IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 November 2022, accepted 17 December 2022, date of publication 19 December 2022,
date of current version 29 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3230844

RESEARCH ARTICLE

Sequential and Parallel Tools for Model Checking Conditional Stable Properties in a Layered Way

CANH MINH DO, YATI PHYO, AND KAZUHIRO OGATA
School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi, Ishikawa 923-1292, Japan
Corresponding author: Canh Minh Do (canhdominh@jaist.ac.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant JP19H04082.

ABSTRACT We invented a divide & conquer approach to conditional stable model checking so as to ease the state space explosion problem. As indicated by its name, the technique concentrates on conditional stable properties expressed as $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions. The properties can be used to formalize desired properties that self-stabilizing systems should satisfy. Self-stabilization in distributed systems was first introduced by Dijkstra and became a very crucial concept in fault tolerance to design robust systems. However, designing self-stabilizing systems need much more effort than non-stabilizing ones because the former are subject to transient errors at any time. Therefore, it is worth dedicating to conditional stable properties. In this paper, we report a sequential tool and a parallel technique/tool for the divide & conquer approach to conditional stable model checking. Some experiments are also conducted showing that our sequential and parallel tools can ease the state space explosion and improve the running performance of model checking for conditional stable properties to a certain scope, respectively.

INDEX TERMS Self-stabilizing systems, conditional stable properties, state space explosion, divide and conquer approach, parallel algorithms.

I. INTRODUCTION

The state space explosion problem is still one of the most challenges in model checking [1]. It frequently makes it impossible to carry out model checking experiments. Many techniques have been proposed to alleviate the problem, such as partial order reduction [2] and abstraction [3], [4], [5]. Although they can ease the problem to a certain scope, there exists the problem left when tackling a large number of (reachable) states. Another challenge is to increase the running performance of model checking. To address the challenge, parallel model checking algorithms and model checkers [6] have been developed so as to make the best use of multicore architectures.

Our research group came up with a divide & conquer approach to model checking leads-to properties [7] expressed as $\varphi_1 \rightsquigarrow \varphi_2$, eventual (or eventually) properties [8] expressed as $\Diamond\varphi$, and conditional stable properties [9] expressed as $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ, φ_1, and φ_2 are state propositions.

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Marozzo.

A basic idea of the approach is that the reachable state space from each initial state is split into multiple layers, generating multiple sub-state spaces, and conducting model checking experiments for each sub-state space. If the size of each sub-state space is much smaller than the one of the original reachable state space, it is feasible to conduct model checking experiments with the approach even if it is impossible to do so for the latter space thanks to the state space explosion. Our research group also built a sequential tool supporting the divide & conquer approach to model checking leads-to properties [10], a parallel version of the tool supporting a divide & conquer approach to leads-to model checking [11], and a sequential tool for eventual properties [12]. Although leads-to properties, eventual properties, and conditional stable properties can be expressed in Linear Temporal Logic (referred to as LTL) and the basic idea is used for model checking the three classes of properties, it is necessary to individually prove the correctness of each of the three divide & conquer approaches to leads-to, eventual, and conditional stable model checking, come up with each algorithm, and develop each sequential tool supporting each approach. It is also necessary to invent

three different algorithms for the three parallel versions and build the three parallel versions of the support tools for the three classes of properties. This present paper focuses on a sequential tool and a parallel technique/tool supporting a divide & conquer approach to conditional stable model checking. Note that from now on we use DCA2CSMC as the abbreviation for a divide & conquer approach to conditional stable model checking to make the paper concise.

Conditional stable properties informally say that whenever something is true, it will eventually happen that something else will be always true (or will be stable). The properties can be used to express desired properties that should be satisfied by self-stabilizing systems. As known, the term of self-stabilization in distributed systems was first introduced by Dijkstra [13] and became a very important concept in fault tolerance to design a robust system because the system is subject to transient errors at any time, such as process crashes. A system is self-stabilizing with respect to a set of legitimate states if starting from an arbitrary initial state, the system guarantees to converge to a legitimate state in a finite number of state transitions and remains in the legitimate states thereafter. A state is legitimate if starting from this state the system satisfies its desired properties. Designing self-stabilizing systems need much more effort than non-stabilizing ones because transient errors can occur at any time in a system, which often drives the system into an arbitrary state after each transient error. Therefore, it is worth dedicating to formal verification of the conditional stable properties so as to guarantee that self-stabilizing systems can reach a legitimate state from an arbitrary state after a finite number of state transitions.

DCA2CSMC has been proposed to aim to ease the state space explosion in model checking. Besides, DCA2CSMC can be naturally parallelized when multiple model checking experiments for multiple sub-spaces generated by DCA2CSMC are basically independent and those in each layer are totally independent. This paper presents a sequential tool, a parallel technique with the master-worker pattern in the form of pseudo-code, and a parallel tool for DCA2CSMC. Both the sequential and parallel tools are implemented in Maude, a high-level programming/specification language based on rewriting logic [14]. Maude has all the necessary facilities, such as meta-programming and sockets, in order to build parallel tools, such as a parallel version of a divide & conquer approach to leads-to model checking [11] and a parallel version of Maude-NPA [15]. Besides, many tools also have been developed in Maude, Maude (LTL) model checker and Spin are comparable in terms of both running time and memory consumption [16], and we use Maude as a formal specification language and its model checker. Hence, we have chosen Maude for our tool development.

Some experimental results are reported showing that the sequential and parallel tools ease the state space explosion and improve the running performance of model checking to a certain scope, respectively, for all case studies used except one that is a simple unidirectional token, namely K-state Machines (referred to as KM), compared to the straightforward use of Maude model checker. Each process used in KM has an equal chance to use a privilege to take its move, while only one process can go ahead at one time, meaning that KM has a symmetry for each process to use the privilege to take its move. Therefore, lots of states are likely to be shared by many sub-state spaces at the final layer in KM (if DCA2CSMC is used), which cannot make the best use of software caches used in the parallel tool to avoid duplicated jobs. The remaining protocols do not have such a symmetry. This would be probably why the parallel tool cannot work well for KM. Meanwhile, the sequential tool can not ease the state space explosion for KM because the size of each sub-state space at the final layer is likely to be still big, making the memory consumption high. It would be better if we could find a proper layer configuration for KM that makes the size of each sub-state space at the final layer small enough so that the sequential tool can be effectively used.

In summary, the present paper makes the following contributions:

- A sequential tool to support the divide & conquer approach to conditional stable model checking (DCA2CSMC) to ease the state space explosion in model checking.
- A parallel technique with the master-worker pattern in the form of pseudo-code and a parallel tool for DCA2CSMC to improve the running performance of model checking.
- Some case studies are conducted to demonstrate the usefulness and power of the sequential and parallel tools.
- The sequential and parallel tools as well as some case studies used in the present paper are publicly available at https://github.com/yatiphyo/DCA2MC.

The remaining part of the paper is structured as follows. Some preliminaries are mentioned in Sect. II. Sect. III and Sect. IV give an overview of the sequential technique/tool for DCA2CSMC and describe how the parallel technique/tool of DCA2CSMC works using a simple example. Sect. V and Sect. VI give an overview of the parallel technique/tool for DCA2CSMC and describe how the parallel technique/tool of DCA2CSMC works using a simple example. Experiments are reported in Sect. VII. Some related work is mentioned in Sect. VIII and the paper is finally concluded in Sect. IX.

II. PRELIMINARIES

A Kripke structure K is $\langle S, I, T, A, L\rangle$ [1]. S is a set of states, I is a set of initial states such that $I \subseteq S$, T is a left-total binary relation over S such that $T \subseteq S \times S$, A is a set of atomic propositions, and L is a labeling function whose type is $S \to 2^A$. $L(s)$ is the set of atomic propositions that hold in a given state s. $(s, s') \in T$ means that a state s directly goes (or transits) to a state s' and may be called a (state) transition. A transition (s, s') may be written as $s \to_K s'$ or $s \to s'$. An infinite sequence $s_0, \ldots, s_i, s_{i+1}, \ldots$ of states is called a path (denoted π) if $s_i \to_K s_{i+1}$ for $i = 0, \ldots, i, \ldots$. Some path notations are adopted: $\pi(i)$ is the ith state s_i (note that

the very first state of π is the 0th state s_0), π^i is a postfix $s^i, s^{i+1}, \ldots$, π_i is a path constructed by adding the ith state s_i to a prefix $s_0, \ldots, s_i$ at the end infinitely many times. We may call s_i the last state of π_i. We use $\mathcal{P}$ to denote the set of all paths. If s_0 is an initial state, we call a path computation. We use $\mathcal{C}$ to denote the set of all computations. Note that $\mathcal{C} \subseteq \mathcal{P}$ by definition. Let $\mathcal{C}$ be the set of all computations.

We use $P_{(K,s)}$ to denote the set of paths that start with $s \in S$. For a natural number b, we use $P^b_{(K,s)}$ to denote the set of all π_b such that $\pi \in P_{(K,s)}$. We use $P^\infty_{(K,s)}$ to be the same as $P_{(K,s)}$.

