Computing and Informatics, Vol. 38, 2019, 1009–1038, doi: 10.31577/cai_2019_5_1009

A MORE FAITHFUL FORMAL DEFINITION OF THE DESIRED PROPERTY FOR DISTRIBUTED SNAPSHOT ALGORITHMS TO MODEL CHECK THE PROPERTY

Ha Thi Thu Doan, Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi,
Ishikawa, 923-1292, Japan
e-mail: {doanha, ogata}@jaist.ac.jp

Abstract. The first distributed snapshot algorithm was invented by Chandy and Lamport: Chandy-Lamport distributed snapshot algorithm (CLDSA). Distributed snapshot algorithms are crucial components to make distributed systems fault tolerant. Such algorithms are extremely important because many modern key software systems are in the form of distributed systems and should be fault tolerant. There are at least two desired properties such algorithms should satisfy: 1) the distributed snapshot reachability property (called the DSR property) and 2) the ability to run concurrently with, but not alter, an underlying distributed system (UDS). This paper identifies subtle errors in a paper on formalization of the DSR property and shows how to correct them. We give a more faithful formal definition of the DSR property; the definition involves two state machines – one state machine M_{UDS} that formalizes a UDS and the other M_{CLDSA} that formalizes the UDS on which CLDSA is superimposed (UDS-CLDSA) – and can be used to more precise model checking of the DSR property for CLDSA. We also prove a theorem on equivalence of our new definition and an existing one that only involves M_{CLDSA} to guarantee the validity of the existing model checking approach. Moreover, we prove the second property, namely that CLDSA does not alter the behaviors of UDS.

Keywords: Distributed snapshot algorithm, reachability, state machine, property specification, model checking

Mathematics Subject Classification 2010: 68N30

1 INTRODUCTION

Many modern key software systems are in the form of distributed systems [1, 2], which consist of many components coordinating their actions by passing messages and working together to achieve a common goal. The modern distributed applications, such as cloud computing [3] and web search [4], are getting more complicated. Such systems should be fault tolerant because they need to run for a long time, keeping on providing services to users, other systems, etc. To make distributed systems fault tolerant, it is necessary to use many non-trivial distributed algorithms, such as snapshot algorithms and self-stabilizing algorithms. Distributed snapshot algorithms (DSAs) deal with a significant problem, recording global states of a distributed system, which helps solving others, such as recovering from faulty states and detecting stable properties. Therefore, DSAs become the core of many fault tolerant distributed systems. However, the challenge is how to determine a consistent global state. To clearly record consistent global states, a DSA should satisfy two properties:

1. the distributed snapshot reachability property (called the DSR property) and
2. the ability to run concurrently with, but not alter, an underlying distributed system (UDS).

Considered as the most important desired property of the algorithm, the DSR property is as follows: Let s_1 be the state in which a DSA initiates, s_2 be the state in which DSA terminates, and s_* be the snapshot taken, then s_* is reachable from s_1 and s_2 is reachable from s_*.

Because distributed systems are usually complicated and it is hard to ensure their reliability, the formal verification of distributed systems is essential. Model checking [5, 6] is a popular automatic formal technique for verifying state transition systems. Many model checkers, such as Spin [7], NuSMV[8], and TLA+ [9], have been developed in order to formally verify various kinds of software and hardware systems. Model checking is highly suitable for formally verifying distributed systems. Many researchers [10, 11, 12] are interested in model checking of distributed systems. However, the application of model checking to verification of DSAs has not been fully investigated due to the specific characteristic of DSAs (they are superimposed on, but do not interfere with UDSs).

This paper focuses on Chandy-Lamport distributed snapshot algorithm (CLDSA) [13] that is the first such algorithm. Several variants of CLDSA, such as Spezialetti-Kearns algorithm [14] and Venkatesans incremental snapshot algorithm [15], have been proposed. In an attempt to formally analyze CLDSA, the authors in [16] have used Maude [17] to model check that CLDSA enjoys the DSR property. Maude is a language and a system supporting executable specification and declarative programming in rewriting logic. Two model checking facilities equipped with Maude are the LTL model checker and the search command. In [16], the Maude search command is used. However, the DSR property is encoded in

terms of the Maude search command and the encoding does not reflect the informal description of the property originally given in [13]. We recognize that the informal description of the DSR property involves both a UDS and the UDS on which CLDSA is superimposed (UDS-CLDSA), while their definition of the property involves only the UDS-CLDSA. Consequently, we do not think that the existing study provides a good foundation for meaningful model checking of the DSR property for CLDSA. Therefore, it is necessary to express the DSR property accurately and then consider the equivalence between the new formal definition and the existing one.

The challenge is that the DSR property relates to two different systems: a UDS and the UDS-CLDSA. It is not straightforward to express the property in typical existing temporal logics, such as linear temporal logic (LTL) and computation tree logic (CTL) because such a temporal logic only takes into account one state machine or a Kripke structure. The authors of the technical paper [18] attempt to find a way to express the DSR property more faithfully. However, the technical report [18] has not been reviewed. We detected flaws lurking in their formalization.

