:

```
        “start-section”
ss : $plc[i] :=$ fetch−inc−mod$(nxt, N)$;
ws : **while until** $arr[plc[i]]$;
        “critical-section”
cs : $arr[plc[i]], arr[(plc[i] + 1)\%N]$
       $:=$ false, true;
        “final-section”
fs : ...
```

The number of processes taking part in Anderson is denoted N. nxt and arr are global variables whose types are natural numbers and Boolean arrays of size N. Each process has its local variable $plc[i]$ whose type is natural numbers. fetch−inc−mod is an atomic operation. nxt is incremented modulo N and its old value is returned by fetch−inc−mod indivisibly. $x1, x2 := e1, e2$; is a concurrent assignment, where $x1, x2$ are variables and $e1, e2$ are expressions. In each initial state, each process i is at ss, nxt is 0, $arr[0]$ is true, each $arr[j]$ is false for $j = 1, ..., N - 1$, and each $plc[i]$ is 0. Whenever i would like to go to cs, it assigns $plc[i]$ nxt and increments nxt modulo N with fetch−inc−mod in an atomic way. It then goes to ws from ss. It waits at ws while $arr[plc[i]]$ is false. Whenever $arr[plc[i]]$ becomes true, i goes to cs. Whenever it would like to exit cs, it assigns $arr[plc[i]]$ false and $arr[(plc[i] + 1)\%N]$ true, going to fs from cs. We assume that processes go to the critical section at most once.

TAS

TAS for each process p can be described as:

```
        “start-section”
ss : ...
ws : **repeat while** test&set($locked$);
        “critical-section”
cs : $locked :=$ false;
        “final-section”
fs : ...
```

TAS uses an atomic operator test&set and a global variable *locked* whose type is Boolean. *locked* is initially assigned false and shared by multiple processes participating in TAS. *locked* is assigned true and its old value is returned by test&set indivisibly. Each process is initially at ss (start-section). Whenever it would like to go to cs (critical-section), it first goes to ws (waiting-section) from ss. It waits at ws while *locked* is true; precisely while test&set(*locked*) returns true. When test&set(*locked*) returns false, the process goes to cs. Whenever it would like to exit cs, it moves to fs (final-section) from cs. When the process goes to cs from ws, it assigns *locked* true with test&set. When it goes to fs from cs, it assigns *locked* false. We assume that processes go to the critical section at most once.

