A Divide & Conquer Approach to Leads-to Model Checking

YATI PHYO\textsuperscript{1}, CANH MINH DO\textsuperscript{1} and KAZUHIRO OGATA\textsuperscript{1,*}

\textsuperscript{1}School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi, Ishikawa 923-1211, Japan
*Corresponding author: ogata@jaist.ac.jp

The paper proposes a new technique to mitigate the state explosion in model checking. The technique is called a divide & conquer approach to leads-to model checking. As indicated by the name, the technique is dedicated to leads-to properties. It is known that many important systems requirements can be expressed as leads-to properties, thus it is worth focusing on leads-to properties. The technique divides an original leads-to model checking problem into multiple smaller model checking problems and tackles each smaller one. We prove a theorem that the multiple smaller model checking problems are equivalent to the original leads-to model checking problem. We conduct two case studies demonstrating the power of the proposed technique.

Keywords: leads-to properties; liveness properties; model checking; Maude; state explosion.

Received 01 June 2020; Revised 08 September 2020; Accepted 08 November 2020
Handling editor: Natasha Alechina

1. INTRODUCTION

Model checking \cite{1} is one of the most successful computer science achievements in the past few decades. This is why Edmund M. Clarke, together with E. Allen Emerson and Joseph Sifakis, was honored with the 2007 A. M. Turing Award\textsuperscript{1} for their role in developing model checking into a highly effective verification technology. Model checking has been widely adopted in the hardware and software industries. There are still some challenges to tackle in model checking, one of which is the state explosion, the most challenging one. Many techniques to mitigate the state explosion, such as partial order reduction \cite{2} and abstraction \cite{3–5}, have been devised as those existing techniques are not enough to deal with the state explosion; however, it is still worth tackling.

The present paper proposes a new technique to mitigate the state explosion. The technique is called a divide & conquer approach to leads-to model checking. As indicated by the name, the technique is dedicated to leads-to properties. An informal description of leads-to properties is that whenever something occurs, something else will eventually occur. For example, one desired property mutual exclusion protocols should enjoy can be expressed as a leads-to property: whenever a process wants to enter a critical section, it will eventually be there. Chandy and Misra \cite{6} designed a temporal logic called UNITY in which the leads-to temporal connective plays an important role and demonstrated that many important systems requirements can be expressed as leads-to properties. Therefore, it is worth focusing on leads-to properties.

The proposed technique splits each infinite state sequence $s_0^1, s_1^1, \ldots, s_n^1, s_0^m, s_1^m, \ldots, s_n^m$ (generated from a Kripke structure) into multiple (say $m$) sub-sequences: for each $s_i^0, \ldots, s_i^{n_i}$ (for $i = 1, \ldots, m - 1$) of all the sub-sequences except for the final one, we add the final state $s_{n_i}^i$ infinitely many times to the end, generating the infinite sequence $s_0^1, \ldots, s_i^1, s_i^2, \ldots$ and conduct a model checking experiment for each infinite sequence. Note that the number of different states in the original infinite state sequence (and also each sub-sequence including the final one) is bounded if there are a bounded number of reachable states. If the number of different states

\footnote{1 https://amturing.acm.org/award_winners/clarke_1167964.cfm}
in each sub-sequence is much smaller than the one in the original infinite state sequence, it would be feasible to conduct the model checking experiment for the infinite sequence generated from each sub-sequence (including the final one) even though it is impossible to conduct the model checking experiment for the original infinite state sequence due to the state explosion. The proposed technique makes it possible to use any existing linear temporal logic (LTL) model checking algorithm and then any existing LTL model checker. We prove a theorem that the original leads-to model checking problem for the original infinite sequence is equivalent to the multiple model checking problems for the multiple infinite sequences.

We design an algorithm based on the theorem and conduct two case studies demonstrating the power of the proposed technique. One case study is a model checking experiment that Qlock protocol, an abstract version of the Dijkstra binary semaphore, enjoys the lockout freedom property that can be expressed as a leads-to property, when 10 processes participate in the protocol, which has been successfully tackled with the proposed technique that uses two layers but has not without it. The other is a model checking experiment that MCS protocol [7], whose variants have been used in Java VMs, enjoys the lockout freedom property, when six processes participate in the protocol, which has been successfully tackled with the proposed technique that uses three layers but has not without it.

The remaining part of the paper is organized as follows. Section 2 explains some preliminaries, such as Kripke structures and LTL. Section 3 describes the theoretical part of the proposed technique. Section 4 describes the proposed technique. Section 5 reports on two case studies. Section 6 mentions some existing related work. Section 7 concludes the paper including some future directions.

2. PRELIMINARIES

We use the symbol `≜` as ‘if and only if’ or ‘be defined as.’

**Definition 2.1. (Kripke structures).** A Kripke structure $K ≜ (S, I, T, A, L)$ consists of a set $S$ of states, a set $I ⊆ S$ of initial states, a left-total binary relation $T ⊆ S × S$ over states, a set $A$ of atomic propositions and a labeling function $L$ whose type is $S → 2^A$. An element $(s, s′) ∈ T$ is called a (state) transition from $s$ to $s′$ and may be written as $s →_K s′$.

$S$ does not need to be finite. The set $R$ of reachable states is inductively defined as follows: $I ⊆ R$ and if $s ∈ R$ and $(s, s′) ∈ T$, then $s′ ∈ R$. We suppose that $R$ is finite. $K$ in $s →_K s′$ may be omitted if it is clear from the context.

An infinite sequence $s_0, s_1, s_2, ..., s_{i+1}, ...$ of states is called a path of $K$ if and only if for any natural number $i$, $(s_i, s_{i+1}) ∈ T$. Let $π$ be $s_0, s_1, s_2, ..., s_i, ..., s_{i+1}, ...$ and some notations are defined as follows:

$$
π(i) ≜ s_i
\pi_1 ≜ s_i, s_{i+1}, ...
\pi_1 ≜ s_0, s_1, ..., s_i, s_j, ...
\pi_∞ ≜ π
\pi(i,j) ≜ \begin{cases} s_i, s_i+1, ..., s_j, s_j, ... & \text{if } i ≤ j \\ s_i, s_i, ... & \text{otherwise} \end{cases}
$$

where $i$ and $j$ are any natural numbers. Note that $\pi(0,j) = π_j$.

Note that $π_i(k) = π(k)$ if $k = 0, ..., i$ and $π_i(k) = π(j)$ if $k > i$. Note that $π_{i,j}(k) = π(k) + j$ if $i ≤ j & k = 0, ..., m$, where $j = i + m$. Then, $\pi_{i,j}(k) = π(j)$ if $i ≤ j & k > j$ and $\pi_{i,j}(k) = π(k)$ if $i > j & k$ is a natural number. A path $π$ of $K$ is called a computation of $K$ if and only if $π(0) ∈ I$.

Let $P_K$ be the set of all paths of $K$. Let $P_{K,s} = \{π | π ∈ P_K, π(0) = s\}$, where $s ∈ S$. Let $P_{K,s} = \{π_k | π ∈ P_{K,s}\}$, where $s ∈ S$ and $b$ is a natural number. Note that $P_{K,s}^{∞} = P_{K,s}$.

**Definition 2.2. (Syntax of LTL).** The syntax of LTL is as follows:

$$\varphi ::= a | \top | ¬\varphi | \varphi ∨ \varphi | ∅ \varphi | \varphi \cup \varphi,$$

where $a ∈ A$.