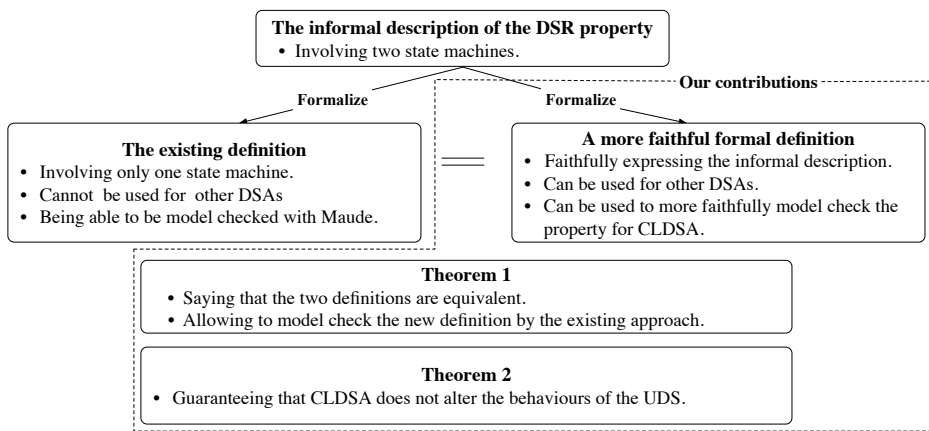


Figure 1. The contributions of the paper

This paper is an extended and revised version of the paper [19], which shows a thorough study on formal expression of the DSR property. We have pointed out mistakes in [18] in detail. Correcting these mistakes, we formalize a UDS and the UDS-CLDSA as state machines more precisely. Carefully investigating the informal description of the DSR property, we give a formal definition of the property, which is more likely to faithfully express the informal description and can be used to more precisely model check of the CLDSA property. The essential of our method to formalize a UDS and the UDS-CLDSA is that the formalization of a UDS-CLDSA is generated from the one of the UDS. This is because

we take into account an important aspect of the algorithm, namely that the algorithm runs concurrently with, but does not alter, the behaviors of UDS. Not only snapshot algorithms but also many other distributed algorithms (called control algorithms [21]), such as checkpointing algorithms and termination detection algorithms, do not alter the behaviors of the UDS. These algorithm executions are superimposed on the underlying application execution, but do not interfere with the application execution. Therefore, one considerable contribution of our research is a method by which the class of these algorithms can be precisely formalized, and then any of their desired properties that are related to both a UDS and the UDS on which these algorithms are superimposed could be formally expressed. Furthermore, we prove that our new definition is equivalent to the existing one in [16] to confirm the validity of the model checking approach. It is not straightforward to directly model check the new definition with an existing model checker because two different state machines should be taken into account, while existing model checkers, such as Spin, NuSMV, and TLC, support model checking for systems that can be formalized as only one state machine. The equivalence of the two definitions suggests we can use an existing model checker to tackle the DSR property. Moreover, we prove that CLDSA runs concurrently, but does not alter, a UDS as the second property. More on our contributions are depicted in Figure 1.

The rest of the paper is organized as follows. The next section introduces some preliminaries. Section 3 describes how to formalize a UDS and the UDS-CLDSA as state machines. Section 4 gives the new definition of the DSR property. Section 5 presents the theorem that guarantees the validity of the existing model checking approach. Section 6 proves the theorem on simulation between M_{CLDSA} and M_{UDS} to confirm the second property. Section 7 mentions related work, and Section 8 concludes the paper.

2 PRELIMINARIES

2.1 Chandy-Lamport Distributed Snapshot Algorithm (CLDSA)

CLDSA works by using control messages called *markers*. Each process can record its local state when it has not yet received any marker from its incoming channels. It then sends one marker along each of its outgoing channels. Because the channel is FIFO, the marker acts as a separator for the messages in the channel to determinate the sequence of messages that should be recorded in the state of the channel. There are two rules by which a process will execute the marker-sending rule on recording its local state and the marker-receiving rule on receiving a marker, respectively. The outline of the algorithm is as follows:

Marker-Sending Rule for a process p: for each its outgoing channel c, p sends one marker along c after recording its state and before sending further messages along c.

Marker-Receiving Rule for a process p: when the process p gets a marker from one of its incoming channels c,

- **if** p has not yet recorded its state **then** p records its state according to Marker-Sending Rule for p and the state of channel c as an empty sequence
- **else** p records the state of c as the sequence of messages received along c after recording p's state and before receiving the marker along c.

The algorithm will terminate when each process has already recorded its state and the states of all of its incoming channels. The global snapshot is made by combining those recorded process and channel states.

2.2 State Machine

A state machine consists of a set of states, some of which are initial states, and a binary relation over states. The definition is as follows:

Definition 1 (State Machine)**.** A state machine $M \triangleq \langle S, I, T \rangle$ consists of

1. a set of states S;
2. a set of initial states $I \subseteq S$;
3. a total binary relation[1] over states $T \subseteq S \times S$.

Elements (s, s') of the binary relation are called state transitions, where (s, s') says that s can change to s'. Since T is total, for each state $s \in$ S there exists a state $s' \in S$ such that $(s, s') \in T$. The reachable states of a state machine are inductively defined as follows:

1. each initial state is reachable and
2. for each reachable state s and each state transition (s, s'), s' is reachable.

A state predicate p is an invariant property of the state machine if and only if $p(s)$ holds for all reachable states s of the state machine.