MCS

MCS [22] for each process p can be described as:

```
          "start-section"
ss  : ...
l1  : $nxt[p] := nop$;
l2  : $prd[p] :=$ fetch−store($glock, p$);
l3  : if $prd[p] \neq nop$ {
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 (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.

5.3.2 Experimental Results

We performed model checking experiments using the formula $\Diamond inFs1$ on various protocols, namely Qlock, Anderson, MCS, and TAS, with different numbers of processes. The experiments were conducted using both our support tool, DCA2EMC, and Maude LTL model checker. The results are summarized in Table 5.1 and Table 5.2.
In the table, the `No of Processes` column denotes the number of processes involved in each protocol. The `Layers` column specifies the layer configuration, where $d_1d_2...d_L$ indicates that $L + 1$ layers were used, with the ith layer having a depth of d_i. Notably, the Maude LTL model checker does not require to use the layers.

A Limited Amount of Memory

For example, the first record of the table 5.1 shows that when there are 8 processes taking part in Qlock, the model checking experiment took 1 minute and 6 seconds if we use our tool with three layers in which the depth for the first and second layers is 2. It took 15 seconds when we use the Maude LTL Model Checker for which we do not require to use the layers. In some cases, denoted by 'N/A' (Not-Available) in the table, the model checking experiments could not be completed due to the state space explosion problem within the limited amount of memory we employed. However, for Qlock, Anderson, MCS, and TAS with 8, 8, 4, and 11 processes, respectively, both the Maude LTL model checker and our tool successfully finished the experiments within a reasonable time frame.

For Qlock, Anderson, MCS and TAS with 8, 8, 4, and 11 processes, the Maude LTL model checker achieved the results in a short time, and our tool also demonstrated efficient performance, except for the TAS case study.

For Qlock, Anderson, MCS, and TAS with 9, 9, 5, and 12 processes, respectively, the Maude LTL model checker was not able to conduct the model checking experiment due to state space explosion within the 2 GB memory constraint. Conversely, our tool successfully completed the model checking

Table 5.1: Model checking results with the limited amount of memory

Protocol	No of Processes	Layers	DCA2EMC	Maude LTL Model Checker
Qlock	8	2 2	1m 6s	15s
Qlock	9	2 2	4m 52s	N/A
Anderson	8	2 2	2m 9s	2m 52s
Anderson	9	2 2	20m 25s	N/A
MCS	4	2 2 2 2 2 2	36s	1s
MCS	5	2 2 2 2 2 2	21m 39s	N/A
TAS	11	3 3	56m 59s	30s
TAS	12	3 3	7h 0m 40s	N/A

Table 5.2: Model checking results with the large mount of memory

Protocol	No of Processes	Layers	DCA2EMC	Maude LTL Model Checker
Qlock	10	2 2	3h 12m 12s	10d 11h 31m 8s
Anderson	9	2 2	27m 19s	1d 3h 18m 43s
MCS	6	4 4 4 4 2	1d 15h 32m 21s	5d 20h 52m 47s
TAS	13	3 3 3	2d 4h 19m	2h 28m 29s

experiments. Thus, our approach/tool can operate optimally within memory constraints, showing that our approach/tool can mitigate the state space explosion to some extent.

A Large Amount of Memory

In these experiments, we used two optimization techniques proposed in [23] with the sequential in order to find a good layer configuration and enhance the running performance on generating counterexample states up to the final layer for each case study. In the case of utilizing a large amount of memory, the Maude LTL model checker could finish the model checking experiments for Qlock, Anderson, MCS, and TAS with 10, 9, 6, and 13 processes. However, the time taken for these experiments was considerably longer when compared to our support tool except for the TAS case study. Our approach/tool splits the reachable state spaces from the initial state into multiple layers, generating smaller sub-state spaces. Tackling the smaller sub-state space has some advantages. Firstly, we do not need to manage a large amount of memory. Additionally, due to dealing with the smaller size of state spaces, the hash tables are also smaller and so state matching is less burdensome. It is also good for the hardware cache. This is why our approach/tool has a better running performance than the Maude LTL model checker.

For the TAS case study with 11, 12, and 13 processes, there may have many states that are shared between many sub-state spaces at the final layer of TAS because each process that is waiting at ws has an equivalent chance to enter cs which means that TAS has many symmetries from the perspective of individual processes. This makes the running performance of our tool degrade for both cases of memory usage.

In conclusion, the results highlight the effectiveness of our developed tool, DCA2EMC, in overcoming the state space explosion problem and achieving efficient model checking in comparison to the Maude LTL model checker for various protocols under different process configurations. Thus, our tool alleviated the state space explosion problem to a certain extent.

5.4 Summary

In this chapter, we have described how we implemented the support tool for the divide & conquer approach to eventual model checking in Maude using the facilities of metaprogramming. Our tool is easy to use in which the system specification, desired property, and layer configuration are required to provide by users. The main weakness of our tool is that our tool cannot handle the model checking experiments if the specifications have the long lasso backward transition. This can make the sub-spaces in the final layer as large as the original reachable state space. Therefore, we are required to create system formal specifications in which the multiple smaller sub-state spaces from the original reachable state must have a much smaller number of states than the number of states in the original reachable state space in order to make the best use of our approach/tool. The four mutual exclusion protocols that we used in the case studies, do not have the long lasso loops in the specification.

Through case studies, we compared the performance of the Maude LTL model checker with our support tool. We found that when there are only a few processes in the protocol, the Maude model checker can get the result faster than our support tool in the utilization of limited memory. However, as the number of processes increases, the Maude model checker could not finish the experiments due to state space explosion and needed a large memory to finish the model checking experiments. In contrast, our support tool managed to get the result even in such cases as our approach/tool divides the original reachable state spaces into multiple smaller sub-state spaces, defines layer configuration and tackles the model checking experiments layer by layer sequentially. This makes efficient use of memory. In conclusion, our support tool is a promising solution, especially for scenarios with more processes, where the Maude LTL model checker falls short due to state space explosion.

Chapter 6

A Case Study: The Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol

In this chapter, we will discuss the formal specification and model checking process of the Lim-Jeong-Park-Lee autonomous vehicle intersection control (LJPL) protocol [11] using both Maude LTL model checker and our support tool. We aim to conduct experiments to demonstrate the effectiveness of our support tool, especially in handling good instances of the autonomous vehicle intersection protocol that is challenging for Maude LTL model checker.

6.1 LJPL Protocol

In the context of the depicted intersection shown in Figure 6.1, the road network consists of two streets intersecting with each other. Vehicles are assumed to follow a right-hand traffic system, where each street is divided into two lanes. The lanes are labeled accordingly, as illustrated in 6.1.

During the crossing of the intersection, vehicles traveling on the right lane of one street, such as lane0, are expected to proceed straight or make a right turn. Conversely, vehicles in the left lane of one street, such as lane1, are intended to make left turns. The area where the two streets overlap forms a critical section, as highlighted in Figure 6.1. Within this critical section, it is crucial to implement effective control mechanisms to ensure that vehicles never collide with each other. The intersection has eight lanes labeled as lane0 to lane7. The following relationships exist between the lanes:

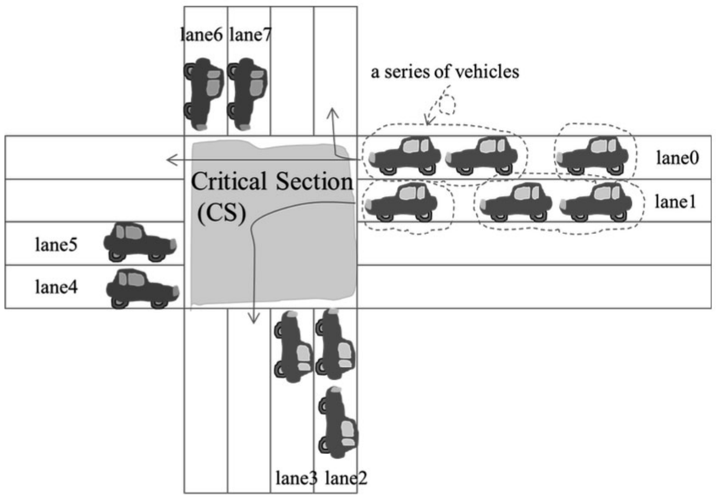

Figure 6.1: An intersection

- For $i = 0, 2, 4, 6$:
  The conflict lanes of lane$i$ are lane$j$, where $j = (i + 2) \bmod 8$, $(i + 5) \bmod 8$, $(i + 6) \bmod 8$, and $(i + 7) \bmod 8$. The concurrent lanes of lane$i$ are lane$j$, where $j = (i + 1) \bmod 8$, $(i + 3) \bmod 8$, and $(i + 4) \bmod 8$.

- For $i = 1, 3, 5, 7$:
  The conflict lanes of lane$i$ are lane$j$, where $j = (i + 1) \bmod 8$, $(i + 2) \bmod 8$, $(i + 3) \bmod 8$, and $(i + 6) \bmod 8$. The concurrent lanes of lane$i$ are lane$j$, where $j = (i + 4) \bmod 8$, $(i + 5) \bmod 8$, and $(i + 7) \bmod 8$.

Each vehicle is assigned a status, such as `running`, `approaching`, `stopped`, `crossing`, or `crossed`. When a vehicle is far enough from the intersection, its status is `running`. As vehicles approach the intersection, their statuses change to `approaching`. Their IDs are added to the specific lane's queue, and they maintain their lane and position in the queue without overtaking other vehicles.

When a vehicle is enqueued and there are no stopped vehicles ahead of it (including two cases: being at the front of the queue with no vehicles ahead or being followed by a crossing vehicle), it becomes the lead vehicle and its status changes to `stopped`. Otherwise, it becomes a non-lead vehicle and its status also changes to `stopped`.

A lead vehicle checks if there are no vehicles crossing the intersection on any conflict lanes and if the time assigned to the lead vehicle is earlier than the times assigned to the lead vehicles on the conflict lanes. If both conditions are met, the lead vehicle and all the stopped non-lead vehicles following it are allowed to cross the intersection, and their statuses change to crossing.

The LJPL protocol considers a group of vehicles as a train, allowing them to pass through the intersection simultaneously. For instance, let's consider the vehicles on lane0 in Figure 6.1. Suppose the first vehicle is the lead vehicle, the second vehicle is non-lead, and the third vehicle is also a lead vehicle, with no more vehicles following. When the first vehicle is permitted to cross the intersection, the second vehicle is also allowed, treating the series of the first and second vehicles as a train. The algorithms and the detail of the LJPL protocol are described in the paper [10].

6.2 Formal Specification of LJPL Protocol

The K_{LJPL} Kripke structure incorporates four types of observable components:

- $(v[vid] : lid, vstat, t, lt)$ – Here, *vid* represents the vehicle ID (a natural number), *lid* denotes the lane ID (a natural number) where the vehicle *vid* is located, *vstat* signifies the status of the vehicle *vid*, t represents the estimated time when the vehicle *vid* will reach the intersection, and *lt* represents the estimated time when the lead vehicle will reach the intersection, if any.

- $(\text{lane}[lid] : q)$ – This component represents a specific lane with ID *lid*. It consists of a queue q that contains vehicle IDs (natural numbers) belonging to that lane.

- $(\text{clock} : t, b)$ – The abstract notion of the current time is denoted by this component. It includes t, a natural number representing the current time, and b, a Boolean value indicating whether the time can increment. Whenever b is true, t is allowed to increase, and b becomes false thereafter. When a vehicle retrieves the current time t, b is set to true (without modifying t).

- $(\text{gstat} : gstat)$ – This component represents the global status of the intersection. gstat can take one of two values: `fin` or `nFin`. If gstat is fin, it indicates that all relevant vehicles have crossed the intersection.

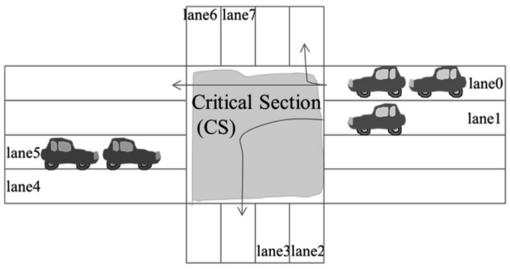

Figure 6.2: An initial state

In the K_{LJPL} Kripke structure, these observable components are utilized to formalize the system's state, encompassing information about the vehicles, lanes, current time, and the global intersection status.

Let's consider an initial state where two vehicles are currently in motion on `lane0`, one vehicle is actively running on `lane1`, and two vehicles are currently in motion on `lane5`. Additionally, there are no vehicles currently in motion on the other lanes. This initial state can be formally expressed as follows:

```
{(gstat: nFin) (clock: 0,false)
 (lane[0]: oo) (lane[1]: oo) (lane[2]: oo)
 (lane[3]: oo) (lane[4]: oo) (lane[5]: oo)
 (lane[6]: oo) (lane[7]: oo)
 (v[oo]: 0,stopped,oo,oo) (v[oo]: 1,stopped,oo,oo)
 (v[oo]: 2,stopped,oo,oo) (v[oo]: 3,stopped,oo,oo)
 (v[oo]: 4,stopped,oo,oo) (v[oo]: 5,stopped,oo,oo)
 (v[oo]: 6,stopped,oo,oo) (v[oo]: 7,stopped,oo,oo)
 (v[0]: 0,running,oo,oo) (v[1]: 0,running,oo,oo)
 (v[2]: 1,running,oo,oo) (v[3]: 5,running,oo,oo)
 (v[4]: 5,running,oo,oo) }
```

In the initial state, each lane's queue, denoted by `(lane[0]:oo)`, `(lane[1]: oo)`, etc., contains only the symbol `oo`, which represents infinity (∞). This indicates that there are no vehicles sufficiently close to the intersection on those lanes. Additionally, there are eight observable components `v[oo]` representing dummy vehicles. Among them, `v[0]`, `v[1]`, `v[2]`, `v[3]`, and `v[4]` correspond to the five actual vehicles, with the first two on `lane0`, the third on `lane1`, and the last two on `lane5`.

The global clock is denoted by `(clock:0, false)`, with an initial value of 0. The second value of the clock observable component, `false`, prevents the abstract notion of the current time from incrementing. Lastly, the initial value of the `gstat` observable component is `nFin`, indicating that not all vehicles have crossed the intersection.
The 12 rewrite rule can be used to define the state transitions T_{LJPL}.