**Definition 2.3. (Semantics of LTL).** For any Kripke structure $K$, any path $π$ of $K$ and any LTL formulas $ϕ$, $K, π ⊨ ϕ$ is inductively defined as follows:

- $K, π ⊨ a$ if and only if $a ∈ L(π(0))$
- $K, π ⊨ T$
- $K, π ⊨ ¬ϕ_1$ if and only if $K, π ⊭ ϕ_1$
- $K, π ⊨ ϕ_1 ∨ ϕ_2$ if and only if $K, π ⊨ ϕ_1$ and/or $K, π ⊨ ϕ_2$
- $K, π ⊨ ∅ ϕ_1$ if and only if $K, π^1 ⊨ ϕ_1$
- $K, π ⊨ ϕ_1 \cup ϕ_2$ if and only if there exists a natural number $i$ such that $K, π^i \equiv ϕ_2$ and for each natural number $j < i$, $K, π^j \equiv ϕ_1$

where $ϕ_1$ and $ϕ_2$ are LTL formulas. Then, $K, π ⊨ ϕ$ if and only if $K, π ⊨ ϕ$ for all computations $π$ of $K$.

$⊥ ≜ ¬\top$ and some other connectives are defined as follows:

$ϕ_1 \land ϕ_2 ≜ ¬((¬ϕ_1) ∨ (¬ϕ_2))$, $ϕ_1 ⇒ ϕ_2 ≜ ¬(¬ϕ_1) ∨ ϕ_2$, $ϕ_1 ⇔ ϕ_2 ≜ (ϕ_1 ⇒ ϕ_2) ∧ (ϕ_2 ⇒ ϕ_1)$, $ϕ_1 ≜ T \cup ϕ_1$, $□ϕ_1 ≜ (¬(¬ϕ_1))$ and $ϕ_1 ≅ ϕ_2 ≜ □(ϕ_1 ⇒ ϕ_2)$. $\cup, \circ, ◇$ and $¬$ are called next, until, eventually, always and leads-to temporal connectives, respectively. Although it is unnecessary
to directly define the semantics for $\Diamond$, $\square$ and $\rightarrow$, we can define it as follows:

- $K, \pi \models \Diamond \varphi_1$ if and only if there exists a natural number $i$ such that $K, \pi^i \models \varphi_1$
- $K, \pi \models \square \varphi_1$ if and only if for all natural numbers $i$, $K, \pi^i \models \varphi_1$
- $K, \pi \models \varphi_1 \rightarrow \varphi_2$ if and only if for each natural number $i$ such that $K, \pi^i \models \varphi_1$, there exists a natural number $j \geq i$ such that $K, \pi^j \models \varphi_2$.

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

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

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

Leads-to properties are those that are expressed in the form of $\varphi_1 \rightsquigarrow \varphi_2$, where $\varphi_1$ and $\varphi_2$ are LTL formulas. In this paper, we give the constraints to $\varphi_1$ and $\varphi_2$: $\varphi_1$ and $\varphi_2$ are state propositions. Chandy and Misra [6] demonstrate that many important systems requirements can be expressed as $\varphi_1 \rightsquigarrow \varphi_2$, where $\varphi_1$ and $\varphi_2$ are state propositions.

Let $K, s \models \varphi$, where $s \in S$, be $K, \pi \models \varphi$ for all $\pi \in P(K,s)$. 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 $\infty$, be $K, \pi \models \varphi$ for all $\pi \in P(K,s)$. Note that $K, s, \infty \models \varphi$ is $K, s \models \varphi$.

Some logical connectives are abused for $K, \pi \models \varphi$ as follows:

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

3. **MULTIPLE-LAYER DIVISION OF LEADS-TO MODEL CHECKING**

First, we prove in Theorem 3.1 that a leads-to model checking problem for a Kripke structure $K$ and a path $\pi$ of $K$ is equivalent to a model checking problem that is composed of three sub-model checking problems for $K$ and two paths of $K$, where the two paths are obtained by splitting $\pi$ into two parts. We then prove in Theorem 3.2 that a leads-to model checking problem is equivalent to a model checking problem that is composed of $2L + 1$ sub-model checking problems for $K$ and $L + 1$ paths of $K$, where $L \geq 1$ and the $L + 1$ paths are obtained by splitting $\pi$ into $L + 1$ parts.
\( K, \pi^1_k \models \varphi_2 \). Because \( \varphi_1 \) is a state proposition, \( K, \pi^1 \models \varphi_1 \) from Proposition 2.1. Because \( \varphi_2 \) is a state proposition, \( K, \pi^j \not\models \varphi_2 \) for any \( j \) such that \( i \leq j \leq k \) from Proposition 2.1. From the assumption \( K, \pi \models \pi \models \varphi_2 \) as well, however, there must exist \( j' > k \) such that \( K, \pi^{j'} \models \varphi_2 \). Thus, \( K, \pi^k \models \varphi_2 \).

The third lemma says that if \( \varphi_2 \) holds for the second layer and \( \varphi_1 \models \varphi_2 \) holds for the second layer, then \( \varphi_1 \models \varphi_2 \) holds for the whole reachable state space.

**Lemma 3.3.** For any natural number \( k \), \( (K, \pi^k \models \varphi_2) \land (K, \pi^k \models \varphi_1 \models \varphi_2) \Rightarrow (K, \pi \models \varphi_1 \models \varphi_2) \).

**Proof.** From the assumption \( K, \pi^k \models \varphi_2 \), there exists \( i \geq k \) such that \( K, \pi^i \models \varphi_2 \). From the assumption \( K, \pi^i \models \varphi_1 \) as well, for each \( j \geq i \leq k \) such that \( K, \pi^j \models \varphi_1 \), there exists \( j' \geq j \leq i \) such that \( K, \pi^{j'} \models \varphi_2 \). Thus, \( K, \pi \models \varphi_1 \models \varphi_2 \).

The fourth lemma says that if \( \varphi_1 \models \varphi_2 \) holds for the first layer and \( \varphi_1 \models \varphi_2 \) holds for the second layer, then \( \varphi_1 \models \varphi_2 \) holds for the whole reachable state space.

**Lemma 3.4.** Let \( \varphi_1 \) and \( \varphi_2 \) be any state propositions of \( K \). For any natural number \( k \), \( (K, \pi_k \models \varphi_1 \models \varphi_2) \land (K, \pi^k \models \varphi_1 \models \varphi_2) \Rightarrow (K, \pi \models \varphi_1 \models \varphi_2) \).

**Proof.** From the assumption \( K, \pi_k \models \varphi_1 \models \varphi_2 \), for each \( i \) such that \( K, \pi^i \models \varphi_1 \), there exists \( j \geq i \) such that \( K, \pi^j \models \varphi_2 \). Because \( \varphi_1 \) and \( \varphi_2 \) are state propositions, \( K, \pi^i \models \varphi_1 \) and \( K, \pi^j \models \varphi_2 \) for \( i \) and \( j \) such that \( 0 \leq i \leq j \leq k \) from Proposition 2.1. From the assumption \( K, \pi^k \models \varphi_1 \models \varphi_2 \) as well, for each \( i \) such that \( K, \pi^{k+i} \models \varphi_1 \), there exists \( j \geq i \) such that \( K, \pi^{k+j} \models \varphi_2 \). Thus, \( K, \pi \models \varphi_1 \models \varphi_2 \).