2.3 Maude

Maude [17] is a specification and programming language based on rewriting logic. The basic units of specifications and programs are modules. A module contains syntax declarations, providing a suitable language to describe a system. A module

[1] Standard definitions of state machines do not require that T is total, but it is convenient to be able to generate infinite state sequences from state machines for proving that one state machine can simulate another one and then we assume that T is total. Note that a Kripke structure, which is used to define semantics of temporal logics and includes a state machine, requires that T is total and then our assumption would be reasonable.

consists of sort, sub-sort, operator, variable, equation and membership declarations, as well as rewrite rule declarations. Note that membership declarations are not used at all in this paper.

A sort denotes a set, corresponding to a type in conventional programming languages. For example, the sort `Nat` denotes the set of natural numbers. There is the relation among sorts which is the same as the subset relation. A sort is a subsort of another sort if and only if the set denoted by the former is a subset of the one by the latter, and the latter is called a supersort of the former. Let `Zero` and `NzNat` be the sorts denoting $\{0\}$ and the set of non-zero natural numbers, and then both of which are subsets of the set of natural numbers. `Zero` and `NzNat` are subsorts of `Nat` and `Nat` is a supersort of `Zero` and `NzNat`. An operator is declared as follows: $f : S_1 \ldots S_n \to S$, where $S_1, \ldots, S_n, S$ are sorts and $n \geq 0$. Note that `->` may be used instead of $\to$. $S_1 \ldots S_n$, S and $S_1 \ldots S_n \to S$ are called the arity, the sort and the rank of f, respectively. When $n = 0$, f is called a constant. f takes a sequence of n things of $S_1, \ldots, S_n$ and makes something of S, where "things" and "something" are what are called terms and will be described soon. Operators denote functions or data constructors. Each variable has its own sort. Terms of a sort S are inductively defined as follows:

1. each variable whose sort is S is a term of S and
2. for each operator f whose rank is $S_1 \ldots S_n \to S$ and n terms $t_1, \ldots, t_n$ of $S_1 \ldots S_n$, $f(t_1, \ldots, t_n)$ is a term of S.

Note that when $n = 0$, f as it is a term of S. An operator may contain underscores, such as `_+_ : Nat Nat -> Nat`. If that is the case, a different notation than $f(t_1, \ldots, t_n)$ is used. For example, if `X` is a variable of `Nat`, then `X + X` is a term of `Nat`. Use of operators containing underscores allows us to define context-free grammars. An equation declares that two terms are equal. A rewrite rule declares that a term changes to another one. Equations can be used to define functions, while rewrite rules can be used to define state transitions. A sort and the set denoted by the sort are interchangeably used in this paper.

3 THE FORMALIZATIONS OF A UDS AND THE UDS-CLDSA

3.1 Formalizing a UDS as a State Machine

A UDS consists of a finite set of processes that are connected and communicated by channels. This paper considers snapshot algorithms suited for FIFO channels. Channels are assumed to deliver messages in the order sent. Messages are delivered reliably without any error in finite but arbitrary time. There may be more than one channel from one process to another. A UDS can be described as a labeled, directed graph in which the vertices represent the processes and the directed edges represent the channels. Figure 2 describes a UDS consisting of three processes p, q, r and four channels $c1$, $c2$, $c3$, $c4$. The channel $c4$ is used to directly send messages from r to p, but not vice versa.

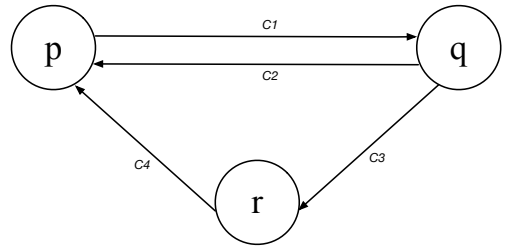


Figure 2. A distributed system with processes p, q, r and channels $c1$, $c2$, $c3$, $c4$

A global state of a distributed system is a collection of component process and channel states. The state of a channel is characterized by the sequence of in-trans messages sent along the channel and have not yet received by the destination process. The initial global state is one in which each process is in its initial state and the state of each channel is the empty sequence.

Since state machines are suitable to formalize concurrent systems, they are used to formalize a UDS and the UDS-CLDSA in our research. In this paper, Maude notation is used to describe state machines. Let M_{UDS} be the state machine that formalizes a UDS. We first consider how to express the states of a UDS and the state transitions of M_{UDS}.

Because our aim is to model check desired properties for CLDSA, we suppose that the reachable states of a UDS are bounded and so are the reachable states of the UDS-CLDSA. Note that the whole states of a UDS may be unbounded and so may be the whole states of the UDS-CLDSA. Because we formalize a channel from one process to another as an inductively defined queue, the whole states of a UDS are unbounded. However, the reachable states of the UDS could be bounded, for example, by preventing new messages from being generated.

3.1.1 State Expression

The states of process and channel are described by *name: value* pairs (called observable components), where *name* may have parameters: *p-state[_]:_* for process, where the parameter of *name* is a process identifier and *value* is the state of the process and *c-state[_,_,_]:_* for channel, where the three parameters are a source process, a destination process and a natural number, respectively, and *value* is the state of the channel. OCom is the sort for those observable components. These observable components are expressed by the following operators that are constructors as specified with *ctor*:

op p-state[_]:_ : Pid PState → OCom [ctor].

op c-state[_, _, _]:_ : Pid Pid Nat MsgQueue$P$ → OCom [ctor].

where Pid, PState, Nat and MsgQueue are the sorts for process identifiers, process

states, natural numbers and queues of messages for which the sort Msg is used, respectively.