```
rl [stutter] : {(gstat: fin) OCs} => {(gstat: fin) OCs} .
```

This rule, `stutter`, guarantees that the T_{LJPL} protocol is total, allowing for all possible states to be reached and ensuring that the system remains in a consistent state when all vehicles have already crossed the intersection.

```
crl [fin] : {(gstat: nFin) OCs} => {(gstat: fin) OCs}
            if fin?(OCs) .
```

This rule, `fin`, ensures that the global status accurately reflects the state of the system, transitioning it to `fin` when `fin?(OCs)` returns true which means that all vehicles have successfully crossed the intersection.

```
rl [tick] : {(gstat: nFin) (clock: T,true) OCs}
         => {(gstat: nFin) (clock: (T + 1),false) OCs} .
```

If the global status `(gstat)` is `nFin`, the clock `(clock)` has a value `T` (a natural number) with the second value being true, and the observable components (OCs) remain unchanged, then the rule allows for the transition to a new state where the clock is updated to `(T + 1)` and the second value becomes false. This transition represents the increment of the abstract notion of the current time in the system.
The following two rules are the transition rules that change a vehicle's status from `running` to `approaching` based on the presence or absence of other vehicles close to the intersection on the same lane.

```
rl [approach1] : {(gstat: nFin) (clock: T,B) (lane[LI]: oo)
                 (v[VI]: LI,running,oo,oo) OCs}
              => {(gstat: nFin) (clock: T,true) (lane[LI]: VI)
                 (v[VI]: LI,approaching,T,oo) OCs} .
```

If the global status `gstat` is `nFin`, the clock `(clock)` has a value `T` and a Boolean value `B`, the lane with ID `LI` has no vehicle close enough to the intersection (represented by `oo`), and there exists a vehicle with ID `VI` running on lane `LI`, then the rule allows for the transition to a new state where the clock remains unchanged but the second value becomes true, the `lane[LI]` observable component is updated to contain the ID `VI`, and the `v[VI]` observable component changes its status from `running` to `approaching` with the time of arrival.

```
rl [approach2] : {(gstat: nFin) (clock: T,B) (lane[LI]:(VI';VS))
                  (v[VI]: LI,running,oo,oo) OCS}
               => {(gstat: nFin) (clock: T,true)
                  (lane[LI]: (VI'; VS; VI))
                  (v[VI]: LI, approaching,T,oo) OCs} .
```

B is Maude variable of Boolean values, LI, VI & VI' are Maude variables of natural numbers, VS is Maude variable of queues of natural numbers ∞, and _;_ is the constructor of queues, where an underscore _ is a place holder where an argument is put. _;_ is associative, meaning that $(q_1; q_2); q_3 = q_1; (q_2; q_3)$, and a single element(a natural number or ∞) is treated as the singleton queue that only consists of the element.
If the global status (gstat) is initially nFin, the clock (clock) has a value T and a Boolean value B, the lane with ID LI has at least one vehicle close enough to the intersection represented by the sequence of vehicle IDs (VI'; VS), and there exists a vehicle with ID VI running on lane LI, then the rule allows for the transition to a new state where the clock remains unchanged but the second value becomes true, the lane[LI] observable component is updated to include the ID VI in the sequence (VI'; VS; VI), and the v[VI] observable component changes its status from running to approaching with the time of arrival.

There are three transition rules that change a vehicle's status from approaching to stopped based on different conditions related to its position in the queue and the status of the preceding vehicle on the same lane.

```
rl [check1] : {(gstat: nFin) (lane[LI]: (VI ; VS))
               (v[VI]: LI,approaching,T,oo) OCs}
             => {(gstat: nFin) (lane[LI]: (VI ; VS))
                (v[VI]: LI,stopped,T,T) OCs} .
```

If a vehicle is in the approaching state (approaching) and it is at the top of the queue (the first vehicle), then the rule allows for a transition where the vehicle's status changes from approaching to stopped on the concerned lane. In this case, the vehicle becomes the lead vehicle on the lane.

```
rl [check2] : {(gstat: nFin) (lane[LI]: (VS'; VI'; VI; VS))
               (v[VI']: LI, stopped,T, T')
               (v[VI]: LI, approaching,T'', oo) OCs}
             => {(gstat: nFin) (lane[LI]: (VS'; VI'; VI; VS))
                (v[VI']: LI, stopped, T, T')
                (v[VI]: LI, stopped, T'', T') OCs}.
```

If a vehicle is in the approaching state (approaching) and there exists another vehicle in front of it on the same lane, such that the preceding vehicle's status is `stopped`, then the rule allows for a transition where the vehicle's status changes from `approaching` to `stopped` on the concerned lane. In this case, the vehicle becomes a non-lead vehicle on the lane.