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

**Theorem 3.1.** (Two layer division of \( \models \)). Let \( \varphi_1 \) and \( \varphi_2 \) be any state propositions of \( K \). For any natural number \( k \), \( (K, \pi \models \varphi_1 \models \varphi_2) \Leftrightarrow ((K, \pi_k \models \varphi_1 \models \varphi_2) \land (K, \pi^k \models \varphi_1 \models \varphi_2)) \).

**Proof.** (1) Case “only if” \( (\Rightarrow) \): It follows from Lemma 3.1 and Lemma 3.2. (2) Case “if” \( (\Leftarrow) \): If \( K, \pi_k \models \varphi_1 \models \varphi_2 \), then it follows from Lemma 3.4. Otherwise, namely if \( K, \pi_k \not\models \varphi_1 \models \varphi_2 \), then it follows from Lemma 3.3.

Theorem 3.1 makes it possible to divide the original model checking problem \( K, \pi \models \varphi_1 \models \varphi_2 \) into the three sub-model checking problems \( K, \pi_k \models \varphi_1 \models \varphi_2 \), \( K, \pi^k \models \varphi_1 \models \varphi_2 \) and \( K, \pi^k \models \varphi_1 \models \varphi_2 \).

We need to have two more lemmas to prove Theorem 3.2 whose description uses two abbreviations of statements on ternary satisfaction relations among Kripke structures, their paths and LTL formulas. The first one of the two lemmas says that an eventually model checking problem for a Kripke structure \( K \) and a path \( \pi \) of \( K \) is equivalent to a model checking problem that is composed of two sub-model checking problems for \( K \) and two paths of \( K \) where the two paths are obtained by splitting \( \pi \) into two parts.

**Lemma 3.5.** Let \( \varphi_1 \) be any state proposition of \( K \). For any natural number \( k \), \( (K, \pi \models \varphi_1) \Leftrightarrow ((K, \pi_k \models \varphi_1) \lor ((K, \pi_k \models \varphi_1) \Rightarrow (K, \pi^k \models \varphi_1)) \).

**Proof.** (1) Case “only if” \( (\Rightarrow) \): There must be \( i \) such that \( K, \pi^i \models \varphi_1 \). If \( i \leq k \), \( K, \pi^i \models \varphi_1 \) from Proposition 2.1 because \( \varphi_1 \) is a state proposition. Thus, \( K, \pi_k \models \varphi_1 \). Otherwise, \( K, \pi_k \not\models \varphi_1 \) and \( K, \pi^i \models \varphi_1 \). Hence, \( K, \pi^k \models \varphi_1 \). (2) Case “if” \( (\Leftarrow) \): If \( K, \pi_k \models \varphi_1 \), there must be \( j \) such that \( j > k \) and \( K, \pi^j \models \varphi_1 \). Thus, \( K, \pi \models \varphi_1 \).

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

**Lemma 3.6.** Let \( \varphi_1 \) and \( \varphi_2 \) be any state propositions of \( K \). For any natural number \( k \), \( (K, \pi \models \varphi_1 \models \varphi_2) \Leftrightarrow ((K, \pi_k \models \varphi_1 \models \varphi_2) \lor ((K, \pi_k \models \varphi_1 \models \varphi_2) \Rightarrow (K, \pi^k \models \varphi_1 \models \varphi_2)) \).

**Proof.** It follows from Theorem 3.1 and Lemma 3.5.

We define Leads2_2 and E&L-Leads2_2 that are abbreviations of statements on ternary satisfaction relations among Kripke structures, their paths and LTL formulas. To this end, let \( \pi \) be split into \( L + 1 \) paths \( \pi^{(d(0),d(1))}, \pi^{(d(2),d(l+1))}, \ldots, \pi^{(d(L),d(l+1))} \) where \( L \geq 1 \), \( d(0) = 0 \), \( d(L + 1) = \infty \) and \( d(l) \) is a non-zero natural number for \( l = 1, \ldots, L \). (see Fig. 2). \( \pi^{(d(l),d(l+1))} \) can be regarded as the \( l + 1 \) st layer of the reachable state space for \( l = 0, 1, \ldots, L \).
Definition 3.1. (Leads₂L and E&Leads₂L). Let L be any non-zero natural number, k be any natural number and d be any function such that d(0) is 0, d(x) is a non-zero natural number for x = 1, ..., L and d(L + 1) is infinite.

1. 0 ≤ k < L - 1

\[
\text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, k) \triangleq ((K, \pi^{(d(k), d(k+1))}) \models \varphi_1 \leadsto \varphi_2) \\
\quad \Rightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, k + 1)) \land \\
((K, \pi^{(d(k), d(k+1))}) \not\models \varphi_1 \leadsto \varphi_2) \\
\quad \Rightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, k + 1))
\]

\[
\text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, k) \triangleq ((K, \pi^{(d(k), d(k+1))}) \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \\
\quad \Rightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, k + 1)) \land \\
((K, \pi^{(d(k), d(k+1))}) \not\models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \\
\quad \Rightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, k + 1))
\]

2. k = L - 1

\[
\text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, L) \triangleq ((K, \pi^{(d(L), d(L+1))}) \models \varphi_1 \leadsto \varphi_2) \\
\quad \Rightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, L + 1)) \land \\
((K, \pi^{(d(L), d(L+1))}) \not\models \varphi_1 \leadsto \varphi_2) \\
\quad \Rightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, L + 1))
\]

\[
\text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, L) \triangleq ((K, \pi^{(d(L), d(L+1))}) \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \\
\quad \Rightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, L + 1)) \land \\
((K, \pi^{(d(L), d(L+1))}) \not\models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \\
\quad \Rightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, L + 1))
\]

We are ready to prove that a leads-to model checking problem \( K, \pi \models \varphi_1 \leadsto \varphi_2 \) for a Kripke structure \( K \) and a path \( \pi \) of \( K \) is equivalent to a model checking problem that is composed of 2L+1 sub-model checking problems \( K, \pi^{d(1)} \models \varphi_1 \leadsto \varphi_2, \ldots, K, \pi^{d(2L)} \models \varphi_1 \leadsto \varphi_2, K, \pi^{d(2L+1)} \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2) \) for \( K \) and the \( L + 1 \) paths \( \pi^{d(0)(d(1))}, \ldots, \pi^{d(L)(d(L+1))}, \pi^{d(L+2)(d(L+3))} \) for \( K \).

Theorem 3.2. (L + 1-layer division of \( \leadsto \)). Let \( L \) be any non-zero natural number. Let \( d(0) = 0, d(x) \) be any non-zero natural number for \( x = 1, \ldots, L \) and \( d(L + 1) = \infty \). Let \( \varphi_1 \) and \( \varphi_2 \) be any state propositions of \( K \). Then,

1. \( (K, \pi \models \varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0) \)
2. \( (K, \pi \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \Leftrightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0) \)

Proof. (1) and (2) are proved by simultaneous induction [8] on \( L \).