Let p, $q \in$ Pid, $ps \in$ PState, $n \in$ Nat and $ms \in$ MsgQueue. (p-state[p]: ps) is an observable component whose name is p-state[p] where p is a parameter and whose value is ps, expressed as a term of OCom that says that the local state of a process p is ps. (c-state[p, q, n]: ms) is an observable component whose name is c-state[p, q, n] where p, q, n are parameters and whose value is ms, expressed as a term of OCom that says that the state of a channel from the process p to the process q is ms. Since there may be more than one channel from p to q, a natural number n is used in (c-state[p, q, n]: ms) to identify the channel. We use an associative-commutative collection (called a soup) to express global states of a UDS. The corresponding sort is ConFigure. The following operators are prepared to construct the sort.

subsort OCom < Config,

op empConfig : → Config [ctor],

op _ _ : Config Config → Config [ctor assoc comm id:empConfig]

where empConfig denotes the empty soup of observable components. Config is a super-sort of OCom, which means that each term of OCom is treated as the singleton soup only consisting of the term. The juxtaposition operator _ _ is used to construct soups of observable components. For $c_1, c_2 \in$ Config, $c_1 c_2 \in$ ConFigure. The juxtaposition operator is associative and commutative as specified with *assoc* and *comm*, and empConfig is an identity of the operator specified with id:empConfig.

3.1.2 State Transitions

Each process in a UDS may do three kinds of actions:

i. Change of Process State: it may change its state without sending or receiving any message,
ii. Sending of Message: it may send a message by putting the message into one of its outgoing channels and may change its state (or may not change its state), and
iii. Receipt of Message: it may get the top message from one of its incoming channels if the channel is not empty and may change its state (or may not change its state).

These actions are described as transition rules. A transition rule is described in the form of rewrite rules[2]. In the following part, $P, Q \in$ Pid, $PS1, PS2 \in$ PState,

[2] Each rewrite rule (and each equation) should be executable and the rewrite rule should be split into multiple executable ones so that model checking can be doable with Maude. Since the main purpose of the paper is to give a more faithful definition of the DSR property and confirm the validity of the model checking approach used in the existing study, we make each rewrite rule as general as possible to cover all possible situations.

$N \in$ Nat, $CS \in$ MsgQueue and $M \in$ Msg are variables of those sorts.

- Change of Process State is described as the following transition rule:

  $$(\text{p-state}[P] : PS1) \Rightarrow (\text{p-state}[P] : PS2).$$

- Sending of Message is described as the following transition rule:

  $$(\text{p-state}[P] : PS1)(\text{c-state}[P, Q, N] : \text{CS}) \Rightarrow$$
  $$(\text{p-state}[P] : PS2)(\text{c-state}[P, Q, N] : \text{enq(CS, M)})$$

  where $PS1$ may be the same as $PS2$ and enq is a standard function for queues, taking a queue q and an element e and putting e into q at bottom.

- Receipt of Message is described as the following transition rule:

  $$(\text{p-state}[P] : PS1)(\text{c-state}[Q, P, N] : M \mid \text{CS}) \Rightarrow$$
  $$(\text{p-state}[P] : PS2)(\text{c-state}[Q, P, N] : \text{CS})$$

  where $PS1$ may be the same as $PS2$. The operator $\_|\_$ is used to construct queues of messages. For $m \in$ Msg and $q \in$ MsgQueue, $m \mid q \in$ MsgQueue, where m is the top message of the queue.

Although three kinds of actions are generally described by only three transition rules, there may be more than three ground instances of the transition rules. This is caused by the number of states for each process, the number of channels, the number of states for each channel, etc. Given a transition rule L $\Rightarrow$ R, a ground instance of the transition rule is obtained by replacing each variable in L $\Rightarrow$ R with a ground constructor term (or a value) of the sort of the variable.

Definition 2 (TR_{UDS})**.** Let TR_{UDS} be the set of all ground instances of the three transition rules.

3.1.3 State Machine M_{UDS}

A UDS is formalized as state machine $M_{UDS} \triangleq \langle S_UDS, I_{UDS}, T_{UDS} \rangle$. S_UDS is the set of all ground constructor terms whose sorts are ConFigure. I_{UDS} is a subset of S_UDS such that for each state $s \in I_{UDS}$, for each channel c in s the message queue in c is empty, for each process $p \in$ Pid there exists at most one p-state[p] observable component in s, for each (p, q, n) where $p, q \in$ Pid and $n \in$ Nat, there exists at most one c-state[p, q, n] observable component in s, and there is no dangling channel in s. T_{UDS} is the binary relation over S_UDS made from TR_{UDS}.

Definition 3 (M_{UDS})**.** The state machine formalizing a UDS is $M_{UDS} \triangleq \langle S_UDS, I_{UDS}, T_{UDS} \rangle$, where

1. S_UDS is the set of all ground constructor terms whose sorts are Config;

2. I_{UDS} is a subset of S_UDS such that $(\forall s \in I_{UDS})\,(\forall c \in \text{chans}(s))\,(\text{msg}(c) = \text{empChan})$, $(\forall s \in I_{UDS})\,(\forall p \in \text{Pid})\,(\#(s,p) \leq 1)$, $(\forall s \in I_{UDS})\,(\forall p,q \in \text{Pid})$ $(\forall n \in \text{Nat})\,(\#(s,p,q,n) \leq 1)$ and $(\forall s \in I_{UDS})\,(\forall(\text{c-state}[p,q,n] : \text{cs}) \in s)$ $((\#(s,p) = 1) \wedge (\#(s,q) = 1))$;
3. T_{UDS} is the binary relation over S_UDS defined as follows: $\{(C, C') \mid C,\, C' \in \text{Config},\, L \Rightarrow R \in TR_{UDS},\, \exists C''.(C = LC'' \wedge C' = RC'')\}$.