```
rl[check3] : {(gstat: nFin) (lane[LI]: (VS'; VI'; VI; VS))
              (v[VI']: LI, crossing,T, T')
              (v[VI]: LI, approaching, T'', oo) OCs}
           => {(gstat: nFin) (lane[LI]: (VS'; VI'; VI; VS))
               (v[VI']: LI, crossing, T, T')
               (v[VI]: LI, stopped, T'', T'') OCs}.
```

`T'` & `T''` are Maude variables of natural numbers and `VS'` is a Maude variable of queues. The reason why v[*vid*] observable components, where *vid* is a vehicle ID, do not have any values that correspond to lead in $information_x$ is that it is possible to use queues, etc. to manage which vehicles are lead or not.
If a vehicle is in the approaching state (approaching) and there exists another vehicle in front of it on the same lane, such that the preceding vehicle's status is `crossing`, then the rule allows for a transition where the vehicle's status changes from `approaching` to `stopped` on the concerned lane. In this case, the vehicle becomes the lead vehicle on the lane.
Two rules are employed to define transitions that convert the status of a lead vehicle from `stopped` to `crossing`. One rule pertains to the scenario where the lead vehicle is located on a lane with an even ID, while the other rule applies to the case where the lead vehicle is on a lane with an odd ID.

```
crl [enter1] :
{(gstat : nFin)(lane[LI] : (VI; VS))(v[VI] : LI,stopped,T,T)OCs}
=> {(gstat : nFin)(lane[LI] : (VI; VS))
    (v[VI] : LI,crossing,T,T)OCs'}
    if isEven(LI) /\ LI1 := (LI + 2) rem 8 /\ (lane[LI1] :
    (VI1; VS1)) (v[VI1] : LI1,VSt1,T11,T12) OCs1 := OCs /\
    VSt1 = stopped /\ T < T12 /\ LI2 := (LI + 5) rem 8 /\
    (lane[LI2] : (VI2; VS2))(v[VI2] : LI2,VSt2,T21,T22)
    OCs2 := OCs /\ VSt2 = stopped /\ T < T22  /\ LI3 := (LI + 6)
    rem 8 /\ (lane[LI3] : (VI3; VS3)) (v[VI3]: LI3,VSt3,T31,T32)
    OCs3 := OCs /\ VSt3 = stopped /\ T < T32 /\ LI4 := (LI + 7)
    rem 8 /\ (lane[LI4] : (VI4; VS4))(v[VI4] : LI4,VSt4,T41,T42)
    OCs4 := OCs /\ VSt4 = stopped /\  T < T42 /\ OCs' :=
    letCross(VS, OCs) .
```

LIi for $i = 1, ..., 4$ is a Maude variable of natural numbers, VIi & Tj for $i = 1, ..., 4$ & $j = 11, 12, ..., 41, 42$ are Maude variables of natural numbers & ∞, VSi for $i = 1, ..., 4$ is a Maude variable of queues, VSti for $i = 1, ..., 4$ is a Maude variable of vehicle statuses, and OCsi & OCs' for $i = 1, ..., 4$ are Maude variables of observable component soups.

If a lead vehicle is in the stopped state (stopped) on a lane, and the ID of the lane is even, then the rule allows for a transition where the lead vehicle's status changes from `stopped` to `crossing`. This transition occurs when there are no vehicles on any conflict lanes crossing the intersection, and the time given to the lead vehicle is earlier than the times given to the lead vehicles on the conflict lanes. the second rule `enter2` can be described the same.

Two rules are employed to define transitions that change a vehicle's status from `crossing` to `crossed`. The first rule applies when the vehicle is the only one in the queue associated with the corresponding lane. The second rule applies when the queue contains two or more vehicles for the corresponding lane.

```
rl [leave1] : {(gstat: nFin)(lane[LI]: VI)
(v[VI]: LI, crossing,T,T') OCs} => {(gstat: nFin)
(lane[LI]: oo )(v[VI]: LI,crossed,T,T') OCs} .
rl [leave2] : {(gstat: nFin)(lane[LI]: (VI ; VI'; VS))
(v[VI]: LI,crossing,T,T') OCs} => {(gstat: nFin)
(lane[LI]:(VI'; VS))(v[VI]: LI,crossed,T,T') OCs} .
```

When a vehicle's status changes from `crossing` to `crossed`, it is removed from the queue that corresponds to the respective lane. In the T_{LJPL} protocol, this deletion is explicitly handled in the transition rules, specifically in the `leave1` and `leave2` rules mentioned earlier.

6.3 Model Checking of LJPL Protocol

Desired Properties for LJPL

We tried to model check the desired properties which have been shown in the original LJPL protocol paper [11]. The LJPL protocol proposed takes into account three desired properties the protocol should enjoy:

- `Safety (version 2)` No vehicles in conflict lanes are in the critical section (CS) at the same time.

- `Deadlock-freedom` If a vehicle is trying to pass the intersection (CS), then some vehicle, not necessarily the same one, finally passes the intersection.

- `Starvation-freedom` If a vehicle is trying to pass the intersection (CS), then the vehicle must finally pass the intersection in finite time.

The following command can be used to model check `Safety (version 2)` property.

```
search [1] in IMUTEX : init =>* {(v[VI]: LI,crossing,T,T')
(v[VI']: LI',crossing,T2,T2') OCs}
such that areConflict(LI,LI') .
```

This command tries to find a state from the initial state such that two vehicles on the conflict lanes are crossing the intersection at the same time. After conducting model checking experiment with this command, there is no counterexample. It means that the protocol enjoys the safety (version 2) property. The following command can be used to model check `Deadlock-freedom` property.