Let p is a state proposition in A, an LTL formula φ is defined as:

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

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

- $K, \pi \models \top$
- $K, \pi \models p$ iff $p \in L(\pi(0))$
- $K, \pi \models \neg\varphi_1$ iff $K, \pi \not\models \varphi_1$
- $K, \pi \models \varphi_1 \wedge \varphi_2$ iff $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$
- $K, \pi \models \bigcirc \varphi_1$ 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 each computation $\pi \in \mathcal{C}$ of K. $\bigcirc$ and $\mathcal{U}$ are called the next temporal connective and the until temporal connective, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond \varphi \triangleq \top \,\mathcal{U}\, \varphi$, $\Box\varphi \triangleq \neg(\Diamond \neg\varphi)$, and $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$. $\Diamond$, $\Box$, and $\rightsquigarrow$ are called the eventual (or eventually) temporal connective, the always temporal connective, and the leads-to temporal connective, respectively. LTL formulas that do not have any temporal connectives at all are called state propositions in this paper. Properties that can be expressed as $\varphi_1 \rightsquigarrow \Box\varphi_2$, where φ_1 and φ_2 are state propositions, are referred to as conditional stable properties in this paper. A simple example is used to illustrate the properties in Sect. 1 of our previous work [9].

A soup is a collection whose (non-empty) constructor is associative and commutative. A pair of name and value, such as ($pc[i]$:cs), is called an observable component, where $pc[i]$ is the name, cs is the value, and ($pc[i]$: cs) means that process i is at cs. A state is formalized as a braced soup of observable components in this paper. Transitions are written in terms of rewrite rules. Concretely, Maude [17] is used to specify systems/protocols as Kripke structures and Maude is also equipped with an LTL model checker.

III. A SEQUENTIAL VERSION OF DCA2CSMC

We have proposed a divide & conquer approach to conditional stable model checking (called DCA2CSMC for short as mentioned in Sect. I) [9]. The sequential algorithm of DCA2CSMC is shown in Algorithm 1. An infinite tree can be constructed from each initial state of K by unfolding

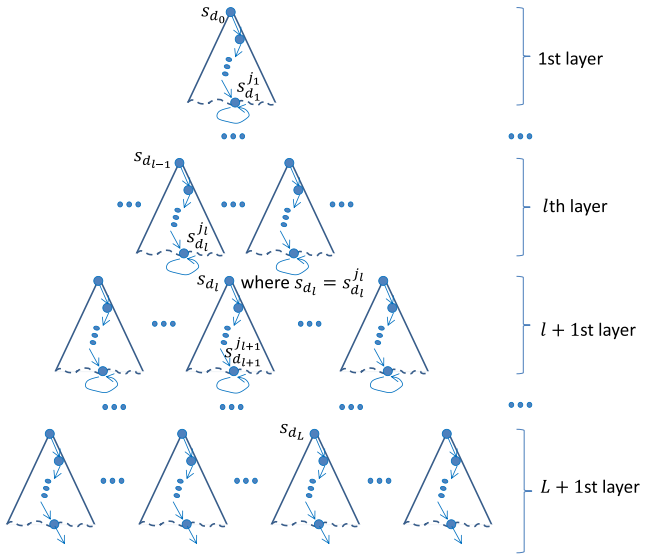

FIGURE 1. Split of the reachable state space into $L + 1$ layers.

transitions. Multiple sub-state spaces are generated by slicing such an infinite tree into layers, such as $L + 1$ layers, where $L \geq 0$, as depicted in Fig. 1. We use $d(i)$ to denote the depth of each layer i. $d(i)$ is a positive natural number if $i \leq L$, while $d(L + 1) = \infty$. Let us assume that there exists layer 0 such that $d(0) = 0$. We use d_l to equal $d(0) + \ldots + d(l)$, the depth of the bottom of layer l (or the top of layer $l + 1$) from the initial state. We call states placed at depth d_l beginning states of layer $l + 1$ (or ending states of layer l). Note that ending states of layer l are the same as beginning states of layer $l + 1$. The depth of a state means the one (at which the state is located) from the initial state. The depth of an ending state of layer l (or a beginning state of layer $l + 1$) is d_l by definition. If the depth of a state is d_l, meaning that the state is an ending state of layer l (or a beginning state of layer $l + 1$), then we let the next depth of the state mean d_{l+1}. We use $s^{j_l}_{d_l}$ to denote an ending state of layer l, while we use s_{d_l} to denote a beginning state of layer $l + 1$. Note that s_{d_l} equals $s^{j_l}_{d_l}$. A self-transition, such as $s^{j_l}_{d_l} \rightarrow s^{j_l}_{d_l}$, is attached to each ending state of each non-final layer, such as $s^{j_l}_{d_l}$. This is because transitions should be left-total for the semantics of LTL.

If s is a beginning state of layer l for $l = 1, \ldots, L$, $P^{d(l)}_{(K,s)}$ is the set of all paths in layer l that start with s, while if s is a beginning state of layer $L + 1$ (namely the final layer), $P_{(K,s)}$ is the set of all paths in layer $L + 1$ that start with s. For $\pi \in P^{d(l)}_{(K,s)}$, $\pi(d(l))$ is an ending state of layer l that repeats forever in π. For $\pi \in P^{d(l)}_{(K,s)}$, if $K, \pi \not\models \Box\neg\varphi_1$, we call $\pi(d(l))$ a `cx` state of layer l. Otherwise, we call it a `non-cx` state. `non-cx` states are stored into $NCxS$, while `cx` states are stored into CxS. For each layer, $NCxS'$ and CxS' are used to collect `non-cx` and `cx` states, respectively, and assign to $NCxS$ and CxS to update the `non-cx` and `cx` states being gathered for the current layer, respectively. For state propositions φ_1, φ_2, $K, s_{d_0} \models \varphi_1 \rightsquigarrow \Box\varphi_2$ can be verified in a layered way as described in Algorithm 1.

Algorithm 1 DCA2CSMC

input : $\boldsymbol{K}$ – a Kripke structure $s_{d_0} \in I$ – an initial state of K φ_1, φ_2 – state propositions L – a positive integer d – a function such that $d(x)$ is a positive integer for $x = 1, \ldots, L$

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

1 $\boldsymbol{NCxS} \leftarrow \boldsymbol{I}$
2 $\boldsymbol{CxS} \leftarrow \emptyset$
3 **forall the** $l \in \{1, \ldots, L\}$ **do**
4 $\quad \boldsymbol{NCxS'} \leftarrow \{\pi(d(l)) \mid s \in \boldsymbol{NCxS}, \pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}\}$
5 $\quad \boldsymbol{CxS'} \leftarrow \{\pi(d(l)) \mid s \in \boldsymbol{CxS}, \pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}\}$
6 $\quad$ **forall the** $s \in \boldsymbol{NCxS}$ **do**
7 $\quad\quad$ **forall the** $\pi \in \boldsymbol{P}^{d(l)}_{(\boldsymbol{K},s)}$ **do**
8 $\quad\quad\quad$ **if** $\boldsymbol{K}, \pi \not\models \Box\neg\varphi_1$ **then**
9 $\quad\quad\quad\quad \boldsymbol{NCxS'} \leftarrow \boldsymbol{NCxS'} - \{\pi(d(l))\}$
10 $\quad\quad\quad\quad \boldsymbol{CxS'} \leftarrow \boldsymbol{CxS'} \cup \{\pi(d(l))\}$
11 $\quad \boldsymbol{NCxS} \leftarrow \boldsymbol{NCxS'}$
12 $\quad \boldsymbol{CxS} \leftarrow \boldsymbol{CxS'}$
13 **forall the** $s \in \boldsymbol{NCxS}$ **do**
14 $\quad$ **forall the** $\pi \in \boldsymbol{P}_{(\boldsymbol{K},s)}$ **do**
15 $\quad\quad$ **if** $\boldsymbol{K}, \pi \not\models \varphi_1 \rightsquigarrow \Box\varphi_2$ **then**
16 $\quad\quad\quad$ **return** *Failure*
17 **forall the** $s \in \boldsymbol{CxS}$ **do**
18 $\quad$ **forall the** $\pi \in \boldsymbol{P}_{(\boldsymbol{K},s)}$ **do**
19 $\quad\quad$ **if** $\boldsymbol{K}, \pi \not\models \Diamond\Box\varphi_2$ **then**
20 $\quad\quad\quad$ **return** *Failure*
21 **return** *Success*

Initially, $NCxS$ and CxS are set to the set of initial states I and an empty set at lines 1–2, respectively. For each layer $l \in \{1, \ldots, L\}$ and given $NCxS$ and CxS from the previous layer, $NCxS'$ is set to the set of ending (`non-cx`) states at line 4, which are obtained from the last state in each $\pi \in P^{d(l)}_{(K,s)}$ where $s \in NCxS$, while CxS' is set to the set of ending (`cx`) states at line 5, which are obtained from the last state in each $\pi \in P^{d(l)}_{(K,s)}$ where $s \in CxS$. The code fragment at lines 6–10 checks if $K, \pi \not\models \Box\neg\varphi_1$ for each $\pi \in P^{d(l)}_{(K,s)}$ where $s \in NCxS$ in layer l. If that is the case, the last state in π is removed from $NCxS'$ and added to CxS'. When all `non-cx` and `cx` states in layer l have been gathered, $NCxS$ and CxS are updated to $NCxS'$ and CxS' at lines 11–12, respectively. The code fragment at lines 13–20 checks if $K, \pi \not\models \varphi_1 \rightsquigarrow \Box\varphi_2$ and $K, \pi \not\models \Diamond\Box\varphi_2$ for each path $\pi \in P_{(K,s)}$ in layer $L+1$ where $s \in NCxS$ and $s \in CxS$, respectively. The algorithm returns `Failure` if so and otherwise `Success`.

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

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

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

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

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

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

$\boldsymbol{NCxS_0}$ and $\boldsymbol{CxS_0}$ are initially $\{(\delta\text{-}stt, s) \mid s \in \boldsymbol{I}\}$ and $\emptyset$, respectively. A `cx` could be made, when `Failure` is returned, by searching through $\boldsymbol{NCxS_L}$, $\boldsymbol{CxS_L}$, …, $\boldsymbol{NCxS_1}$, $\boldsymbol{CxS_1}$, $\boldsymbol{NCxS_0}$, and $\boldsymbol{CxS_0}$. By this, both the sequential and parallel tools show a counterexample when $K \not\models \varphi_1 \rightsquigarrow \Box\varphi_2$.