- Base case \( (L = 1) \): It follows from Theorem 3.1 and Lemma 3.6.
- Induction case \( (L = l + 1) \): The induction hypotheses2 are as follows:

\[
\begin{align*}
& (K, \pi^{d(1)}) \models \varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{Leads}_2L(K, \pi^{d(1)}, \varphi_1, \varphi_2, 0) \\
& (K, \pi^{d(1)}) \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{E&Leads}_2L(K, \pi^{d(1)}, \varphi_1, \varphi_2, 0)
\end{align*}
\]

From the definition of Leads₂L and E&Leads₂L,

\[
\begin{align*}
& \text{Leads}_2L(K, \pi^{d(1)}, \varphi_1, \varphi_2, 0) \text{ is Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0) \\
& \text{E&Leads}_2L(K, \pi^{d(1)}, \varphi_1, \varphi_2, 0) \text{ is E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0)
\end{align*}
\]

Therefore, the induction hypotheses can be rephrased as follows:

\[
\begin{align*}
& (K, \pi^{d(1)}) \models \varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, 1) \\
& (K, \pi^{d(1)}) \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, 1)
\end{align*}
\]

From the definition of Leads₂L, Leads₂L(K, \pi, \varphi_1, \varphi_2, 0) is

\[
\begin{align*}
& ((K, \pi^{d(0)}, d(1))) \models \varphi_1 \leadsto \varphi_2) \\
& \Rightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, 1)) \land \\
((K, \pi^{d(0)}, d(1))) \not\models \varphi_1 \leadsto \varphi_2) \\
& \Rightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, 1))
\end{align*}
\]

which is

\[
\begin{align*}
& ((K, \pi^{d(0)}, d(1))) \models \varphi_1 \leadsto \varphi_2) \\
& \Rightarrow (K, \pi^{d(1)}) \models \varphi_1 \leadsto \varphi_2) \land \\
((K, \pi^{d(0)}, d(1))) \not\models \varphi_1 \leadsto \varphi_2) \\
& \Rightarrow (K, \pi^{d(1)}) \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2))
\end{align*}
\]

2 Precisely, the induction hypotheses are \((K, \pi \models \varphi_1 \leadsto \varphi_2) \Leftrightarrow \text{Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0)\) and \((K, \pi \models (\varphi_2) \land (\varphi_1 \leadsto \varphi_2)) \Leftrightarrow \text{E&Leads}_2L(K, \pi, \varphi_1, \varphi_2, 0)\) for all Kripke structure \( K \), all paths \( \pi \) of \( K \) and all LTL state propositions \( \varphi_1 \) & \( \varphi_2 \) of \( K \). What are used in the proof are the instances of the hypotheses obtained by replacing \( \pi \) with \( \pi^{d(l)} \).
because of the induction hypotheses. From Theorem 3.1, this is equivalent to $K, \pi \models \varphi_1 \rightarrow \varphi_2$.

- From the definition of $E\&Leads_2$, $E\&Leads_{2_{l+1}}(K, \pi, \varphi_1, \varphi_2, 0)$ is

$$(K, \pi^{(d(0),d(1))} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2))$$

$$\Rightarrow E\&Leads_{2_{l+1}}(K, \pi, \varphi_1, \varphi_2, 1)) \land$$

$$(K, \pi^{(d(0),d(1))} \not\models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2))$$

$$\Rightarrow E\&Leads_2(K, \pi, \varphi_1, \varphi_2, 1)),$$

which is

$$(K, \pi^{(d(0),d(1))} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2))$$

$$\Rightarrow (K, \pi^{d(1)} \models \varphi_1 \rightarrow \varphi_2) \land$$

$$(K, \pi^{(d(0),d(1))} \not\models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2))$$

$$\Rightarrow (K, \pi^{d(1)} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)).$$

because of the induction hypotheses. From Lemma 3.6, this is equivalent to $K, \pi \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$.

Theorem 3.2 makes it possible to divide the original model checking problem $K, \pi \models \varphi_1 \rightarrow \varphi_2$ into the $2L + 1$ sub-model checking problems $K, \pi^{d(0),d(1)} \models \varphi_1 \rightarrow \varphi_2$, ..., $K, \pi^{(d(l),d(l+1))} \models \varphi_1 \rightarrow \varphi_2$, ..., $K, \pi^{d(L)} \models \varphi_1 \rightarrow \varphi_2$. The code fragment at lines 6–10 checks $K, \pi^{d(l-1),d(l)} \models \varphi_1 \rightarrow \varphi_2$ for some paths $\pi^{d(l-1),d(l)}$ in the $l$th layer of $R$ for $l = 1, \ldots, L$. The code fragment at lines 11–15 checks $K, \pi^{d(l),d(l+1)} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$ for the remaining paths $\pi^{d(l),d(l+1)}$ in the $l$th layer of $R$ for $l = 1, \ldots, L$. When $l$ is 1, $\mathit{LS}$ consists of all initial states and $\mathit{ELS}$ is $\emptyset$. The code fragment at lines 18–21 checks $K, \pi^{d(l),d(l+1)} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$ for some paths $\pi^{d(l),d(l+1)}$ in the $L + 1$st layer of $R$, where $d(l+1)$ is $\infty$ and then $\pi^{d(l),d(l+1)}$ is $\mathit{ELS}(l+1)$.

4. A DIVIDE & CONQUER APPROACH TO LEADS-TO MODEL CHECKING ALGORITHM

An algorithm can be constructed based on Theorem 3.2, which is shown as Algorithm 1. For each initial state $s_0 \in K$, unfolding $s_0$ by using $T$ without sharing any nodes that are states, an infinite tree whose root is $s_0$ is made. Such an infinite tree can be divided into $L + 1$ layers as shown in Fig. 3, where $L$ is a non-zero natural number. Because $R$ is finite, the number of different states in each layer is finite. Theorem 3.2 makes it possible to check $K \models \varphi_1 \rightarrow \varphi_2$ in a stratified way in that for each layer $l \in \{1, \ldots, L + 1\}$ we can check $K, s, d(l) \models \varphi$ for each $s \in \{\pi^{d(l-1),d(l)} \mid \pi \in P^{d(l-1)}(s_0)\}$, where $d(0)$ is 0, $d(x)$ is a non-zero natural number for $x = 1, \ldots, L, d(L + 1)$ is $\infty$ and $\varphi$ is $\varphi_1 \rightarrow \varphi_2$ or $(\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$. The code fragment at lines 6–10 checks $K, \pi^{d(l-1),d(l)} \models \varphi_1 \rightarrow \varphi_2$ for some paths $\pi^{d(l-1),d(l)}$ in the $l$th layer of $R$ for $l = 1, \ldots, L$. The code fragment at lines 11–15 checks $K, \pi^{d(l),d(l+1)} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$ for the remaining paths $\pi^{d(l),d(l+1)}$ in the $l$th layer of $R$ for $l = 1, \ldots, L$. When $l$ is 1, $\mathit{LS}$ consists of all initial states and $\mathit{ELS}$ is $\emptyset$. The code fragment at lines 18–21 checks $K, \pi^{d(l),d(l+1)} \models (\Diamond \varphi_2) \land (\varphi_1 \rightarrow \varphi_2)$ for some paths $\pi^{d(l),d(l+1)}$ in the $L + 1$st layer of $R$.

The two assignments at lines 9 & 10 (and those at lines 14 & 15 as well) could be replaced with $\mathit{ELS}' \leftarrow \mathit{ELS}' \cup \{\pi^{d(l)}\}$ and the condition at line 13 (and that at line 24) could be replaced with $K, \pi \not\models \Diamond \varphi_2$. The proposed technique can be interpreted as follows. For each initial state, the reachable state space from the initial state is split into multiple layers ($L + 1$ layers) and generates multiple smaller sub-state spaces as shown in Fig. 4. For each smaller
sub-state space, we check if $\varphi_1 \Rightarrow \varphi_2$ and/or $\Diamond \varphi_2$ holds. If the number of different states in each sub-state space is much less than the one in the entire reachable state space, then model checking for each sub-state space is feasible even though model checking for the entire reachable state space is not.