```
search [1] in IMUTEX : init =>! {OCs} .
```

The search command tries to find a state from the initial state init such that no transition can be conducted. We got the counterexample state after model checking of this experiment. The following state is the counterexample state. In this state, there are two vehicles named `vehicle3` and `vehicle4` are on the `lane5`. The `vehicle3` is the lead vehicle and its status is stopped. There is another vehicle on the `lane0` named `vehicle1` which is a lead vehicle and its state is stopped. The `vehicle3` and `vehicle1` have the arrival time of 0, therefore, both vehicles could not be entered into the critical section.

```
(lane[0]: 0 ; 1) (lane[5]: 3 ; 4)(v[0]: 0,stopped,0,0)
(v[1]: 0,stopped,0,0)(v[3]: 5,stopped,0,0)(v[4]: 5,stopped,0,0)
```

Therefore, the counterexample shows that a global clock is not sufficient to use because it can create a symmetry. The order of lane IDs could be used. When there are multiple lead vehicles on conflict lanes such that their arrival times are exactly the same, higher priorities are given to those on lanes whose IDs are less. That is, a logical clock such that times are total order is used. In the specification, the two rules enter1 and enter2 should be revised such that `T` $<$ `T1`i for $i = 1, ..., 4$ used in the conditions is replaced with (`T` $<$ `T1` i or(`T` $==$ `T1`i and `LI` $<$ `LI` i)). After model checking with the revised version, the protocol enjoys the `Safety (version2)` and `Deadlock freedom` property. Although there are three desired properties to model check the protocol in the paper [10], We will focus on the starvation-freedom property which can be expressed as eventual property. The Starvation-freedom property is a crucial focus of my main research, and it is expressed using the LTL (Linear Temporal Logic) eventually connective $\Diamond$. To verify that the protocol satisfies the Starvation-freedom property, two components, P_{LJPL} and L_{LJPL}, need to be defined. P_{LJPL} consists of the atomic proposition `fin`, representing the completion of the vehicles' passage. On the other hand, L_{LJPL} is defined with two equations:

```
eq {(gstat: fin) OCs} |= fin = true .
eq {OCs} |= PROP = false [owise].
```

These equations specify that for all states $s \in S_{LJPL}$, $L_{LJPL}(s) = \text{fin}$ if and only if s contains the atomic proposition `(gstat: fin)`. By formulating the Starvation-freedom property as