IV. HOW THE SEQUENTIAL TECHNIQUE/TOOL WORKS

We use the first self-stabilizing, unidirectional token ring that was proposed by Dijkstra, which is called K-state Machines (KM) [13] as an example to give an overview of how the sequential technique/tool works. The ring system KM consists of N machines, numbered from 0 to $N-1$, and a parameter K, which is a natural number, such that $K > N$. Each machine status is represented by a natural number S, satisfying $0 \leq S < K$. The following notations are used for the ith machine:

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

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

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

We then define the privilege and its corresponding move for the bottom machine as follows:

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

and for the other machines as follows:

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

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

Let us specify the ring system KM in Maude. When there are n machines (processes) in KM, each state in S_{KM} is expressed as:

$$\{(\texttt{k-states}: k)\ (\texttt{pc}[0]: s_0) \ldots (\texttt{pc}[n-1]: s_{n-1})\ (\#\texttt{pc}: n)\}$$

The `#pc` observable component records the number of processes participating in KM, and the $\texttt{pc}[p_i]$ observable component stores the status s_i of the process i, which is a natural number such that $s_i < k$. The `k-states` observable component stores the natural number k. Initially, s_i is an arbitrary natural number such that $s_i < k$.

In this paper, we suppose that there are four processes participating in KM, k is 5, the statuses of four processes are 0, 2, 2 and 0, respectively. Note that $p[0]$ is the bottom process. The initial state (referred to as `init`) is as follows:

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

I_{KM} has `init` as one initial state.

T_{KM} is described in terms of rewrite rules as:

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

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

crl [fin] : {OCs} => {OCs}
if #enabled({OCs}) == 1 .
```

The names `bottom`, `other`, and `fin` are assigned to the rules in the order. The first two rewrite rules specify how to change the statuses of the bottom process and the other processes if their privileges are true, respectively, while the last rewrite rule specifies that when the system reaches a legitimate state, it just stays there and does nothing. The function `#enable` returns the number of available privileges in a given state. Note that a legitimate state has exactly one privilege. `I`, `J`, `S`, `L`, `N`, `K` are Maude variables of natural numbers and `OCs` is a Maude variable of observable component soups. `sd`, which stands for symmetric difference, takes two natural numbers x and y and returns $|x - y|$. Given a state formalized as `{(k-states: 5)(pc[0]: 0)(pc[p1]: 2)(pc[p2]: 2)(pc[p3]: 0)(#pc: 4)}`, each of the two rewrite rules `bottom` and `other` can be applied to the term expressing the state. Rewrite rule `bottom` can be applied to the term at one position and rewrite rule `other` can be applied to the term at two positions. Rewrite rule `bottom` can change it to the following: `{(k-states : 5)(pc[0]: 1) (pc[p1]: 2)(pc[p2]: 2)(pc[p3]: 0)(#pc: 4)}`. The states reachable from `init`, namely the reachable states of KM, are depicted in Fig. 2, where `(#pc: 4)` is not explicitly written just for the sake of simplicity. The number of the reachable states of KM is 17.

Two atomic propositions `illegal` and `legal` are considered in this paper. So, P_{KM} has `illegal` and `legal`. We define L_{KM} as follows:

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

where `OCs` and `PR` are Maude variables whose sorts are of observable component soups and atomic propositions. The three equations say that if there exist more than one privilege in a state, $L_{KM}(s)$ has `illegal`, if there exists solely one privilege in a state, $L_{KM}(s)$ has `legal`, and otherwise $L_{KM}(s)$ have neither `illegal` nor `legal`.

Maude model checker is used to verify $K_{KM} \models \texttt{illegal} \rightsquigarrow \Box\ \texttt{legal}$, namely that KM satisfies the conditional stable property, which can be carried out by reducing the term:

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

where `_|->_` and `[]_` are the Maude operators that express $\rightsquigarrow$ and $\Box$, respectively. Maude model checker concludes that KM satisfies the conditional stable property when there are four processes with our initial configuration.

It is unnecessary to rely on the sequential technique/tool in order to model check the conditional stable property, but we employ it to outline how the sequential technique/tool works. The reachable state space depicted in Fig. 2 is divided into three layers as depicted in Fig. 3. Fig. 3 (a), (b), and (c) exhibit the first, second, and third layers. 15 sub-state spaces are made. There are some lasso loops, but none of them is long. The greatest number of states that belong to each sub-state space is nine, while 17 is the number of states in the entire reachable state one. In summary, the number of states in each sub-state space is less than the one in the entire reachable state space; this is the core idea of DCA2CSMC to ease the state space explosion.

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

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

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

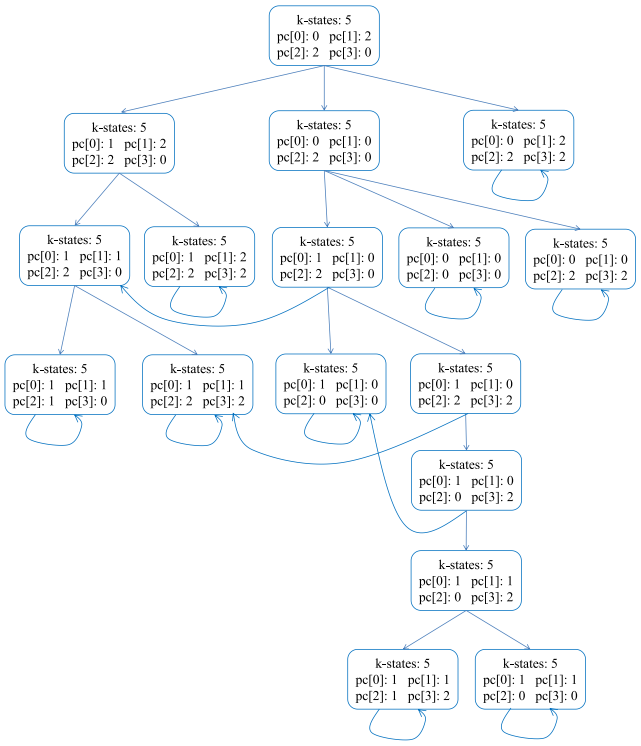

FIGURE 2. Reachable state space of KM.

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

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

crl [fin] : {(depth: D) OCs}
=> {(depth: (D + 1)) OCs}
if #enabled({OCs}) == 1 /\ D < Bound .

crl [stutter] : {(depth: D) OCs}
=> {(depth: D) OCs} if D >= Bound .
```

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

Let `init0` be the state obtained from `init` by adding the `(depth: 0)` observable component as follows:

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

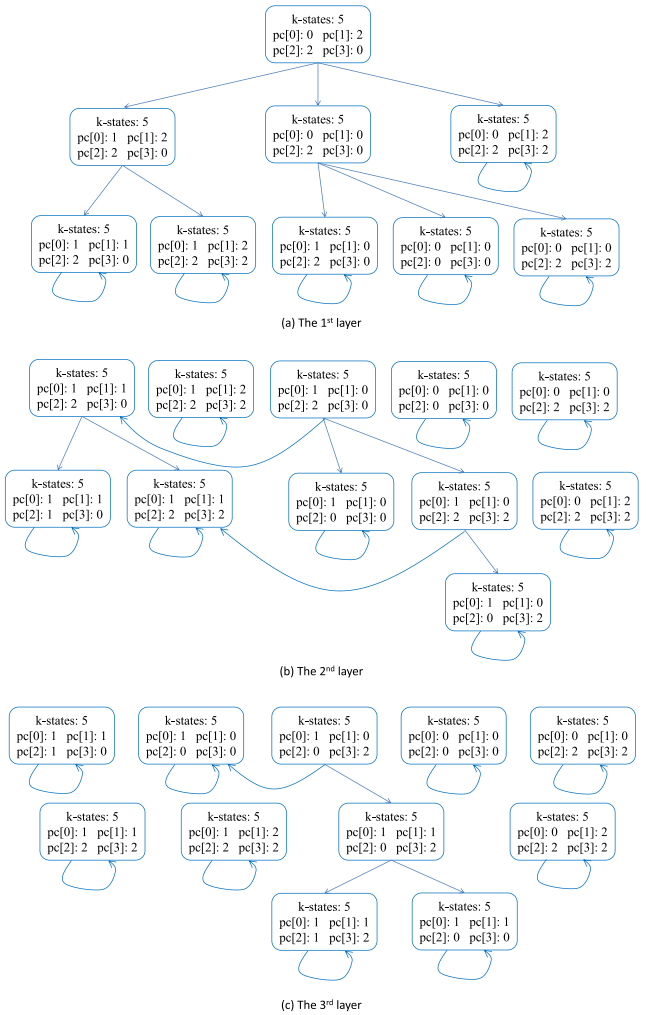

FIGURE 3. Layers 1, 2 and 3 of KM.