Although Algorithm 1 does not construct a counterexample when Failure is returned, it could be constructed. For each $l \in \{0, 1, \ldots, L\}$, $LS_l$ and $ELS_l$ are prepared. As elements of $LS_l$ and $ELS_l$, pairs $(s, s')$ are used, where $s$ is a state in $S$ or a dummy state denoted $\delta$-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 4 should be revised as follows:

$$LS' \leftarrow \{(s, \pi(d(l))) | (s_1, s) \in LS_{l-1} \cup ELS_{l-1}, \pi \in F^{(l)}(K, s)\}$$

the condition at line 6 should be revised as $\langle s_1, s \rangle \in LS_{l-1}$, the condition at line 11 should be revised as $\langle s_1, s \rangle \in ELS_{l-1}$, the condition at line 18 should be revised as $\langle s_1, s \rangle \in LS_L$, the condition at line 22 should be revised as $\langle s_1, s \rangle \in ELS_l$, the two assignments at lines 9 & 10 (and lines 14 & 15 as well) should be revised as follows: $LS' \leftarrow LS' \setminus \{(s, \pi(d(l)))\}$ $ELS' \leftarrow ELS' \cup \{(s, \pi(d(l)))\}$

and the two assignments at lines 16 & 17 should be revised as follows: $LS_l \leftarrow LS' \cup ELS'$ $LS_0$ and $ELS_0$ are set to $\{(\delta$-stt, $s) | s \in I\}$ and $\emptyset$, respectively.

We could then construct a counterexample, when Failure is returned, by searching through $LS_L, ELS_L, \ldots, LS_1, ELS_1, LS_0$ and $ELS_0$.

The number of different states in each layer shown in Fig. 3 should be much smaller than the one of different states in all layers in order to make the proposed technique effective. The number of different states in the $x$th one of the first $L$ layers depends on $d(x)$ and can be much smaller than the one of different states in all layers but the number of different states in the $L + 1$st layer may not be much smaller than that. This could happen if there is a long backward transition from a state $s_{m+n}$ to another state $s_m$ as shown in Fig. 5 such that $n \gg 0$ or $s_{m+n}$ is far from $s_m$. Thus, we should write a systems specification such that there is no such a backward transition in it in order to effectively use the proposed technique.

5. CASE STUDIES

We have mainly used a computer (referred as Computer A) that carries a 3.00GHz microprocessor with 16GB memory to conduct experiments. Maude [9] is a rewriting logic-based specification/programming language. It is equipped with an LTL model checker. We have used Maude in the case studies.

5.1. Qlock: A Mutual Exclusion Protocol

Qlock is a shared-memory mutual exclusion protocol for multiple processes and can be regarded as an abstract version of the Dijkstra binary semaphore in that the queue used is an atomic object. Let us consider that each process would like to enter a critical section once. Each process $p$ is initially located in a place called the start section (ss), putting its ID into a queue shared by all processes (see Fig. 6 (a)) and moving to a place called the wait section (ws). Process $p$ then waits there until its ID becomes the top of the queue (see Fig. 6 (b)). If that is the case, it can enter the critical section (cs) (see Fig. 6 (b)). When it finishes doing what it is supposed to do in the cs, it leaves there, deleting the top from the queue and moving to a place called the finish section (fs) (see Fig. 6 (d)).

A state is expressed as a braced soup of observable components, where observable components are name-value pairs and

---

**FIGURE 4.** Multiple sub-state spaces obtained by splitting the reachable state space into $L + 1$ layers.

**FIGURE 5.** A long backward transition.

(a) start section (ss)

A process $p$ puts its ID $\langle p \rangle$ into a shared queue at end.

(b) wait section (ws)

$p$ waits until it becomes the top of the queue.

(c) critical section (cs)

$p$ enters a critical section when it is the top of the queue.

(d) finish section (fs)

$p$ leaves the cs and deletes the top from the queue.

**FIGURE 6.** Qlock.
soups are associative-commutative collections. When there are \( n \) processes, the initial state of Qlock is as follows:

\[
\{(\text{queue: empq}) \quad (\text{pc}[p_1]: \text{ss}) \ldots (\text{pc}[p_n]: \text{ss}) \quad (\text{cnt: } n)\}
\]

where (queue: empq) is an observable component saying that the shared queue is empty, (pc[p_i]: ss) is an observable component saying that process \( p_i \) is in the ss and (cnt: n) is an observable component whose value is a natural number \( n \).

The role of (cnt: n) will be described later.

Transitions are described in terms of rewrite rules. The transitions of Qlock are specified as follows:

\[
\begin{align*}
\text{rl } [\text{start}] : & \quad \{(\text{queue: Q}) \quad (\text{pc}[I]: \text{ss}) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: (Q | I)}) \quad (\text{pc}[I]: \text{ws}) \quad \text{OCs}\} . \\
\text{rl } [\text{wait}] : & \quad \{(\text{queue: (I | Q)}) \quad (\text{pc}[I]: \text{ws}) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: (I | Q)}) \quad (\text{pc}[I]: \text{cs}) \quad \text{OCs}\} . \\
\text{rl } [\text{exit}] : & \quad \{(\text{queue: Q}) \quad (\text{pc}[I]: \text{cs}) \quad (\text{cnt: } N) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: } \text{deg}(Q)) \quad (\text{pc}[I]: \text{fs}) \quad (\text{cnt: } \text{dec}(N)) \quad \text{OCs}\} . \\
\text{rl } [\text{fin}] : & \quad \{(\text{cnt: } 0) \quad \text{OCs}\} \Rightarrow \{(\text{cnt: } 0) \quad \text{OCs}\} .
\end{align*}
\]

where \( Q \) is a variable of queues, \( I \) is a variable of process IDs, OCs is a variable of observable component soups and \( N \) is a variable of natural numbers. \( I \mid Q \) denotes a non-empty queue such that \( I \) is the top and \( Q \) is the remaining part of the queue. deg(Q) returns the empty queue if \( Q \) is empty and what is obtained by deleting the top from \( Q \) otherwise. dec(N) returns 0 if \( N \) is 0 and the predecessor number of \( N \) otherwise.

start, wait, exit and fin are the labels given to the four rules, respectively. Rule start says that if process \( I \) is in ss, then it puts its ID into \( Q \) at end and moves to ws. Rule wait says that if process \( I \) is in ws and the top of the shared queue is \( I \), then \( I \) enters cs. Rule exit says that if process \( I \) is in cs, then it deletes the top from the shared queue, decrements the natural number \( N \) stored in (cnt: N) and moves to fs. Rule fin says that if the natural number \( N \) stored in (cnt: N) is 0, a self-transition \( s \rightarrow_k s \) occurs. Rule fin is used to make the transitions total. The natural number \( N \) stored in (cnt: N) is the number of processes that have not yet reached fs. Use of it and rule fin make it unnecessary to use any fairness assumptions to model check a leads-to property.