```
eq halt = <> fin .
```

where $\Diamond$ represents the LTL eventually connective, we can proceed to model check the protocol's satisfaction of this property by reducing the following command in Maude.
modelCheck(init, halt).

Experimental Results

We conducted model checking experiments using the formula $\Diamond fin$ on the LJPL protocol involving multiple vehicles with our support tool and Maude LTL model checker.

A Limited Amount of Memory

We used a Docker container, which runs on a virtual machine operating Ubuntu 20.04.3 LTS. The virtual machine was allocated 2 GB of memory and hosted on a Macbook Air with a M1 chip and 16 GB of memory. In the table 6.1, the DCA2EMC represents our support tool. The corresponding time values in the table indicate the time taken required to complete the model checking experiments. The `Layers` column represents the configuration we employed for the model checking experiments. For instance, the entry `2 2` signifies that we utilized three layers, with each of the first two layers having a depth of 2. According to experimental results, for LJPL protocol with 4 and 5 vehicles, the Maude LTL model checker was able to finish the model checking experiments in a short time. When there are four and five vehicles taking part in the protocol, Maude LTL model checker is faster than our tool DCA2EMC. This is because of the overheads introduced by our approach as we divide the original state space into multiple sub-state spaces. We revise the system specification of the protocol on the fly to

Table 6.1: Model Checking Results with the Limited Amount of Memory

No of vehicle	Layers	DCA2EMC	Maude LTL Model Checker
4	2 2	30.69s	5.72s
5	2 2	8m 10.947s	2m 35.98s
6	2 2 2	3h 13m 28s	N/A

Table 6.2: Model Checking Results with the Large Amount of Memory

No of vehicle	Layers	DCA2EMC	Memory Usage	Maude LTL Model Checker	Memory Usage
4	2 2	8s	0.18 GB	1.5s	0.04 GB
5	2 2	1m 53s	0.26GB	1m 57s	0.48 GB
6	2 2 2	1h 59m 39s	0.61 GB	25h 33m 9s	4.61 GB

generate the counterexample states for $\Diamond\varphi$ for each sub-state space in non-final layers where φ is a state proposition. However, when there are six vehicles taking part in the protocol in which no vehicle is on lane1, lane5, and lane7, one vehicle is on each lane of lane0, lane2, lane3, and lane4, and two vehicles are on lane6, the Maude LTL model checker ran out of memory after 33 minutes and 32 seconds and could not complete the mode checking experiments due to the state space explosion while our approach/tool could finish the model checking experiment. In the case study of six vehicles taking part in the protocol, we used the four layers and the bound for each of the first three layers is 2 while the depth for the final layer is unbounded. Our approach divides the original reachable state space from the initial state into multiple layers and uses the Maude LTL model checker to tackle each layer sequentially. Therefore, the computation complexity of the algorithm used by Maude LTL model checker is the same. However, our approach efficiently uses the memory so it can return the result while the Maude LTL model checker could not. This result shows the effectiveness of our approach with the small amount of memory.

A Large Amount of Memory

For a large amount of memory, we used a Macbook Air that carries an M1 chip and 16 GB of memory. We also compare the memory usage of our approach and Maude LTL model checker. When there were four vehicles taking part in the protocol, the memory usage of our approach was a bit larger than the memory usage of the Maude LTL model checker. This is because of another overhead of our approach as we divide the original sub-state into multiple sub-state spaces and tackle each smaller one. Therefore, we need to store some extra information such as states located at the final depth together with the log list to trace back when a counterexample is found. When there are six vehicles taking part in the protocol, Maude LTL model checker can return the result but it is quite obvious that our approach is faster than Maude LTL model checker. The reason is that dealing with the small state spaces is much easier than dealing with the large state spaces because we do not need to manage a large memory and a big size has table and so the state storing, accessing, and matching is much faster. It also makes the best use of the hardware cache.

Table 6.2 provides a comprehensive comparison of the model checking results, showcasing the impact of employing our support tool, DCA2EMC, compared to utilizing the Maude LTL model checker. Based on the experimental results, the overhead introduced by our approach does not affect the model checking running performance very much. It offers insights into the efficiency and effectiveness of our approach in accelerating the model checking process. We can say that our approach/tool can mitigate the state space explosion to a certain scope.

6.4 Summary

In this chapter, we have described how the LJPL protocol is formally specified in Maude. We then model checked the protocol, focusing on the eventual property. We compared the results obtained from our support tool and the Maude LTL model checker. We have shown the experimental results of using our approach/tool and the Maude LTL model checker. We also found a case study that our support can conduct the model checking experiments while the Maude LTL model checker could not. We prepared the LJPL protocol specification in which there are no long lasso backward transitions are not included. This is the most important preparation to make the best use of our support tool. As our approach splits the original reachable state space from the initial state into multiple layers, finding a good layer configuration is one of the important facts in our approach. Therefore, this will be one of our future work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In our research, we have proposed a divide & conquer approach to eventual model checking to address the state space explosion issue in model checking. As indicated by the name, the technique is dedicated to eventual properties. DCA2EMC [1] is an extension of DCA2L2MC [24] and DA2CSMC [25], so as to deal with the termination system requirements that can be expressed as eventual properties. The main idea of our approach is to split the original eventual model checking problem into multiple smaller eventual model checking problems and tackled each smaller one. Since it tackles multiple smaller problems sequentially, we can make efficient use of memory resources. This is the main principle behind our technique. We proved a theorem that demonstrates the equivalence between these smaller model checking problems and the original problem. Based on the theorem, we have designed the algorithm to use for conducting the model checking experiments with the technique. However, manually conducting experiments with DCA2EMC is time-consuming and error-prone when there are many states placed at each layer and we cannot handle it well if the number of layers being checked is considerable. Therefore, it is indispensable to develop a support tool to facilitate conducting experiments with DCA2EMC. This tool, implemented using Maude, leverages its reflective programming capabilities (meta-programming) to automatically revise the system specifications and do some useful computations.

We use the Maude LTL model checker as the model checker in our support tool. When we conducted case studies, we prepare the specifications for each protocol. As our tool can not handle the model checking experiments if the specifications have the long lasso backward transitions in the final layer.