We are supposed to first gather all `non-cx` states and all `cx` ones placed at depth 2 from `init0`. The union of `non-cx` and `cx` states is all states placed at depth 2 reachable from `init0`. We follow Algorithm 1; if a path satisfies $\Box\neg$*illegal*, the last state (that has the self-transition) of the path is regarded as a `non-cx` state; otherwise, the last state is a `cx` state. A `non-cx` state can become a `cx` state if it is found later as a `cx` state. There are only six `cx` states in the first layer (see Fig. 3 (a)):

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

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

Fig. 3 (b) shows eight states positioned at depth 4 (from `init0`) in the second layer. Two states are positioned at depth 4 reachable from `init4`. Four states are positioned at depth 4 reachable from `init6`, where two states of them are also reachable from `init4`. One state is positioned at depth 4 reachable from each `init5`, `init7`, `init8`, and `init9`. The eight states are denoted `init12`, `init13`, `init18`, `init19`, `init5'`, `init7'`, `init8'`, and `init9'`. Note that `init5'`, `init7'`, `init8'`, and `init9'` are the same as `init5`, `init7`, `init8`, and `init9` except the `depth` observable component. Hence, there are actually four new states `init12`, `init13`, `init18`, and `init19` as follows:

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

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

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

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

We use Maude to build a sequential tool for DCA2CSMC. Given the initial KM specification in which the conditional stable property and the layer configuration (a positive natural number list) are written, the sequential tool automates the model checking experiment described above and returns `Success`. We intend to insert the following rule into the formal specification of KM in order to demonstrate what is shown by the sequential tool when a counterexample is discovered:

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

The following is shown:

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

Once we get to the following state:

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

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

V. A PARALLEL VERSION OF DCA2CSMC

We use Maude to build a parallel version of DCA2CSMC. Maude supports object-oriented systems, where objects exchange messages to communicate with each other. Sockets can be used in Maude to make it possible for objects residing in a Maude instance (running as an OS process) to communicate with external objects residing in another Maude instance (running as another OS process). Object-oriented systems and sockets are used to develop the parallel tool.

$K \models \varphi_1 \rightsquigarrow \Box\varphi_2$ can be verified with DCA2CSMC [9] in a layered way. DCA2CSMC categorizes the states at the bottom of each non-final layer into `non-cx` states and `cx` ones by verifying $\Box\neg\varphi_1$ for the sub-state spaces in the layer. Meanwhile, it verifies $\varphi_1 \rightsquigarrow \Box\varphi_2$ or $\Diamond\Box\varphi_2$ for the sub-state spaces in the final layer.

Each model checking problem for each sub-state space is encapsulated as a job. A job has a state located at the beginning of a layer that is considered the initial state of the model checking problem. If the initial state of a job is a `non-cx` one, the type of the job is `ncx`; if it is a `cx` one, the type of the job is `cx`. Each real initial state of a given Kripke structure K is a `non-cx` one and then layer 1 only has `ncx` jobs. Any other layers may have both `ncx` and `cx` jobs. Once a state s located at the beginning of layer l $(= 2, \ldots, L)$ is a `cx` one, any states located at layer l' $(= 3, \ldots, L+1)$ and reachable from s is also a `cx` one. Therefore, for a `cx` job of any non-final layers, it is unnecessary to conduct any model checking experiments. For the final layer, $\varphi_1 \rightsquigarrow \Box\varphi_2$ is verified for each `ncx` job, while $\Diamond\Box\varphi_2$ is verified for each `cx` job. If a counterexample is found for the final layer, $K \models \varphi_1 \rightsquigarrow \Box\varphi_2$ does not hold; otherwise, $K \models \varphi_1 \rightsquigarrow \Box\varphi_2$ holds. To construct a (global) counterexample when $K \models \varphi_1 \rightsquigarrow \Box\varphi_2$ does not hold, both the sequential and parallel tools manage a log list for each state s_{d_l} located at the beginning of each layer $l+1$. Such a list is in the form $< s_{d_{l-1}} : d_l > \ldots < s_{d_1} : d_2 >< s_{d_0} : d_1 >$,

a list of pairs of natural numbers and states. Each element $< s_{d_{i-1}} : d_i >$ consists of a state located at the beginning of layer i and the depth of layer i. From s_{d_l} (and d_l) and the log list, we can construct a finite computation from s_{d_0} to s_{d_l}. When layer l is the final one and a (local) counterexample is found for a job whose initial state is s_{d_l}, the global counterexample can be constructed from the finite computation and the local counterexample. For each real initial state s_{d_0} of K, the log list is nil.

Because model checking experiments are independent for sub-state spaces in each layer, they can be carried out in parallel. A master-worker pattern is used to build our parallel tool. It is necessary to take load balancing of tasks tackled by workers and communication overhead between the master and workers into account. To this end, we use two types of caches: one shared cache and multiple local caches. The shared cache is utilized by the master so as to prevent the same jobs from being delivered to workers. Each worker utilizes one local cache in order to prevent the same jobs from being created and supplied to the master. This is how the communication overhead between the master and workers can be reduced. Map data structures are used to implement caches. The master creates the very initial job, and workers create all the other jobs and supply them to the master. Two queues are utilized by the tool. One (named `jobs`) is a queue of jobs, while the other (named `workers`) is one of worker identifiers. Whenever both `jobs` and `workers` are not empty, the top extracted `jobs` is delivered to the top extracted from `workers` by the master. This way of delivering jobs to workers contributes to the load balancing of tasks tackled by workers.

There are three types of messages used in the tool: *job*, *getJob*, and *stop*. A *job* message is a five tuple `< s, jtype, d, d', log >`. `s` is a beginning state of a layer. `jtype` is either `cx` or `non-cx`. `d` is the depth of `s`. `d'` is the next depth of `s`. `log` is a log list of `s`. The master extracts the top job and the top worker identifier from `jobs` and `workers`, respectively, constructs a *job* message from the job, and sends the worker the message in order to deliver the job to the worker. Workers also send the master *job* messages in order to supply jobs constructed by the workers to the master. Workers send the master *getJob* messages to ask the master to deliver jobs to the workers. As a worker finds a local counterexample in the final layer, it sends the master a *stop* message so as to notify the master of it and displays the global counterexample. When the *stop* message arrives at the master, the master closes all connections to the workers, displays `Failure`, and terminates the tool. A *stop* message has no parameters, and neither does a *getJob* message.

Each worker is responsible for processing jobs assigned by the master, creating new jobs, and supplying the new jobs to the master. We suppose that a job `< s, jtype, d, d', log >` has been delivered to a worker. If `s` is a beginning state of a non-final layer and `jtype` is `cx`, the worker just gathers all ending states of the layer that are reachable from `s`. All the ending states are `cx` states. If `s` is a beginning state of a non-final layer and `jtype` is `non-cx`, the worker gathers all ending states of the layer that are reachable from `s` by carrying out a model checking experiment as described. Some of the ending states are `non-cx` states, while the remaining states are `cx` states. For each of such ending states gathered of the layer, the worker creates a new job, uses/updates its local cache to check whether the jobs created have been already processed by the worker, and only sends the unprocessed jobs to the master as *job* messages. If `s` is

Algorithm 2 Delivering Jobs to Workers by a Master

input : K – a Kripke structure
$s_{d_0} \in I$ – an initial state of K
φ_1, φ_2 – state propositions
$d_1 \ldots d_L$ – a list of positive integers, where L is a positive integer
$d_0 = 0, d_{L+1} = \infty$
N – a number of workers
output: Success (K, $s_{d_0} \models \varphi_1 \rightsquigarrow \Box\varphi_2$) or
Failure (K, $s_{d_0} \not\models \varphi_1 \rightsquigarrow \Box\varphi_2$)

1 $NcxStates \leftarrow empty$; $CxStates \leftarrow empty$;
2 $next \leftarrow empty$; $jobs \leftarrow empty$; $workers \leftarrow empty$;
3 $JOB \leftarrow (s_{d_0}, ncx, d_0, d_1, nil)$;
4 $enq(jobs, JOB)$;
5 $NcxStates[d_0] \leftarrow NcxStates[d_0] \cup s_{d_0}$;
6 **while** *True* **do**
7 **for** $k \leftarrow 1$ **to** N **do**
8 **if** $MSG \leftarrow rec(worker_k)$ **then**
9 **if** $MSG = getJob$ **then**
10 $enq(workers, worker_k)$
11 **else if** $MSG = stop$ **then**
12 closeConnect();
13 **return** *Failure*;
14 **else**
15 $(s_{d_l}, type, d_l, d_{l+1}, log) \leftarrow MSG$;
16 **if** $type = ncx \wedge s_{d_l} \notin NcxStates[d_l]$ **then**
17 $enq(next, MSG)$;
18 $NcxStates[d_l] \leftarrow NcxStates[d_l] \cup s_{d_l}$;
19 **if** $type = cx \wedge s_{d_l} \notin CxStates[d_l]$ **then**
20 $enq(next, MSG)$;
21 $CxStates[d_l] \leftarrow CxStates[d_l] \cup s_{d_l}$;
22 **while** $\neg\, isEmpty(workers) \wedge \neg\, isEmpty(jobs)$ **do**
23 $worker \leftarrow deq(workers)$;
24 $job \leftarrow deq(jobs)$;
25 snd(worker, job);
26 **if** $size(workers) = N \wedge isEmpty(jobs)$ **then**
27 $jobs \leftarrow filterJobs(next)$;
28 $next \leftarrow empty$;
29 **if** $isEmpty(jobs) \wedge isEmpty(next) \wedge size(workers) = N$ **then**
30 closeConnect();
31 **return** *Success*;