Function chans gets all channels of a state $s \in S_UDS$, msg gets the state of a channel c in s, and # counts the number of occurrences of a process or a channel in s.

3.2 Generating State Machine M_{CLDSA} from State Machine M_{UDS}

Let M_{CLDSA} be the state machine that formalizes the UDS-CLDSA. Because CLDSA runs concurrently with, but does not alter, the behaviors of a UDS, it is possible to generate the formalization of the UDS-CLDSA from the UDS's. The authors in [18] show their way to do that. Carefully investigating, however, we found mistakes in their formalizations. Our detection of their mistakes will be presented in the rest of this section, where we focus on how to generate M_{CLDSA} from M_{UDS}. Let us consider the state expression and the state transition for M_{CLDSA}.

3.2.1 State Expression

Each state of M_{CLDSA} should consist of the local states of all processes and channels, the state (called the start state) when CLDSA initiates, the snapshot, the state (called the finish state) when CLDSA terminates and the information to control the behaviors of CLDSA.

With superficially observing, we may mistakenly comprehend that the local states of each process and channel in the UDS-CLDSA are exactly the same as the one of a UDS. They are, however, different. The key difference is that due to the working of CLDSA, a channel includes not only the normal data messages as those of a UDS, but also the control messages, markers. This is an important point needed to be noticed explicitly when modeling the system. As mistakes, the authors in [18] did not see this point. They consider the states of a channel in the system exactly the same as those for a UDS. The same sorts Msg, MsgQueue, OCom and Config for messages, the sequence of messages, observable components and a soup of *p-state* and *c-state* observable components, respectively, are used in their formalization of the UDS-CLDSA. Overcoming the problem, we use other sorts to describe the UDS-CLDSA. In detail, MMsg, a super-sort of sort Msg, is used for messages and markers. The sorts BOCom and BConfig, replacing OCom and Config, correspondingly, are used for observable components and a soup of *p-state* and *c-state* observable components. For this end, the following sorts and operators are defined:

subsort Msg < MMsg.

op marker :→ MMsg[*ctor*].

op empChan :→ MMsgQueue[*ctor*].

op _|_ : MMsg MMsgQueue → MMsgQueue[*ctor*].

op p-state[_]:_ : Pid PState → BOCom[*ctor*].

op c-state[_, _, _]:_ : Pid Pid Nat MMsgQueue → BOCom[*ctor*].

subsort BOCom < BConfig.

op empBConfig :→ BConfig[*ctor*].

op _ _ : BConfig BConfig → BConfig[ctor assoc comm id:empBConfig].

We use base-state(bc), start-state(sc), snapshot(©ssc) and finish-state(fc) called meta configuration components, in which bc, sc, ssc and fc are ground constructor terms of sort BConfig, to express the local states of all processes and channels, the start state, the snapshot and the finish state, respectively. The corresponding sort for those components is MBCom. The information to control behaviors of CLDSA is expressed as control(ctl) that is also a meta configuration component. ctl is a soup of cnt, prog, #ms, and done control observable components that will be described soon. CtlOCom is the sort for those components, and CtlConfig is the sort for soups of CtlOCom.

ops base-state start-state snapshot finish-state : BConfig→ MBCom[ctor].

op control : CtlConfig → MBCom[ctor].

The control observable components are as follows.

(cnt : n): n is the number of processes that have not yet completed CLDSA.

(prog[p] : pg): pg is the progress (notYet, started or completed) of a process p, indicating that the process has not yet started, has started, or completed CLDSA.

(#ms[p] : n): n is the number of incoming channels to a process p from which markers have not yet been received.

(done[p, q, n] : b): b is either $true$ or $false$. If b is $true$, q has received a marker from the incoming channel identified by n from p, otherwise, q has not.

Each state of M_{CLDSA} is expressed as the soup of the meta configuration components, which is in the form:

base-state(bc) start-state(sc) snapshot(©ssc) finish-state(fc) control(ctl)

which is called a meta configuration and the corresponding sort is MBConFigure. We define the following operators for the sort:

subsort MBCom < MBConfig.

op _ _ : MBConfig MBConfig → MBConfig[ctor assoc comm].

Initially, *bc* is an initial state of M_{UDS}, and all of *sc*, *ssc* and fc are *empBConfig*. The number of processes that have not yet completed CLDSA is equal to the number of processes in the system, the progress of all processes are *notYet*, and all processes have not yet received any markers. If fc is not *empBConfig*, a distributed snapshot has been taken and then *ssc* is the snapshot.

3.2.2 State Transitions

Each process in the system preserves the basic actions of those for a UDS, but needs to do two more kinds of actions for executing CLDSA as follows.

iv. Record of Process State: it may record its state and put markers into all of its outgoing channels when it has not yet received any markers, and

v. Receipt of Marker: it may get a marker from one of its incoming channels.

In the following part, P, $Q \in$ Pid, $PS, PS1, PS2 \in$ PState, BC, $SSC \in$ BConfig, $MMS \in$ MMsgQueue, $CC \in$ CtlConfig, $N \in$ Nat, $NzN \in$ NzNat and $M \in$ Msg are variables of those sorts, where NzNat is the sort for non-zero natural numbers and a subsort of Nat.