Therefore, we are required to prepare the specification in which the multiple smaller sub-state spaces from the original reachable state must have a much smaller number of states than the state in the original ones. We prepare the eventual properties to model check by using the Maude LTL model checker. Our support tool uses these specifications and eventual properties to conduct model checking experiments. As we use meta-programming facilities in our support tool to update our specifications, the specifications are updated for each sub-state space in each non-final layer on the fly to generate all counterexample states which do not satisfy the eventual property in each layer. This overhead should be considered for the running performance of our support tool.

We conducted case studies using five protocols named `Qlock`, `Anderson`, `MCS`, `TAS`, and `LJPL` by using a limited amount of memory and a large amount of memory to demonstrate the effectiveness of our approach. According to the experimental results, our approach/tool can ease the state space explosion to a certain scope while the Maude LTL model checker is not able to complete the model checking experiments due to the state space explosion. The model checking algorithm adopted by Maude LTL model checker is the same as the one used by SPIN [14], which is one of the most popular model checkers for model checking software systems. It has been reported that Maude LTL model checker is comparable with SPIN with respect to model checking running performance. This implies that whenever Maude LTL model checker encounters the state space explosion problem, making it impossible to conduct model checking experiments, SPIN does so as well, and so do most existing model checkers. Thus, it is meaningful to compare our tool with the Maude LTL model checker.

7.2 Future Work

One area we are currently working on is parallelizing our approach/tool. This involves optimizing the tool to make the best use of parallel computing, where multiple tasks can be performed simultaneously. Since our approach/tool allows for independent model checking experiments on multiple sub-state spaces, parallelization can significantly speed up the process. Our research group has already made progress in parallelizing both the technique itself and the support tool and the detail can be read in this paper [26]. Parallelization has the potential to greatly enhance the performance and scalability of our approach. It allows us to analyze larger and more complex systems in a shorter amount of time by processing multiple sub-state spaces concurrently. By incorporating parallel computing techniques into our support tool, we aim to increase the running performance of our approach/tool.

There are two more directions that we may concern for our approach/-tool. The first one is how to deal with long lasso loops in the specification. The second one is how to find a unified technique to handle not only eventual properties but also other properties such as leads-to properties and conditional state properties which can be expressed freely in LTL. We will come up with good strategies to handle these two concerns in our future work.

References

[1] Aung, M. N., Phyo, Y., Do, C. M. & Ogata, K. A divide and conquer approach to eventual model checking. *Mathematics* **9**, 368 (2021).

[2] Clarke, E. M., Klieber, W., Nováček, M. & Zuliani, P. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, 1–30 (Springer, 2011).

[3] Clarke, E. M., Grumberg, O., Minea, M. & Peled, D. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer* **2**, 279–287 (1999).

[4] Clarke, E. M., Grumberg, O. & Long, D. E. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)* **16**, 1512–1542 (1994).

[5] Clarke, E., Grumberg, O., Jha, S., Lu, Y. & Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* **50**, 752–794 (2003).

[6] Meseguer, J., Palomino, M. & Martí-Oliet, N. Equational abstractions. In *Automated Deduction–CADE-19: 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28–August 2, 2003. Proceedings 19*, 2–16 (Springer, 2003).

[7] Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)* **24**, 293–318 (1992).

[8] De Roever, W.-P. *Concurrency verification: Introduction to compositional and non-compositional methods*, vol. 54 (Cambridge University Press, 2001).

[9] Zhang, F. *et al.* Compositional reasoning for shared-variable concurrent programs. In *Formal Methods: 22nd International Symposium, FM*

2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings 22, 523–541 (Springer, 2018).

[10] Aung, M. N., Phyo, Y. & Ogata, K. Formal specification and model checking of the lim-jeong-park-lee autonomous vehicle intersection control protocol (s). In *SEKE*, vol. 2019, 159–208 (2019).

[11] Lim, J., Jeong, Y. S., Park, D.-S. & Lee, H. An efficient distributed mutual exclusion algorithm for intersection traffic control. *The Journal of Supercomputing* **74**, 1090–1107 (2018).

[12] Clavel, M. *et al. All about maude-a high-performance logical framework: how to specify, program, and verify systems in rewriting logic*, vol. 4350 (Springer, 2007).

[13] Clarke, E., Biere, A., Raimi, R. & Zhu, Y. Bounded model checking using satisfiability solving. *Formal methods in system design* **19**, 7–34 (2001).

[14] Holzmann, G. J. *The SPIN model checker: Primer and reference manual*, vol. 1003 (Addison-Wesley Reading, 2004).

[15] Sheeran, M., Singh, S. & Stålmarck, G. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, 127–144 (Springer, 2000).

[16] De Moura, L., Rueß, H. & Sorea, M. Bounded model checking and induction: From refutation to verification: (extended abstract, category a). In *International Conference on Computer Aided Verification*, 14–26 (Springer, 2003).

[17] McMillan, K. L. Interpolation and sat-based model checking. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, 1–13 (Springer, 2003).

[18] Holzmann, G. J. An improved protocol reachability analysis technique. *Software: Practice and Experience* **18**, 137–161 (1988).

[19] Peled, D. Combining partial order reductions with on-the-fly model-checking. In *Computer Aided Verification: 6th International Conference, CAV'94 Stanford, California, USA, June 21–23, 1994 Proceedings 6*, 377–390 (Springer, 1994).

[20] Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Syst.* **1**, 6–16 (1990). URL `https://doi.org/10.1109/71.80120`.

[21] Mellor-Crummey, J. M. & Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* **9**, 21–65 (1991). URL `https://doi.org/10.1145/103727.103729`.

[22] Bui, D. D. & Ogata, K. Better state pictures facilitating state machine characteristic conjecture. In *26th DMSVIVA*, 7–12 (KSI Research Inc., 2020). URL `https://doi.org/10.18293/DMSVIVA20-007`.

[23] Do, C. M., Phyo, Y., Riesco, A. & Ogata, K. Optimization techniques for model checking leads-to properties in a stratified way. *ACM Transactions on Software Engineering and Methodology* (2023).

[24] Phyo, Y., Minh Do, C. & Ogata, K. A divide & conquer approach to leads-to model checking. *The Computer Journal* **65**, 1353–1364 (2022).

[25] Phyo, Y., Do, C. M. & Ogata, K. A divide & conquer approach to conditional stable model checking. In *ICTAC*, vol. 12819 of *Lecture Notes in Computer Science*, 105–111 (Springer, 2021).

[26] Phyo, Y., Aung, M. N., Do, C. M. & Ogata, K. A layered and parallelized method of eventual model checking. *Information* **14**, 384 (2023).

Publications

1. Moe Nandi Aung, Yati Phyo, Canh Do Minh, Kazuhiro Ogata. "A Divide & Conquer Approach to Eventual Model Checking." *Mathematics* 2021, 9, 368. DOI: 10.3390/math9040368

2. Moe Nandi Aung, Yati Phyo, Kazuhiro Ogata. "Formal Specification and Model Checking of the Lim-Jeong-Park-Lee Autonomous Vehicle Intersection Control Protocol." In *31st International Conference on Software Engineering and Knowledge Engineering (31st SEKE)*, KSI Research Inc., pp.159-164 (2019). DOI: 10.18293/SEKE2019-021

3. Moe Nandi Aung, Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. "A Tool for Model Checking Eventual Model Checking in a Stratified Way." In *International Conference on Dependable Systems and Their Applications (DSA 2022)*, IEEE, pp.270-279 (2022). DOI: 10.1109/DSA56465.2022.00045

4. Yati Phyo, Moe Nandi Aung, Canh Minh Do, and Kazuhiro Ogata. "A Layered and Parallelized Method of Eventual Model Checking." *Information* 2023, 14, 384 . DOI: 10.3390/info14070384