Algorithm 3 Processing Jobs With Workers

input : K – a Kripke structure
φ_1, φ_2 – state propositions
$d_1 \ldots d_L$ – a list of positive integers, where L is a positive integer
$d_0 = 0, d_{L+1} = \infty$
output: a counterexample if any

```
1  NcxStates ← empty; CxStates ← empty;
2  snd(server, getJob);
3  while isOpen() do
4      if MSG ← rec(server) then
5          (s_{d_l}, type, d_l, d_{l+1}, log) ← MSG;
6          if type = ncx then
7              if d_{l+1} ≠ ∞ then
8                  forall the π ∈ P^{d_{l+1}}_{(K,s_{d_l})} do
9                      s_{d_{l+1}} ← π(d_{l+1});
10                     if K, π ⊭ □¬φ_1 then
11                         if s_{d_{l+1}} ∉ CxStates[d_{l+1}] then
12                             JOB ← (s_{d_{l+1}}, cx, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log);
13                             snd(server, JOB);
14                             CxStates[d_{l+1}] ← CxStates[d_{l+1}] ∪ s_{d_{l+1}};
15                     else
16                         if s_{d_{l+1}} ∉ NcxStates[d_{l+1}] then
17                             JOB ← (s_{d_{l+1}}, ncx, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log);
18                             snd(server, JOB);
19                             NcxStates[d_{l+1}] ← NcxStates[d_{l+1}] ∪ s_{d_{l+1}};
20             else
21                 forall the π ∈ P_{(K,s_{d_l})} do
22                     if K, π ⊭ φ_1 ⇝ □φ_2 then
23                         snd(server, stop);
24                         return buildCx();
25         if type = cx then
26             if d_{l+1} ≠ ∞ then
27                 forall the π ∈ P^{d_{l+1}}_{(K,s_{d_l})} do
28                     s_{d_{l+1}} ← π(d_{l+1});
29                     if s_{d_{l+1}} ∉ CxStates[d_{l+1}] then
30                         JOB ← (s_{d_{l+1}}, cx, d_{l+1}, d_{l+2}, < s_{d_l} : d_{l+1} > log);
31                         snd(server, JOB);
32                         CxStates[d_{l+1}] ← CxStates[d_{l+1}] ∪ s_{d_{l+1}};
33             else
34                 forall the π ∈ P_{(K,s_{d_l})} do
35                     if K, π ⊭ ◇□φ_2 then
36                         snd(server, stop);
37                         return buildCx();
38         snd(server, getJob);
```

a beginning state of the final layer, the worker carries out a model checking experiment as described. Whenever the worker finds out a local counterexample in the final layer for the model checking experiment, it sends the master a *stop* message, constructs a global counterexample, and displays it. Whenever a worker becomes idle, it sends the master a *getJob* message in order to ask the master to deliver a new job to it if any. On the other hand, the master is mostly responsible for delivering unprocessed jobs to workers. After the master has distributed all jobs in each layer to workers with the two queues, it waits until all jobs are to be processed by workers. Meanwhile, the master receives *job* messages sent by workers and temporarily stores the jobs in another queue `next`. When all jobs have been processed by workers,

the master uses/updates the shared cache to check the jobs in `next` have been already processed and only saves the unprocessed jobs in `jobs`. Whenever the master receives a *stop* message from a worker, it closes all connections to all workers, displays some information, such as `Failure`, and terminates the tool. Whenever the master checks all model checking problems in the final layer without finding any counterexample, it closes all connections to all workers, displays some information, such as `Success`, and terminates the tool.

The pseudo-code of job scheduling carried out by the master is shown as Algorithm 2. The shared cache is implemented by two map data structures: `NcxStates` and `CxStates`. This is because there are two types of jobs: `ncx` and `cx`. `NcxStates` and `CxStates` are used for `ncx` and `cx` jobs, respectively. A natural number (the depth of a state) is utilized as a key, while a set of states is utilized as a value. For example, for a natural number (or a depth) d, `NcxStates[`d`]` is a set of `non-cx` states positioned at depth d. Code fragment 1 – 5 is the initialization part. In code fragment 7 – 21, the master receives a message from each worker and checks what type of message it is. The master carries out what it is supposed to do depending on the type as described. In code fragment 22 – 25, the master checks if neither workers nor jobs is empty and delivers a job to a worker if so as described. In code fragment 26 – 28, the master checks if all jobs in the layer to be tackled currently have been processed and carries out what it is supposed to do if so as described. Function *filterJobs* uses the shared cache to delete the jobs that have been processed from `next`. In code fragment 29 – 31, the master checks if all jobs in the layer (namely all layers) have been processed and closes all connections to the workers, returning `Failure` and terminating the tool, if so.

The pseudo-code of job handling carried out by each worker is shown as Algorithm 3. Its local cache is implemented by two map data structures `NcxStates` and `CxStates` as the shared cache. Code fragment 1 – 2 is the initialization part. Function *isOpen* returns *true* if the connection between the master and the worker is open; it returns *false* if the connection is closed. If the connection is closed, the worker terminates. The worker receives a *job* message from the master at line 4. If the type of the job is `ncx`, the worker carries out what it is supposed to do in code fragment 7 – 24 as described. Note that only the depth of the final layer is ∞. If the state encapsulated in the job is in a non-final layer, the worker follows code fragment 8 – 19 as described. If the state encapsulated in the job is in the final layer, the worker follows code fragment 21 – 24 as described. Function *buildCx* constructs a global counterexample as described. If `cx` is the job type, the worker carries out what it is supposed to do in code fragment 26 – 37 as described. If the state inside the job is in a non-final layer, the worker follows code fragment 27 – 32 as described. Otherwise, the state is in the final layer, the worker follows code fragment 34 – 37 as described. Whenever the worker becomes idle, a *getJob* message is sent to the master by the worker at line 38.

VI. HOW THE PARALLEL TECHNIQUE/TOOL WORKS

We use KM with the same layer configuration as in Sect. IV, where only four processes participate in KM, to outline how the parallel technique/tool works. Readers can visit again the reachable state space of KM in Fig. 2 and the three layers divided in Fig. 3 (a), (b), and (c) while reading this section. The labels of states `init0`, `init4`, `init5`, `init6`, `init7`, `init8`, `init9`, `init12`, `init13`, `init18`, `init19`, `init5'`, `init7'`, `init8'`, and `init9'` are also preserved to denote their actual states as in Sect. IV. For the sake of simplicity, we use only one worker and a master in this example.

Our parallel tool can be utilized as a usual existing model checker. A user is only supposed to provide two formal specifications to the tool: one is a formal systems specification and the other one is a formal property specification under verification. A formal systems specification needs to be revised as described, which is also automated by the parallel tool and a user does not need to care about it. First, several variables, such as `workers`, `jobs`, and `next`, are initialized and the first job is created by the master. The job (called `job0`) is `(init0, ncx, 0, 2, empty)`. `job0` is put into `jobs`. `init0`, together with its key 2, is registered into `NcxStates` of the shared cache. Some values managed by the master are as follows:

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

The worker sends the master a *getJob* message. On receipt of the message, the master puts the worker identifier into `workers`, extracting the top job and the top worker identifier from `jobs` and `workers`, respectively, and sending the job as a job message to the worker, where the job is `job0` and the worker is the one worker. The worker handles `job0`, generating all `non-cx` states and all `cx` states placed at depth 2 reachable from `init0`. There are six states denoted `init4`, `init5`, `init6`, `init7`, `init8`, and `init9` (see Fig. 3 (a)). All of them are `cx` states. The job `(init4, cx, 2, 4, < init0 : 2 >)` is made by the worker. The job is named `job1`. The worker then updates `CxStates` of its local cache. `job1` is sent to the master by the worker as a *job* message. When the message arrives at the master, the master uses `CxStates` of the shared cache to check if `job1` has been processed. Because `job1` has not, the master updates `CxStates`. For the other jobs made by the worker for `init5`, `init6`, `init7`, `init8`, and `init9`, similar things are carried out. The values managed by the master are as follows:

```
NcxStates = 0 |-> {init0}
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9}
workers   = empty
jobs      = empty
next      = job4 | job5 | job6 | job7 |
            job8 | job9
```

where job4, job5, job6, job7, job8, and job9 are as follows:

```
(init4, cx, 2, 4, <init0 : 2>)
(init5, cx, 2, 4, <init0 : 2>)
(init6, cx, 2, 4, <init0 : 2>)
(init7, cx, 2, 4, <init0 : 2>)
(init8, cx, 2, 4, <init0 : 2>)
(init9, cx, 2, 4, <init0 : 2>)
```

On the other hand, some values managed by the worker at this moment are as follows:

```
NcxStates = empty
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9}
```

The worker completes job0 and sends the master a *getJob* message for a new job being delivered to it. On receipt of the message, the master enqueues the worker identifier into workers. At this moment, the worker has completed generating all possible jobs for the next layer; jobs is empty and next contains all jobs generated from the worker. We check that there is no unnecessary job in next, and so we assign next to jobs and then assign empty to next for the next layer (the second layer).

job4 is the top of jobs and delivered to the worker next. On receipt of the job, the worker handles it for layer 2. Because init4 is a cx state, it is enough to generate all states located at depth 4 reachable from init4. There are two states denoted init12 and init13 (see Fig. 3 (b)), which are two cx states. The worker then constructs jobs to send to the master. When job4 is completed by the worker, some values managed by the master at the moment are as follows:

```
NcxStates = 0 |-> {init0}
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13}
workers   = empty
jobs      = job5 | job6 | job7 | job8 | job9
next      = job12 | job13
```

where job12 and job13 are as follows:

```
(init12, cx, 4, unbounded, log4)
(init13, cx, 4, unbounded, log4)
```

where log4 is <init4 : 4> <init0 : 2>. The values managed by the worker at the moment are as follows:

```
NcxStates = empty
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13}
```

The worker sends the master a *getJob* message for a new job being delivered to it. Because the top of jobs is job5, job5 is delivered to the worker. On receipt of the job, the worker then handles job5 for layer 2. One state is positioned at depth 4 reachable from init5 denoted init5′ (see Fig. 3 (b)). When job5 has been processed by the worker, some values managed by the master at the moment are as follows:

```
NcxStates = 0 |-> {init0}
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5'}
workers   = empty
jobs      = job6 | job7 | job8 | job9
next      = job12 | job13 | job5'
```

where job5′ is as follows:

```
(init5', cx, 4, unbounded, <init5 : 4>
 <init0 : 2>)
```