- Change of Process State is described as the following transition rule:

  $$\text{base-state}((\text{p-state}[P] : PS1)BC) \Rightarrow \text{base-state}((\text{p-state}[P] : PS2)BC).$$

- Sending of Message is described as the following transition rule:

  $$\text{base-state}((\text{p-state}[P] : PS1)\ (\text{c-state}[P, Q, N] : MMS)BC) \Rightarrow$$
  $$\text{base-state}((\text{p-state}[P] : PS2)\ (\text{c-state}[P, Q, N] : \text{enq}(MMS, M))BC).$$

- Receipt of Message is split into four subcases:

  1. The process has not yet started CLDSA.

     $$\text{base-state}((\text{p-state}[P] : PS1)\ (\text{c-state}[Q, P, N] : M \mid MMS)BC)$$
     $$\text{control}((\text{prog}[P] : \text{notYet})\ CC)$$
     $$\Rightarrow$$
     $$\text{base-state}((\text{p-state}[P] : PS2)\ (\text{c-state}[Q, P, N] : MMS)BC)$$
     $$\text{control}((\text{prog}[P] : \text{notYet})CC).$$

  2. The process has completed CLDSA.

     $$\text{base-state}((\text{p-state}[P] : PS1)\ (\text{c-state}[Q, P, N] : M \mid MMS)BC)$$
     $$\text{control}((\text{prog}[P] : \text{completed})CC)$$
     $$\Rightarrow$$
     $$\text{base-state}((\text{p-state}[P] : PS2)\ (\text{c-state}[Q, P, N] : MMS)BC)$$
     $$\text{control}((\text{prog}[P] : \text{completed})CC).$$

3. The process has started CLDSA, not yet completed it, and not yet received a marker from the incoming channel.

   base-state((p-state[P] : $PS1$) (c-state[Q, P, N] : M | *MMS*)BC)
   snapshot((c-state[Q, P, N] : *MMS'*)SSC)
   control((prog[P] : started)(done[Q, P, N] : *false*)CC)
   $\Rightarrow$
   base-state((p-state[P] : $PS2$)(c-state[Q, P, N] : *MMS*)BC)
   snapshot((c-state[Q, P, N] : enq(*MMS'*, M))SSC)
   control((prog[P] : started)(done[Q, P, N] : *false*)CC).

4. The process has started CLDSA, but not yet completed it and it has already received a marker from the incoming channel.

   base-state((p-state[P] : $PS1$)(c-state[Q, P, N] : M | *MMS*)BC)
   control((prog[P] : started)(done[Q, P, N] : *true*)CC)
   $\Rightarrow$
   base-state((p-state[P] : $PS2$)(c-state[Q, P, N] : *MMS*)BC)
   control((prog[P] : started)(done[Q, P, N] : *true*)CC).

- Record of Process State is split into two subcases:

1. The process globally initiates CLDSA. This case is further split into three subcases:

   (a) The UDS only consists of the process.

   base-state((p-state[P] : PS)) start-state(empBConfig)
   snapshot(empBConfig) finish-state(empBConfig)
   control((prog[P] : notYet)(cnt : 1)(#ms[P] : 0)CC)
   $\Rightarrow$
   base-state((p-state[P] : PS)) start-state((p-state[P] : PS))
   snapshot((p-state[P] : PS)) finish-state((p-state[P] : PS))
   control((prog[P] : completed)(cnt : 0)(#ms[P] : 0)CC).

   (b) The system consists of more than one process, and the process does not have any incoming channels.

   base-state((p-state[P] : PS)BC) start-state(empBConfig)
   snapshot(empBConfig) finish-state(empBConfig)
   control((prog[P] : notYet)(cnt : NzN)(#ms[P] : 0)CC)
   $\Rightarrow$
   base-state((p-state[P] : PS) bcast(BC, P, marker))
   start-state((p-state[P] : PS)BC) snapshot((p-state[P] : PS))
   control((prog[P] : completed)(cnt : sd(NzN, 1))(#ms[P] : 0)CC)x
   if NzN > 1

   where bcast is a function putting markers in all outgoing channels from process P and sd is a function for natural number taking two natural

numbers x and y and then returning $x - y$ if $x > y$ and $y - x$ otherwise.

(c) The system consists of more than one process, and the process has one or more incoming channels.

base-state((p-state[P] : PS)BC) start-state(empBConfig)
snapshot(empBConfig) finish-state(empBConfig)
control((prog[P] : notYet)(#ms[P] : NzN')CC)
$\Rightarrow$
base-state((p-state[P] : PS) bcast(BC, P, marker))
start-state((p-state[P] : PS)BC)
snapshot((p-state[P] : PS) inchans(BC, P))
control((prog[P] : started)(#ms[P] : NzN')CC).

2. The process does not globally initiate CLDSA. This case is further split into three subcases:

(a) The process does not have any incoming channel, and there are no processes except for the process that have not completed CLDSA.

base-state((p-state[P] : PS)BC) start-state(SC) snapshot(SSC)
finish-state(empBConfig)
control((prog[P] : notYet)(cnt : 1)(#ms[P] : 0)CC)
$\Rightarrow$
base-state((p-state[P] : PS)) start-state(SC)
snapshot((p-state[P] : PS)SSC)
finish-state((p-state[P] : PS)BC)
control((prog[P] : completed)(cnt : 0)(#ms[P] : 0)CC)
if ($SC \neq$ empBConfig).