Let us consider two atomic propositions \( \text{inWs1} \) and \( \text{inCs1} \). \( \text{inWs1} \) holds in a state if and only if the state matches \( \{(\text{pc}[p_1]: \text{ws}) \quad \text{OCs}\} \), namely that process \( p_1 \) is in ws. \( \text{inCs1} \) holds in a state if and only if the state matches \( \{(\text{pc}[p_1]: \text{cs}) \quad \text{OCs}\} \), namely that process \( p_1 \) is in cs. It completes soon to model check \( \text{inWs1} \rightarrow \text{inCs1} \) for Qlock when there are five processes, finding no counterexample. It is, however, impossible to model check the same property for Qlock when there are 10 processes. We then use Algorithm 1 to tackle the latter case, where \( L = 1 \) and \( d(1) = 3 \).

We use one more observable component (depth: \( d \)), where \( d \) is a natural number, to work on the 1st layer. The initial state turns into the following:

\[
\{(\text{queue: empq}) \quad (\text{pc}[p_1]: \text{ss}) \ldots (\text{pc}[p_{10}]: \text{ss}) \quad (\text{cnt: } 10) \quad (\text{depth: } 0)\}
\]

The rules turn into the following:

\[
\begin{align*}
\text{rl } [\text{start}] : & \quad \{(\text{queue: Q}) \quad (\text{pc}[I]: \text{ss}) \quad (\text{depth: } D) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: (Q | I)}) \quad (\text{pc}[I]: \text{ws}) \\
& \qquad \quad (\text{depth: } (D + 1)) \quad \text{OCs}\} . \\
\text{if } D < \text{Bound} . \\
\text{rl } [\text{wait}] : & \quad \{(\text{queue: (I | Q)}) \quad (\text{pc}[I]: \text{ws}) \\
& \qquad \quad (\text{depth: } D) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: (I | Q)}) \quad (\text{pc}[I]: \text{cs}) \\
& \qquad \quad (\text{depth: } (D + 1)) \quad \text{OCs}\} . \\
\text{if } D < \text{Bound} . \\
\text{rl } [\text{exit}] : & \quad \{(\text{queue: Q}) \quad (\text{pc}[I]: \text{cs}) \quad (\text{cnt: } N) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{queue: } \text{deg}(Q)) \quad (\text{pc}[I]: \text{fs}) \\
& \qquad \quad (\text{cnt: } \text{dec}(N)) \quad (\text{depth: } (D + 1)) \quad \text{OCs}\} . \\
\text{if } D < \text{Bound} . \\
\text{rl } [\text{fin}] : & \quad \{(\text{cnt: } 0) \quad \text{OCs}\} \Rightarrow \{(\text{cnt: } 0) \quad \text{OCs}\} . \\
\text{rl } [\text{stutter}] : & \quad \{(\text{cnt: } 0) \quad (\text{depth: } D) \quad \text{OCs}\} \\
& \quad \Rightarrow \{(\text{cnt: } 0) \quad (\text{depth: } (D + 1)) \quad \text{OCs}\} . \\
\text{if } D < \text{Bound} .
\end{align*}
\]

where \( D \) is a variable of natural numbers and Bound is 3. Rule stutter has been added to make each state at depth 3 have a transition to itself. The revised version of rule start says that if \( D \) is less than Bound and process \( I \) is in ss, then \( I \) puts its ID into \( Q \) at end and moves to ws and \( D \) is incremented. The other revised rules can be interpreted likewise. When we model check \( \text{inWs1} \rightarrow \text{inCs1} \) for the revised specification of Qlock, we find a counterexample that is a finite state sequence starting from the initial state and leading to a state loop that consists of one state that is as follows:

\[
\{(\text{queue: (p_1 | p_2 | p_3)}) \quad (\text{cnt: } 10) \\
& \quad (\text{depth: } 3) \quad (\text{pc}[p_1]: \text{ws}) \quad (\text{pc}[p_2]: \text{ws}) \\
& \quad (\text{pc}[p_3]: \text{ws}) \quad (\text{pc}[p_4]: \text{ss}) \quad (\text{pc}[p_5]: \text{ss}) \\
& \quad (\text{pc}[p_6]: \text{ss}) \quad (\text{pc}[p_7]: \text{ss}) \quad (\text{pc}[p_8]: \text{ss}) \\
& \quad (\text{pc}[p_9]: \text{ss}) \quad (\text{pc}[p_{10}]: \text{ss})\}
\]

We need to find all counterexamples and then revise the definition of \( \text{inCs1} \) such that \( \text{inCs1} \) holds in the state as well. When we model check the same property for the revised specification, we find another counterexample. This process is
repeated until no more counterexample is found. We totally find 227 counterexamples and 227 states at depth 3.

We gather all states at depth 3 from the initial state, which are totally 820 states that include the 227 states found in the last step. For each of the 820 states as an initial state, we model check \( \text{inWs1} \rightarrow \text{inCs1} \) for the original specification of Qlock, finding no counterexample. For each of the 227 states as an initial state we model check \( \emptyset \rightarrow \text{inCs1} \) for the original specification of Qlock, finding no counterexample. Therefore, we can conclude that it completes to model check \( \text{inWs1} \rightarrow \text{inCs1} \) for Qlock when there are 10 processes, finding no counterexample. It takes about 13h to conduct the model checking experiments for the second layer and it takes less than 100ms to conduct each model checking experiment for the first layer. Because there are 227 counterexamples for \( \text{inWs1} \rightarrow \text{inCs1} \) in the first layer, we need to conduct 228 model checking experiments for the first layer.

5.2. MCS: A Mutual Exclusion Protocol

MCS [7] is also a shared-memory mutual exclusion protocol for multiple processes. Its variants have been used in Java virtual machines and therefore the inventors (John M. Mellor-Crummery and Michael L. Scott) were honored with 2006 Edsger W. Dijkstra Prize in Distributed Computing3.

In MCS, one global variable \( \text{glock} \) shared by all processes participating in the protocol is used. \( \text{glock} \) contains a process ID or nop implying that no process is referred by \( \text{glock} \). Each process \( p \) maintains three local variables \( \text{next}[p] \), \( \text{pred}[p] \) and \( \text{lock}[p] \). Both \( \text{next}[p] \) and \( \text{pred}[p] \) contain a process ID or nop and \( \text{lock}[p] \) contains a Boolean value. Each process is located at one of the 14 locations \( \text{rs} \) (Remainder Section), 11, 12, 13, 14, 15, 16, cs, 17, 18, 19, 110, 111 and 112. Initially, \( \text{glock} \) is nop and for each process \( p \), \( p \) is located at \( \text{rs} \), \( \text{next}[p] \) is nop, \( \text{pred}[p] \) is nop and \( \text{lock}[p] \) is false.

Fig. 7 shows MCS written in an Algol-like pseudo-code and a graphical picture of a state of MCS when there are three processes \( p_1 \), \( p_2 \) and \( p_3 \). Each process \( p \)'s \( \text{next}[p] \) and \( \text{pred}[p] \) are used to construct a virtual queue of process IDs. \( \text{glock} \) refers to the bottom element of the queue if the queue is not empty and is nop otherwise. In Fig. 7, for example, \( p_2 \) is the bottom element of the queue and referred by \( \text{glock} \); \( \text{pred}[p_2] \) is \( p_3 \) and \( p_3 \) is the bottom but one element of the queue; \( \text{pred}[p_3] \) is \( p_1 \) and \( p_1 \) is the third element from the bottom and the top element of the queue. \( \text{next}[p_1] \) is \( p_3 \), while \( \text{next}[p_3] \) is not \( p_3 \) but nop, which means that it has been completed to enqueue \( p_3 \) into the queue, while it has not been completed to enqueue \( p_2 \) into the queue. MCS uses two non-trivial atomic operations (or instructions) that are fetch&store and comp&swap (or CAS).