Some values managed by the worker at the moment are as follows:

```
NcxStates = empty
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5'}
```

The worker sends the master a *getJob* message for a new job being delivered to it. The top of jobs is job6 that is delivered to the worker. On receipt of the job, the worker handles job6 for layer 2. There are four states located at depth 4 reachable from init6 denoted init12, init13, init18, and init19 (see Fig. 3 (b)). Because CxStates of the local cache contains init12 and init13 together with key 4, jobs are not created for the two states. The worker creates job18 and job19 for init18 and init19 as follows:

```
(init18, cx, 4, unbounded, log6)
(init19, cx, 4, unbounded, log6)
```

where log6 is <init6 : 4> <init0: 2>. The worker sends the master job18 and job19 as *job* messages. When the worker has processed job6, some values managed by the master at the moment are as follows:

```
NcxStates = 0 |-> {init0}
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5',
                   init18, init19}
workers   = empty
jobs      = job7 | job8 | job9
next      = job12 | job13 | job5' | job18 |
            job19
```

and some values managed by the worker at the moment are as follows:

```
NcxStates = empty
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5',
                   init18, init19}
```

The worker sends the master a *getJob* message for a new job being delivered to it. The worker handles job7, job8, and job9 as it has handled job5. One state is placed at depth 4 reachable from each init7, init8, and init9. The three states are denoted init7′, init8′,

and `init9'`. When `job7`, `job8`, and `job9` have been processed by the worker, some values managed by the master at the moment are as follows:

```
NcxStates = 0 |-> {init0}
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5',
                   init18, init19, init7',
                   init8', init9'}
workers   = empty
jobs      = empty
next      = job12 | job13 | job5' | job18 |
            job19 | job7' | job8' | job9'
```

where `job7'`, `job8'`, and `job9'` are as follows:

```
(init7', cx, 4, unbounded, <init7 : 4>
 <init0 : 2>)
(init8', cx, 4, unbounded, <init8 : 4>
 <init0 : 2>)
(init9', cx, 4, unbounded, <init9 : 4>
 <init0 : 2>)
```

Some values managed by the worker at the moment are as follows:

```
NcxStates = empty
CxStates  = 2 |-> {init4, init5, init6,
                   init7, init8, init9},
            4 |-> {init12, init13, init5',
                   init18, init19, init7',
                   init8', init9'}
```

When the jobs have been processed by the worker, a *getJob* message is sent to the master by it for a new job being delivered to it. On receipt of the message, the master enqueues the worker identifier into `workers`. The worker has processed all jobs for layer 2 at the moment, when `jobs` is empty and `next` has all jobs created by the worker. Because no job has been processed in `next`, the master sets `jobs` to `next` and `next` to empty for the final layer.

There are eight jobs left in `jobs` at the moment denoted `job12`, `job13`, `job5'`, `job18`, `job19`, `job7'`, `job8'`, and `job9'`. Because the top of `jobs` is `job12`, the master delivers `job12` to the worker. On receipt of `job12`, the worker handles `job12` for the final layer, model checking $\Diamond\Box legal$ instead of $illegal \rightsquigarrow \Box legal$ for the sub-state space from `init12` encapsulated in `job12` because the type of `job12` is `cx`. The worker does not find any counterexample, completes `job2`, and sends the master a *getJob* message for a new job being delivered to it. The master delivers the worker the remaining jobs. On receipt of the jobs, the worker handles the jobs as it has done for `job12`. The worker does not find any counterexample, completes the jobs, and then sends the master a *getJob* message for a new job being delivered to it. Both `jobs` and `next` are empty. The master then closes the connection to the worker, returns `Success`, and terminates the tool. In summary, KM enjoys $illegal \rightsquigarrow \Box legal$ when there are four processes with our initial configuration.

VII. EXPERIMENTS

We conduct model checking experiments with (1) Maude model checker, (2) the sequential tool, and (3) the parallel tool of DCA2CSMC. Maude model checker uses the same model checking algorithm (the explicit-state on-the-fly LTL model checking algorithm) as SPIN [18], which is one of the most popular model checkers for model checking software systems. It has been reported that Maude model checker and SPIN are comparable in terms of both running time and memory consumption [19]. This implies that whenever Maude model checker encounters the state space explosion problem, making it impossible to conduct model checking experiments, so do SPIN and most existing model checkers. Therefore, it is meaningful to compare our sequential and parallel tools with Maude model checker.

We use two mutual exclusion protocols and the KM protocol as systems under model checking: Qlock, Anderson, and KM. Qlock is an abstract version of the Dijkstra binary semaphore. Anderson is an array-based mutual exclusion protocol invented by Anderson [20]. We assume that each process goes to the critical section at most once. For both Qlock and Anderson, we revise their specifications so that they become self-stabilizing systems. We add an observable component `abnorm` that stores a Boolean value that denotes the current state is abnormal or not. `abnorm` is set to *true* whenever we detect that there are at least two processes located at the critical section. `abnorm` can be set back to *false* if the state has been recovered in which there is no process located at the critical section detected.

Qlock for each process p can be described as follows:

$$
\begin{aligned}
&\text{“Start Section”}\\
ss &: enq(qu, p);\\
ws &: repeat\ until\ top(qu) = p;\\
&\text{“Critical Section”}\\
cs &: deq(qu);\\
&\text{“Final Section”}\\
fs &: \ldots
\end{aligned}
$$

qu is an atomic queue of process IDs. All processes taking part in the protocol share qu. enq, top, and deq are atomic operations for atomic queues. qu is initially empty and each process p is initially at ss (Start Section). Whenever p would like to go to cs (Critical Section), it puts its ID into qu at the end with enq and goes to ws (Waiting Section). p waits at ws while $top(qu)$ is not p. Whenever $top(qu)$ becomes p, p goes to cs. When p exits cs, it deletes the top (namely the p's ID) from qu with deq and goes to fs (Final Section). If `abnorm` is *false*, process p works as abovementioned. If `abnorm` is *true*, when process p would like to exit cs, it gets rid of all elements from qu and goes to fs. For Qlock experiments, the initial state is set to an illegitimate state in which processes $p2$, $p3$, and $p5$ are located at cs, qu contains only $p3$, the other processes are located at ss, and `abnorm` is *false*.

TABLE 1. Conditional stable model checking running performance by Maude model checker and the sequential tool with 2GB of memory.

Protocol	#Processes	Maude model checker	Layers	DCA2CSMC
Qlock	10 processes	23s	2 2	1m 3s
	11 processes	N/A		15m 31s
Anderson	5 processes	10s	2 2	44s
	6 processes	N/A		38m 8s

TABLE 2. Conditional stable model checking running performance by Maude model checker and the sequential tool for KM with 1GB of memory.

Protocol	#Processes	Maude model checker	Layers	DCA2CSMC
KM	10 processes	36m	2 2	1d 4h 58m
	11 processes	1h 9m		2d 23h 54m
	12 processes	7h 58m		22d 14h 1m
	13 processes	N/A		N/A

Anderson for each process p can be described as follows:

"Start Section"
ss : $pos[p] := fetch\&inc\%(nxt, M)$;
ws : *repeat until* $a[pos[p]]$;
"Critical Section"
cs : $a[pos[i]], a[(pos[i]+1)\%M] := false, true$;
"Final Section"
fs : ...

where M is the number of processes taking part in Anderson. nxt is a variable of natural numbers and a is an array such that its size is M and the type of each element is Bool. nxt and a are shared with the M processes. $pos[p]$ is a variable of natural numbers and local to process p. $fetch\&inc\%$ is an atomic operation. $fetch\&inc\%(nxt, M)$ increments nxt modulo M and returns the old value of nxt. $x_1, x_2 := e_1, e_2$; is a concurrent assignment. Expressions e_1 and e_2 are evaluated simultaneously (or independently), and their results are assigned to variables x_1 and x_2, respectively. Each process p is initially at ss (Start Section) and the initial value of each variable is as: $nxt = 0$, $a[0] = true$, $a[j] = false$ for $j = 1, \ldots, M-1$, and $pos[p] = 0$. Whenever process p would like to go to cs (Critical Section), it atomically sets $pos[p]$ to nxt and increments nxt modulo M by $fetch\&inc\%$, moving to ws (Waiting Section). Process p waits at ws while $a[pop[p]]$ is $false$. Whenever $a[pop[p]]$ becomes $true$, p goes to cs. When it exits cs, it assigns $false$ and $true$ to $a[pos[p]]$ and $a[(pos[p]+1)\%M]$, respectively, moving to fs (Final Section). When `abnorm` is $false$, process p works as abovementioned. If `abnorm` is $true$ and process p wants to go to or exit cs, it goes to fs instead. For Anderson experiments, the initial state is set to an illegitimate state in which nxt is 0, `abnorm` is $false$, and for each process p, $pc[p]$, $pos[p]$, and $a[p]$ are set to ss, 0, and $true$, respectively.

We take three atomic propositions `inCs1`, `inCs5`, and `inAbnorm` that denote whether processes $p1$ and $p5$ locate at cs or not, and whether the current state is an abnormal state or not, respectively. We model checked `inAbnorm` $\rightsquigarrow$ $\Box\neg$ `(inCs1` $\wedge$ `inCs5)` for Qlock and Anderson while we model checked `illegal` $\rightsquigarrow$ $\Box$ `legal` for KM with the following initial state:

```
{(k-states: 11) (pc[0]: 0) (pc[1]: 2)
 (pc[2]: 2) (pc[3]: 3) (pc[4]: 4) (pc[5]: 5)
 (pc[6]: 6) (pc[7]: 7) (pc[8]: 8) (pc[9]: 0)
 (#pc: 10)}
```

using Maude model checker, the sequential and parallel tools of DCA2CSMC. The size and complexity of each of the three case studies are directly proportional to the number of processes used in each one.