(b) The process does not have any incoming channels, and there are some other processes that have not completed CLDSA.

base-state((p-state[P] : PS)BC) start-state(SC) snapshot(SSC)
control((prog[P] : notYet)(cnt : NzN)(#ms[P] : 0)CC)
$\Rightarrow$
base-state((p-state[P] : PS) bcast(BC, P, marker))
start-state(SC) snapshot((p-state[P] : PS)SSC)
control((prog[P] : completed)(cnt : sd(NzN, 1))(#ms[P] : 0)CC)
if ($SC \neq$ empBConfig)$\wedge$(NzN > 1).

(c) The process has some incoming channels.

base-state((p-state[P] : PS)BC) start-state(SC) snapshot(SSC)
control((prog[P] : notYet)(#ms[P] : NzN')CC)
$\Rightarrow$
base-state((p-state[P] : PS) bcast(BC, P,marker))
start-state(SC) snapshot((p-state[P] : PS) inchans(BC, P)SSC)
control((prog[P] : started)(#ms[P] : NzN')CC)
if ($SC \neq$ empBConfig).

- Receipt of Marker is split into two subcases:

  1. The process has not yet started CLDSA. This case is further split into three subcases:

     (a) The process has only one incoming channel, and there are no processes that have not yet completed CLDSA except for the process.

     base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
     snapshot(SSC) finish-state(empBConfig)
     control((prog[P] : notYet)(cnt : 1)(#ms[P] : 1)
     (done[Q, P, N] : *false*)CC)
     $\Rightarrow$
     base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
     snapshot((p-state[P] : PS)(c-state[Q, P, N] : empChan)SSC)
     finish-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
     control((prog[P] : completed)(cnt : 0)(#ms[P] : 0)(done[Q, P, N] : *true*)CC).

     (b) The process has only one incoming channel, and there are some other processes that have not yet completed CLDSA.

     base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
     snapshot(SSC) control((prog[P] : notYet)(cnt : NzN)(#ms[P] : 1)
     (done[Q, P, N] : *false*)CC)
     $\Rightarrow$
     base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)
     bcast(BC, P,maker)) snapshot((p-state[P] : PS)(c-state[Q, P, N] : empChan)SSC)
     control((prog[P] : completed)(cnt : sd(NzN, 1))(#ms[P] : 0)
     (done[Q, P, N] : *true*)CC)
     if NzN $>$ 1.

     (c) The process has more than one incoming channels.

     base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
     snapshot(SSC) control((prog[P] : notYet)(cnt : NzN)
     (#ms[P] : NzN')(done[Q, P, N] : *false*)CC)
     $\Rightarrow$
     base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)
     bcast(BC, P,maker)) snapshot((p-state[P] : PS)
     (c-state[Q, P, N] : empChan)inchans(BC, P)SSC)
     control((prog[P] : started)(cnt : sd(NzN, 1))(#ms[P] : sd(NzN', 1))
     (done[Q, P, N] : *true*)CC)
     if NzN' $>$ 1.

  2. The process has already started CLDSA. This case is further split into three subcases:

(a) There is no incoming channel from which markers have not been received except for the incoming channel, and there are no processes that have not yet completed CLDSA except for the process.

base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
finish-state(empBConfig) control((prog[P] : started)(cnt : 1)
(#ms[P] : 1)(done[Q, P, N] : *false*)CC)
$\Rightarrow$
base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
finish-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
control((prog[P] : completed)(cnt : 0)(#ms[P] : 0)
(done[Q, P, N] : *true*)CC).

(b) There are no incoming channels from which markers have not been received except for the incoming channel, and there are some other processes that have not yet completed CLDSA.

base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
control((prog[P] : started)(cnt : NzN)(#ms[P] : 1)
(done[Q, P, N] : *false*)CC)
$\Rightarrow$
base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
control((prog[P] : completed)(cnt : sd(NzN, 1)(#ms[P] : 0)
(done[Q, P, N] : *true*)CC)
if NzN $>$ 1.

(c) There are some other incoming channels from which markers have not been received.

base-state((p-state[P] : PS)(c-state[Q, P, N] : marker | MMS)BC)
control((prog[P] : started)(cnt : NzN)(#ms[P] : NzN')
(done[Q, P, N] : *false*)CC)
$\Rightarrow$
base-state((p-state[P] : PS)(c-state[Q, P, N] : MMS)BC)
control((prog[P] : started)(cnt : NzN)(#ms[P] : sd(NzN', 1))
(done[Q, P, N] : *true*)CC) /] if NzN' $>$ 1.

There are 18 transition rules described as above. Those transition rules are classified into three parts: UDS, UDS & CLDSA, and CLDSA. The UDS part consists of the transition rules describing the actions purely related to the UDS, namely i, ii and iii-1. The UDS part depends on the UDS concerned, can be constructed from the three transition rules of the UDS and changes the base-state meta configuration component of a state of M_{CLDSA}. The UDS & CLDSA part also depends on the UDS concerned and can be constructed from the three transition rules of the UDS, but changes the other meta configuration components of a state of M_{CLDSA} as well. Three transition rules describing actions iii-2, iii-3 and iii-4 are in the UDS & CLDSA part. The CLDSA part is independent from the UDS concerned, can be constructed regardless of any UDSs, and does not change the base-state meta configuration com-

ponent of a state of M_{CLDSA}. The transition rules describing two kinds of actions iv and v are in the CLDSA part.