5.3. Summary of case studies

The proposed divide & conquer approach to leads-to model checking makes it possible to successfully conduct the two model checking experiments: (1) \( \text{inWs1} \rightarrow \text{inCs1} \) for Qlock when there are 10 processes and each process enters the critical section once and (2) \( \text{inWs1} \rightarrow \text{inCs1} \) for MCS when there are six processes and each process enters the critical section once, which cannot be otherwise tackled by Computer A. Case (1) may be able to be tackled with SGI Computer even though
the divide & conquer approach to leads-to model checking is not used. We do not think that case (2) can be tackled even with SGI Computer if the approach is not because it would take longer than three weeks to complete case (2) but 3 weeks are the longest time for which we are allowed to continuously use SGI Computer in our institute. We should make best use of multiple nodes or cores to make the approach effective.

The specifications in Maude used in the case studies are available at the webpage5.

6. RELATED WORK

The state explosion is one of the most notorious problems in model checking. Many researchers have tackled the problem and come up with several techniques to mitigate it. Among such techniques are partial order reduction [2], symmetry reduction [12], abstraction [3–5], abstract logical model checking [13] and SAT-based bounded model checking (BMC) [14]. The proposed divide & conquer approach to leads-to model checking is a new technique to mitigate the challenge on space.

Clarke, et al. summarize several techniques that address the state space explosion problem in model checking [15]. Among them are counterexample-guided abstraction refinement (CEGAR) [4] and SAT-based BMC. For a system model whose (reachable) state space is enormous, CEGAR starts with one abstract model whose (reachable) state space is reasonably small such that it can be tractable by a model checker. If a counterexample is found, CEGAR checks if the counterexample is real. If no, CEGAR refines the abstract model based on the counterexample and repeats the experiment for the refined model until a real counterexample is found or no counterexample is found for some refined model (including the initial abstract model). It may be the case that the (reachable) state space of some refined model becomes enormous and then it will be infeasible to do model checking for the model. SAT-based BMC is used in industries, especially hardware industries. BMC can find a flaw located within some reasonably shallow depth k from each initial state but cannot prove that systems whose (reachable) state space is enormous (including infinite-state systems) enjoy desired properties. Some extensions have been made to SAT-based BMC so that we can prove that such systems enjoy desired properties: Craig interpolation-based model checking [16] and k-induction [17, 18]. Craig interpolation-based model checking approximately traverses the reachable state space of a system under model checking by calculating an interpolation of two logical formulas obtained from the logical formula used for SAT-based BMC. Such an interpolation can be efficiently calculated by a SAT solver. McMillan demonstrates that some model checking problems that cannot be handled with symbolic model checking can be efficiently tackled with Craig interpolation-based model checking [16]. k-induction is a combination of mathematical induction and SAT/SMT-based BMC. The bounded state space of each initial state up to depth k is tackled with BMC, which is regarded as the base case. For each state sequence s_0, s_1, ..., s_k, where s_0 is an arbitrary state, such that a property concerned is not broken in each state s_i for i = 0, 1, ..., k, it is checked that the property is not broken in all successor states s_{k+1} of s_k, which is done with a SAT/SMT solver and regarded as the induction case. If an SMT solver is used, infinite-state systems, for example, in which integers are used, could be handled. Our proposed technique can be regarded as another extension of BMC, although we do not use any SAT/SMT solvers.

Barnat et al. [19] survey some recent advancement of parallel model checking algorithms for LTL. Graph search algorithms need to be redesigned to make best use of multi-core and/or

5 http://www.jaist.ac.jp/~ogata/code/dca2l2mc/
multi-processor architectures. Parallel model checkers based on such parallel model checking algorithms have been developed, among which are DiVinE 3.0 [20] and Garakabu2 [21, 22]. We do not describe how to parallelize Algorithm 1 in the present paper, although we conduct sub-model checking experiments in the second case study in parallel by hand with several nodes in SGI Computer and it is possible to parallelize Algorithm 1. We do not need to redesign graph search algorithms so as to parallelize Algorithm 1 but can use any existing LTL model checking algorithm and model checker, which is one advantage of the approach, although Algorithm 1 is limited to properties in the form \( p \rightarrow q \) such that \( p \) and \( q \) are state propositions, though. The proposed divide & conquer approach to leads-to model checking can mitigate the challenge on time as well.

7. CONCLUSION

We have proposed a new technique to mitigate the state explosion in model checking. The technique is dedicated to leads-to properties. It divides a leads-to model checking problem into multiple smaller model checking problems and tackles each smaller one. We have proved that the multiple smaller model checking problems are equivalent to the original leads-to model checking problem. We have reported on two case studies demonstrating the power of the proposed technique.

A prototype tool [23, 24] has been first implemented in Maude and is dedicated to the case when \( L = 1 \), namely the 2 layer divide & conquer approach to leads-to model checking. We have also implemented a support tool for the \( L + 1 \) layer divide & conquer approach to leads-to model checking [25]. The tool does all necessary modifications to systems specifications (or systems models). Thus, human users do not need to change both their systems specifications and property specifications (namely LTL formulas of leads-to properties) to use the divide & conquer approach to leads-to properties. The proposed technique can be naturally parallelized because all sub-state spaces in one layer can be tackled independently and even many sub-state spaces in different layers can be. One piece of our future work is to parallelize Algorithm 1 and develop a tool that can make best use of multi-core and/or multi-processor architectures for the proposed technique. As usual, we need to conduct more case studies to make sure that the proposed technique is really useful. We may also want to make it possible to apply the proposed technique to a larger class of LTL properties.

We suppose that systems (or protocols) are specified in Maude. Maude is a rich-typed specification language, for example, making it possible to use inductively defined data structures in systems under model checking. We could translate systems specifications in Maude to those in other rich-typed specification languages, such as VDM++ [26] and SOFL [27]. If so, a support tool can be used to model check that systems specified in those specification languages enjoy leads-to properties.

A mutual exclusion protocol for ad hoc mobile networks [28] has motivated us to come up with the proposed technique. When there are only three nodes and at most three link failures could occur, the state explosion makes it impossible to conduct a leads-to model checking experiment [29]. It is also a piece of our future work to complete the leads-to model checking experiment for the ad hoc mobile network mutual exclusion protocol. Note that although we mention our preliminary idea of a divide & conquer approach to leads-to model checking in [29] without any proofs, it is flawed. The present paper proposes a divide & conquer approach to leads-to model checking together with its correctness proof.

It is crucial to find \( L \) and \( d(l) \) for \( l = 1, \ldots, L \) reasonably well so that we can effectively use the proposed technique. Good \( L \) and \( d(l) \) for \( l = 1, \ldots, L \) depend on systems or Kripke structures formalizing systems. It is also a piece of our future work to investigate how to find good \( L \) and \( d(l) \) for \( l = 1, \ldots, L \). One possible idea for it is as follows. As written, we should make a formal specification such that it does not have any long backward transitions as shown in Fig. 5 so as to effectively use the proposed technique. The two formal specifications of the two mutual exclusion protocols used in the case study reported do not have any long backward transitions and have a terminal-like state. For a formal specification that has a terminal-like state, we could calculate the depth to the terminal-like state from each initial state. We could use the depth to find out good \( L \) and \( d(l) \) for \( l = 1, \ldots, L \).