A. EXPERIMENTS WITH MAUDE MODEL CHECKER AND THE SEQUENTIAL TOOL

We conducted case studies with Maude model checker and the sequential tool using a docker container running Ubuntu 20.04.3 LTS as a virtual machine that ran on a host machine (an iMac) with a 4 GHz processor and 32 GB of memory. For Qlock and Anderson experiments, we restricted to use 2 GB of memory for the virtual machine. The experimental data are exhibited in Table 1. $d_1\ d_2\ \ldots\ d_L$ in the *layers* column says that $L+1$ layers are employed and ith layer depth is d_i. `N/A` means that the model checking experiment made it impossible to be carried out in that it did not suffice to employ 2 GB of memory for the model checking. Thus, the state space explosion can be eased by the sequential tool to a certain scope.

For KM experiments, we restricted to use only 1 GB of memory for the virtual machine. The experimental data are shown in Table 2. For KM with 10 processes, 11 processes and 12 processes, both the sequential tool and Maude model checker could complete the model checking experiments. For KM with 13 processes, both the sequential tool and Maude model checker could not complete the model checking experiments because 1 GB of memory was not sufficient for model checking experiments, leading to the state space explosion. We can see that the verification time of Maude model checker is much smaller than that of the sequential tool. That is

TABLE 3. Conditional stable model checking running performance by Maude model checker, the sequential tool, and the parallel tool.

Protocol	Maude model checker	Layers	DCA2CSMC	Parallel DCA2CSMC			
				4 workers	8 workers	12 workers	16 workers
Qlock (11 processes)	39m 31s	2 2	17m 55s	5m	3m 27s	2m 47s	2m 23s
Anderson (6 processes)	6h 44m 33s	2 2	40m 9s	10m 46s	5m 49s	3m 59s	3m 15s
KM (10 processes)	58m 48s	2 2	2d 13h 42m 7s	19h 27m 22s	10h 17m 56s	7h 17m 11s	5h 44m 54s

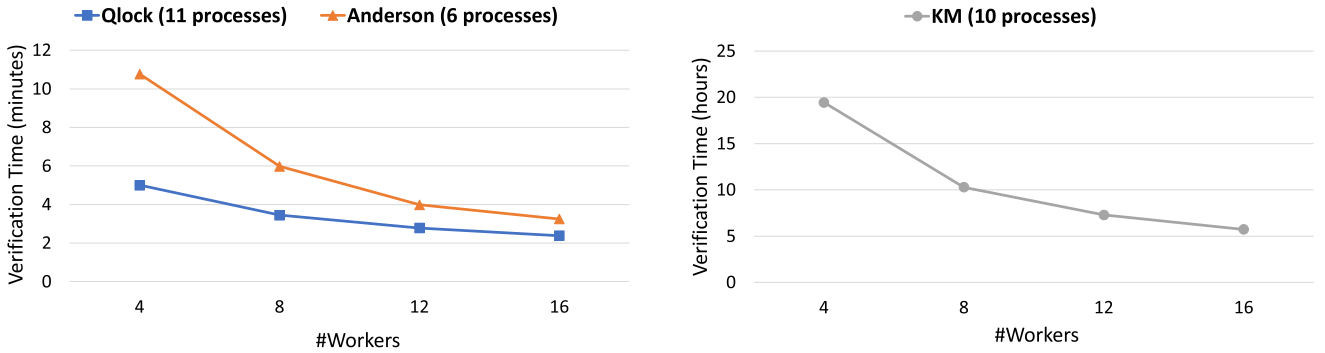

FIGURE 4. Verification time with different numbers of workers.

because many states are likely to be shared by the sub-state spaces at the final layer. Therefore, the sequential tool may need to explore again many shared states in those sub-state spaces at the final layer, making the running performance of the sequential tool degrade. For KM with 13 processes, the sequential tool also could not ease the state space explosion because the size of each sub-state space at the final layer is likely to be still big, making the memory consumption high. We need to find a better layer configuration for KM with 13 processes in which the size of each sub-state space at the final layer is small enough. This is one piece of our future work.

B. EXPERIMENTS WITH MAUDE MODEL CHECKER, THE SEQUENTIAL TOOL, AND THE PARALLEL TOOL

We used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB of memory to conduct experiments with (1) Maude model checker, (2) the sequential tool, and (3) the parallel version of DCA2CSMC. We have demonstrated above that the sequential tool can ease the state space explosion in model checking to a certain scope with limited memory. However, we do not restrict the memory used in these experiments to demonstrate that the parallel tool can increase the running performance of model checking to a certain scope. The experimental data are shown in Table 3.

We conducted experiments for Qlock with 11 process participants, Anderson with 6 process participants, and KM with 10 process participants. Our sequential version of DCA2CSMC exhibits better model checking running performance than Maude model checker for the two mutual exclusion protocols as shown in Table 3. Our parallel version of DCA2CSMC is better than the sequential version from model checking running performance for the two protocols as shown in Table 3. The model checking running performance achieved by the former is about 3.6 and 3.7 times faster than the one by the latter for the two protocols, respectively, where four workers were used in the parallel tool. The model checking running performance improvement is understandable because there are some extra costs when using our parallel version, such as socket communication overheads between the master and workers. On the other hand, Maude model checker is better than both of our sequential and parallel versions from the running performance point of view for KM. KM is a simple unidirectional token ring in which each process has an equal chance to use the privilege to change its status if the privilege is present at the process. Meanwhile, there is only one process that can take its move at one time. In other words, KM has a symmetry for each process that can take its move if the privilege is present at the process. Furthermore, the initial state used in KM gives an equal chance to each process that can take its move at the beginning. In addition, Fig. 2 shows the reachable state space of KM with 2 processes used that contains some lasso loops, although they are not long lasso loops. That implies that there may be some lasso loops in the reachable state space of KM with 10 processes used. Therefore, lots of sub-state space in the final layer are then likely to share lots of states, which makes it impossible to effectively utilize both shared and local

caches to keep away from tackling duplicated jobs. Qlock and Anderson do not have such a symmetry and so each process does not have an equal chance to enter the critical section. This would be probably why both (2) and (3) do not exhibit better model checking running performance than (1) for KM, although our parallel version outperforms our sequential version when using four workers as shown in Table 3. Because it is a commonly used practice to break any symmetries when designing good concurrent/distributed protocols [21], both the sequential and parallel versions of DCA2CSMC would effectively handle well-designed concurrent/distributed self-stabilizing systems/protocols and moreover the parallel version would outperform the sequential version.

It is possible to increase the number of workers in a flexible way when we would like to employ more computing resources available. We conducted experiments with different numbers of workers with the parallel tool for the three protocols. The experimental data are shown in Table 3, and also plotted on the graph shown in Fig. 4 as well. We can simply see that the verification time of the protocols improves quickly when we increase the number of workers from 4 to 8, however, it improves slower when we increase the number of workers from 8 to 12 and 12 to 16. Up to a certain point, the more workers used, the busier the master needs to handle and communicate with workers that may make the improvement slower compared to the number of workers increased. Thus, based on the power of the machine used to conduct model checking experiments, we may choose an appropriate number of workers when using our parallel tool. When the number of workers is 16, our parallel tool can largely improve the running performance of the sequential version by 87%, 92%, and 91% for Qlock, Anderson, and KM, respectively. In summary, our parallel version of DCA2CSMC has better model checking running performance than our sequential version of DCA2CSMC for all three protocols and than Maude model checker for the two mutual exclusion protocols but not for the simple token ring protocol.

VIII. RELATED WORK

A successful technique that eases the state space explosion is SAT/SMT-based bounded model-checking (SAT/SMT-BMC) [22]. Although it is possible to discover a counterexample placed at a non-deep depth from each initial state, it is in general impossible to prove that a system enjoys desired properties. SAT/SMT-BMC has been extended so that it is possible to prove that a system enjoys desired properties. k-induction [23], [24] is one of such extensions and combines mathematical induction and SAT/SMT-BMC. SAT/SMT-BMC is used to initially tackle the bounded state space from each initial state up to depth k. This is considered the base case. For each sequence $s_0, s_1, \ldots, s_k$ of states that starts with an arbitrary state s_0 in which the property to be verified is satisfied in each state s_i, the following is inspected: the property is satisfied in all successor states s_{k+1} of s_k. This is carried out by an SAT/SMT solver and considered the induction step. Our technique [9] and SAT/SMT-BMC share the basic idea that the former tackles the bounded state space that starts with each state placed at the beginning of each non-final layer. DCA2CSMC can be considered another extension of BMC but never uses any solvers of SAT/SMT.

Some recent advancements of parallel model checking algorithms for LTL are surveyed by Barnat, et al. [6]. It is necessary to redesign graph search algorithms so as to make the best use of multi-core architectures. Among parallel model checkers based on such algorithms are DiVinE 3.0 [25], and a multicore extension of SPIN [26]. We do not need to redesign graph search algorithms to implement a parallel version of DCA2CSMC and can use any existing LTL model checker for it, which is an essential difference from any existing parallel model checkers.

Inverso, et al. [27] have extended SAT/SMT-based BMC in order to model check concurrent programs. Let u be the unwinding (or unfolding) bound and r be the number of round-robin schedules. A concurrent program P is first transformed to an intermediate bounded program P_u by unfolding all loops and inlining all function calls in P with u as a bound except for those used for creating threads. P_u is next converted into a sequential program $Q_{u,r}$ that simulates all behaviors of P_u within r round-robin schedules. $Q_{u,r}$ is then translated into a propositional formula that can be analyzed by a SAT/SMT solver. Analyzing such a propositional formula with a SAT/SMT solver can be parallelized by decomposing the formula into multiple sub-formulas, assigning these sub-formulas to multiple instances of a SAT/SMT solver, and tackling the sub-formulas with the multiple instances in parallel [28]. This approach seems to be able to deal with safety properties, while our tools are able to deal with conditional stable properties, a class of liveness properties.

Lerda and Sisto [29] propose distributed-memory model checking with SPIN. Their proposed technique and ours in the present paper share a purpose. If a systems formal specification under verification with SPIN becomes larger than the physical memory carried by a computer in use, then the model checking running performance gets much slower or even might get impossible. To address it, Lerda and Sisto invented a way to divide the reachable state space of a large-state systems formal specification into multiple nodes (or computers) connected with networks. Their technique can be used together with some optimization techniques employed by SPIN, such as partial order reduction and bit state hashing. Lerda and Sisto carried out some case studies to show the effectiveness of their proposed technique. Their distributed-memory SPIN makes it possible to handle safety properties only, while our sequential and parallel versions of DCA2CSMC are able to deal with conditional stable properties, a class of liveness properties.

Although a bit-state verification mode of SPIN may be able to detect a bug hiding in a large-state systems formal specification that cannot be exhaustively tackled, it is more likely to overlook a bug lurking in a larger system specification. Holzmann, et al. have proposed Swarm Verification [30] to alleviate the situation. Parallelism and search diversity are

the key ideas of Swam Verification. Multiple instances of bit-state verification use multiple different search strategies so as to be more likely to traverse different portions of the entire reachable state space, increase the coverage of the entire reachable state space, and find bugs lurking in a large system specification. Multiple instances of bit-state verification can run in parallel. DeFrancisco, et al. [31] have implemented Swarm Verification on GPUs, called Grapple. This adoption may be able to detect a flaw hiding in a large-state systems formal specification more quickly than the current parallel technique/tool.

IX. CONCLUSION

We have described a sequential tool, a parallel technique with the master-worker pattern in the form of pseudo-code, and a parallel tool for DCA2CSMC. Both the sequential and parallel tools have been implemented in Maude. We have carried out some case studies showing that the sequential and parallel tools ease the state space explosion and improve the running performance of model checking for conditional stable properties to a certain scope, respectively. The parallel technique/tool exhibits better running performance than the sequential technique/tool. As usual, however, there are several things left to do as future work. For example, more case studies should be carried out with the tools.

To effectively use our proposed techniques/tools, it is necessary to make a formal systems specification so that each sub-state space generated has a much smaller number of states than the number of states in the whole reachable state space of the formal systems specification. For example, we need to get rid of any long lasso loops that may prevent some sub-state spaces in the final layer from having a much smaller number of states than the number of states in the whole reachable state space of the formal systems specification. The formal systems specifications of the self-stabilizing versions of Qlock and Anderson utilized for the case studies in the present paper do not have any lasso loops, which may make the behavior of self-stabilizing protocols less exciting. For Qlock and Anderson, we suppose that each process goes to the critical section (*cs*) at most once and finally stays at the final section (*fs*) forever, which prevents their formal specifications from having long lasso loops. The protocols recover an illegitimate state to a legitimate state by basically making processes that stay in and wait for entering *cs* move to *fs*. Such processes will never enter *cs*. If it is possible to freely use long lasso loops, we can revise the protocols such that each process can enter *cs* as many times as it wants, making the behaviors of the protocols more fascinating. It is a challenge to efficiently deal with long lasso loops for our approach. Therefore, we need to come up with a technique that can handle such long lasso loops as one piece of our future work. One possible approach to it is as follows: each long lasso loop is divided into multiple short finite sequences of states, model checking experiments for these finite sequences are conducted and their model checking results are combined to conclude the model checking experiment for the long lasso loop.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers who carefully read an earlier version of the article and gave them valuable comments without which they were not able to complete the present article.

REFERENCES

[1] M. E. Clarke, A. T. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Cham, Switzerland: Springer, 2018.
[2] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *Int. J. Softw. Tools Technol. Transf. (STTT)*, vol. 2, no. 3, pp. 279–287, Nov. 1999.
[3] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.
[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.
[5] J. Meseguer, M. Palomino, and N. Martí-Oliet, "Equational abstractions," *Theor. Comput. Sci.*, vol. 403, nos. 2–3, pp. 239–264, Aug. 2008.
[6] J. Barnat, V. Bloemen, A. Duret-Lutz, A. Laarman, L. Petrucci, J. V. D. Pol, and E. Renault, "Parallel model checking algorithms for linear-time temporal logic," in *Handbook of Parallel Constraint Reasoning*. Cham, Switzerland: Springer, 2018, pp. 457–507.
[7] Y. Phyo, C. M. Do, and K. Ogata, "A divide & conquer approach to leads-to model checking," *Comput. J.*, vol. 63, no. 2, pp. 1353–1364, Jun. 2021.
[8] M. N. Aung, Y. Phyo, C. M. Do, and K. Ogata, "A divide and conquer approach to eventual model checking," *Mathematics*, vol. 9, no. 4, p. 368, 2021.
[9] Y. Phyo, C. M. Do, and K. Ogata, "A divide & conquer approach to conditional stable model checking," in *Proc. 18th Int. Colloq. Theor. Aspects Comput.*, 2021, pp. 105–111.
[10] Y. Phyo, C. Minh Do, and K. Ogata, "A support tool for the $L + 1$-layer divide & conquer approach to leads-to model checking," in *Proc. IEEE 45th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jul. 2021, pp. 854–863.
[11] C. M. Do, Y. Phyo, A. Riesco, and K. Ogata, "A parallel stratified model checking technique/tool for leads-to properties," in *Proc. 7th Int. Symp. Syst. Softw. Rel. (ISSSR)*, Sep. 2021, pp. 155–166.
[12] M. N. Aung, Y. Phyo, C. M. Do, and K. Ogata, "A tool for model checking eventual model checking in a stratified way," in *Proc. 9th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Aug. 2022, pp. 270–279.
[13] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.
[14] J. Meseguer, "Twenty years of rewriting logic," *J. Logic Algebraic Program.*, vol. 81, nos. 7–8, pp. 721–781, 2012.
[15] C. M. Do, A. Riesco, S. Escobar, and K. Ogata, "Parallel Maude-NPA for cryptographic protocol analysis," in *Rewriting Logic and Its Applications* (Lecture Notes in Computer Science), vol. 13252. Cham, Switzerland: Springer, 2022, pp. 253–273.
[16] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," *Electron. Notes Theor. Comput. Sci.*, vol. 71, pp. 162–187, Apr. 2004.
[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, *All About Maude—A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic* (Lecture Notes in Computer Science), vol. 4350. Berlin, Germany: Springer, 2007.
[18] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[19] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker and its implementation," in *Model Checking Software*. Berlin, Germany: Springer, 2003, pp. 230–234.
[20] E. T. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.
[21] A. Nancy Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann, 1996.
[22] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, 2001.

[23] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Proc. FMCAD*, in Lecture Notes in Computer Science, vol. 1954. Austin, TX, USA: Springer, Nov. 2000, pp. 108–125.
[24] L. M. D. Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *Proc. Int. Conf. Comput. Aided Verification*, in Lecture Notes in Computer Science, vol. 2725. Boulder, CO, USA: Springer, Jul. 2003, pp. 14–26.
[25] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser, "DiVinE 3.0—An explicit-state model checker for multithreaded C & C++ programs," in *Proc. Int. Conf. Comput. Aided Verification*, vol. 8044. Berlin, Germany: Springer, 2013, pp. 863–868.
[26] G. J. Holzmann and D. Bosnacki, "The design of a multicore extension of the SPIN model checker," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 659–674, Oct. 2007.
[27] O. Inverso, E. Tomasco, B. Fischer, S. L. Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *Proc. Int. Conf. Comput. Aided Verification*, vol. 8559. Cham, Switzerland: Springer, 2014, pp. 585–602.
[28] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2020, pp. 202–216.
[29] F. Lerda and R. Sisto, "Distributed-memory model checking with SPIN," in *Proc. Int. SPIN Workshop Model Checking Software*, vol. 1680. Berlin, Germany: Springer, 1999, pp. 22–39.
[30] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 845–857, Nov. 2011.
[31] R. DeFrancisco, S. Cho, M. Ferdman, and S. A. Smolka, "Swarm model checking on the GPU," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 5, pp. 583–599, Oct. 2020.

CANH MINH DO received the B.S. degree in information technology from the National Economics University, in 2013, and the M.S. and Ph.D. degrees in information science from the Japan Advanced Institute of Science and Technology (JAIST), in 2019 and 2022, respectively.

He has been working on how to mitigate the state space explosion in model checking and improve the running performance of model checking by parallelization.

YATI PHYO received the B.S. degree in information technology from the University of Technology, in 2017, and the M.S. and Ph.D. degrees in information science from the Japan Advanced Institute of Science and Technology (JAIST), in 2019 and 2022, respectively.

She has been working on how to mitigate the state space explosion in model checking by a divide and conquer approach.

KAZUHIRO OGATA received the B.S., M.S., and Ph.D. degrees in engineering from Keio University, in 1990, 1992, and 1995, respectively.

He is currently a Professor with the Japan Advanced Institute of Science and Technology (JAIST). His research interest includes applications of formal methods to systems, such as distributed systems and security protocols.