The model checking performance may depend on the number of transitions as well as the number of states in a system under model checking. The proposed technique addresses the problem caused by the number of states but does not the one caused by the number of transitions. It may be the case that there are some sub-state spaces that have a smaller number of states than the whole state space but a similar number of transitions than the whole state space. If so, the proposed technique may not efficiently tackle such a system. It is one piece of our future work how we should handle such a system.

Acknowledgments

The authors wish to thank anonymous reviewers who commented on an earlier version of the present paper. The authors were not able to complete the present paper without their useful comments.

Funding

Japan Society for the Promotion of Science (JSPS), Grant-in-Aid for Scientific Research (B) (grant no. JP19H04082).

Conflict of interest

The authors declare that they have no conflict of interest.
REFERENCES

[1] Clarke, E.M., Henzinger, T.A., Veith, H. and Bloem, R. (eds) (2018) Handbook of Model Checking. Springer, Berlin, Heidelberg.

[2] Clarke, E.M., Grumberg, O., Minea, M. and Peled, D.A. (1999) State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf., 2, 279–287.

[3] Clarke, E.M., Grumberg, O. and Long, D.E. (1994) Model checking and abstraction. ACM Trans. Program. Lang. Syst., 16, 1512–1542.

[4] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2003) Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50, 752–794.

[5] Meseguer, J., Palomino, M. and Martí-Oliet, N. (2008) Equational abstractions. Theor. Comput. Sci., 403, 239–264.

[6] Chandy, K.M. and Misra, J. (1988) Parallel Program Design: A Foundation. Addison-Wesley, Boston, MA, USA.

[7] Mellor-Crummery, J.M. and Scott, M.L. (1991) Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst., 9, 21–65.

[8] Lovász, L., Pelikán, J. and Vesztergombi, K. (2003) Discrete Mathematics: Elementary and Beyond. Undergraduate Texts in Mathematics. Springer, Berlin, Heidelberg.

[9] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. (2007) All About Maude A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. In Lecture Notes in Computer Science (LNCS) (vol. 4350). Springer, Berlin, Heidelberg.

[10] Nguyen, T.T.T. and Ogata, K. (2017) Graphical animations of state machines. In Proc. of DASC/PiCom/DataCom/Cyber-SciTech 2017, pp. 604–611. IEEE, New Jersey, Orlando, FL, USA, 6–10 November.

[11] Nguyen, T.T.T. and Ogata, K. (2017) Graphically perceiving characteristics of the MCS lock and model checking them. In Proc. of SOFL+MSVL 2017. Xi’an, China, 16 November, Lecture Notes in Computer Science (vol. 10795), pp. 3–23. Springer, Berlin, Heidelberg.

[12] Clarke, E.M., Emerson, E.A., Jha, S. and Sistla, A.P. (1998) Symmetry reductions in model checking. In Proc. of CAV 1998, Vancouver, BC, Canada, 28 June–2 July, Lecture Notes in Computer Science (vol. 1427), pp. 147–158. Springer, Berlin, Heidelberg.

[13] Bae, K., Escobar, S. and Meseguer, J. (2013) Abstract logical model checking of infinite-state systems using narrowing. In Proc. of RTA 2013, Eindhoven, The Netherlands, 24–26 June, LIPIcs, 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Wadern, Germany.

[14] Clarke, E.M., Biere, A., Raimi, R. and Zhu, Y. (2001) Bounded model checking using satisfiability solving. Form. Methods Syst. Des., 19, 7–34.

[15] Clarke, E.M., Klieber, W., Nováček, M. and Zuliani, P. (2011) Model checking and the state explosion problem. In Proc. of LASER Summer School 2011, Elba Island, Italy, 4–10 September, Lecture Notes in Computer Science (vol. 7682), pp. 1–30. Springer, Berlin, Heidelberg.

[16] McMillan, K.L. (2003) Interpolation and SAT-based model checking. In Proc. of CAV 2003, Boulder, CO, USA, 8–12 July, Lecture Notes in Computer Science (vol. 2725), pp. 1–13. Springer, Berlin, Heidelberg.

[17] Sheeran, M., Singh, S. and Stålmarck, G. (2000) Checking safety properties using induction and a SAT-solver. In Proc. of FMCAD, Austin, TX, USA, 1–3 November, Lecture Notes in Computer Science (vol. 1954), pp. 108–125. Springer, Berlin, Heidelberg.

[18] de Moura, L.M., Rueß, H. and Sorea, M. (2003) Bounded model checking and induction: From refutation to verification. In Proc. of CAV 2003, Boulder, CO, USA, 8–12 July, Lecture Notes in Computer Science (vol. 2725), pp. 14–26. Springer, Berlin, Heidelberg.

[19] Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J. and Renault, E. (2018) Parallel model checking algorithms for linear-time temporal logic. In Handbook of Parallel Constraint Reasoning, pp. 457–507. Springer, Berlin, Heidelberg.

[20] Barnat, J., Brim, L., Havel, V., Havelček, J., Kriho, J., Lenco, M., Rockai, P., Stull, V. and Weiser, J. (2013) DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In Proc. of CAV 2013, Saint Petersburg, Russia, 13–19 July, Lecture Notes in Computer Science (vol. 8044), pp. 863–868. Springer, Berlin, Heidelberg.

[21] Kong, W., Liu, L., Ando, T., Yatsu, H., Hisazumi, K. and Fukuda, A. (2015) Facilitating multicore bounded model checking with stateless explicit-state exploration. Comput. J., 58, 2824–2840.

[22] Kong, W., Hou, G., Hu, X., Ando, T., Hisazumi, K. and Fukuda, A. (2016) Garakabu2: an SMT-based bounded model checker for HSTM designs in ZIPC. J. Inf. Sec. Appl., 31, 61–74.

[23] Phyo, Y. (2019) A divide & conquer approach to leads-to model checking. In Master’s Thesis, School of Information Science, JAIST.

[24] Phyo, Y., Do, C.M. and Ogata, K. (2019) Toward development of a tool supporting a 2-layer divide & conquer approach to leads-to model checking. In Proc. of ICAIT 2019, pp. 250–255. IEEE, New Jersey, Yangon, Myanmar, 6–7 November.

[25] Phyo, Y., Do, C.M. and Ogata, K. (2020) A support tool for the L+1-layer divide & conquer approach to leads-to model checking. Submitted for publication.

[26] Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M. (2005) Validated Designs for Object-oriented Systems. Springer, London.

[27] Liu, S., Ofutt, A.J., Ho-Stuart, C., Sun, Y. and Ohba, M. (1998) SOFL: A formal engineering methodology for industrial applications. IEEE Trans. Softw. Eng., 24, 24–45.

[28] Walter, J.E., Welch, J.L. and Vaidya, N.H. (2001) A mutual exclusion algorithm for ad hoc mobile networks. Wireless Netw., 7, 585–600.

[29] Phyo, Y. and Ogata, K. (2018) Formal specification and model checking of the Walter-Welch-Vaidya mutual exclusion protocol for ad hoc mobile networks. In Proc. of APSEC 2018, pp. 89–98. IEEE, New Jersey, Nara, Japan, 4–7 December.

Section A: Computer Science Theory, Methods and Tools
The Computer Journal, 2021