Definition 4 (TR_{CLDSA})**.** Let TR_{CLDSA} be the set of all ground instances of the 18 transition rules.

3.2.3 State Machine M_{CLDSA}

We propose the function CL that takes a state machine M_{UDS} and returns another state machine M_{CLDSA}. Note that M_{UDS} is the state machine of a UDS and M_{CLDSA} is the state machine of the UDS-CLDSA. Since the authors in [18] treat the states of processes and channels of the UDS superimposed by CLDSA as the same as those of a UDS, the definition of function CL in [18] is incorrect. The definition is as follows:

> For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$, $\mathrm{CL}(S_U DS) = \{\text{base-state}(bs)\ \text{start-state}(ss)\ \text{snapshot}(sss)©\ \text{f-state(fs)}\ \text{control}(ctl) \mid bs \in S_U DS,\ ss \in \text{Config},\ sss \in \text{Config},\ fs \in \text{Config},\ ctl \in \text{CtlConfig}\}$, where Config and CtlConfig are used as the sets of terms whose sorts are Config and CtlConfig, respectively.

They consider that bs is in $S_U DS$. Obviously, this is incorrect since the channels in bc may contain markers, which do not exist in the channels of a UDS. Hence, bs cannot be in $S_U DS$. We redefine the function CL as follows:

Definition 5 ($\mathrm{CL}(M_{UDS})$)**.** For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$ formalizing a UDS, CL is the function that takes M_{UDS} and returns $\mathrm{CL}(M_{UDS}) \triangleq \langle \mathrm{CL}_{State}(S_U DS), \mathrm{CL}_{Init}(I_{UDS}), \mathrm{CL}_{Trans}(T_{UDS}) \rangle$, where

1. $\mathrm{CL}_{State}(S_U DS)$ is the set of all ground constructor terms of sort MBConfig;
2. $\mathrm{CL}_{Init}(I_{UDS})$ is $\{$base-state(bc) start-state(empBConfig) snapshot(©empBConfig) finish-state(empBConfig) control(ctl) $\mid$ bc $\in I_{UDS}$, ctl = InitCtlConfig(bc)$\}$;
3. $\mathrm{CL}_{Trans}(T_{UDS}) \subseteq \mathrm{CL}_{State}(S_U DS) \times \mathrm{CL}_{State}(S_U DS)$ is $\{(MC, MC') \mid MC, MC' \in \text{MBConfig}, L \Rightarrow R \in TR_{CLDSA}, \exists MC''.(MC = LMC'' \wedge MC' = RMC'')\}$.

Function InitCtlConfig(bc) initializes values for all control information components. Let M_{CLDSA} be $\mathrm{CL}(M_{UDS})$. Note that S_{CLDSA} is the set of all ground constructor terms of sort MBConfig, although each reachable state from an initial state in S_{CLDSA} is in the following form:

> base-state(bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl).

Some functions on S_{CLDSA} are defined for convenience.

Definition 6 (b-state, s-state, snapshot, f-state, finished)**.** For each $s \in S_{CLDSA}$,

- b-state(s) is bc if there exists exactly one occurrence of the base-state(bc) meta configuration component in s and empBConfig otherwise,
- s-state(s) is sc if there exists exactly one occurrence of the start-state(sc) meta configuration component in s and empBConfig otherwise,
- snapshot(s) is ssc if there exists exactly one occurrence of the snapshot(ssc) meta configuration component in s and empBConfig otherwise,
- f-state(s) is fc if there exists exactly one occurrence of the finish-state(fc) meta configuration component in s and empBConfig otherwise, and
- finished(s) is false if f-state(s) is empBConfig and *true* otherwise.

The following is the definition that CLDSA has terminated in a state s in M_{CLDSA}:

Definition 7 ($M_{CLDSA} \models$ terminated(s))**.** For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$, for each $s \in S_{CLDSA}$, $M_{CLDSA} \models$ terminated(s) if and only if finished(s).

In the rest of the paper, terminated is abbreviated as trmtd. We have the following proposition on M_{CLDSA}:

Proposition 1 (No marker in s-state, snapshot and f-state)**.** For each $s \in S_{CLDSA}$, if $M_{CLDSA} \models$ trmtd(s), then there is no marker in s-state(s), snapshot(s) and f-state(s), equivalently that the least sort of s-state(s), snapshot(s) and f-state(s) are ConFigure.

Note that, whenever CLDSA has terminated in a state s, the function s-state(s), snapshot(s) and f-state(s) return the start state, the snapshot and the finish state, respectively.

4 A MORE FAITHFUL DEFINITION OF THE DSR PROPERTY

4.1 The Informal Description of the DSR Property

The informal description of the DSR Property is given in [13] as follows. Let s_1, s_* and s_2 be the state in which CLDSA initiates, the snapshot taken, and the state in which CLDSA terminates, respectively. Although the snapshot s_* may not be identical to any of the global states that occur in the computation from s_1 to s_2, one desired property (called the DSR property) CLDSA should satisfy is that s_* is reachable from s_1 and s_2 is reachable from s_*, whenever CLDSA terminates. Note that s_1, s_2 and s_* are states of the UDS, but not those of the UDS-CLDSA.

4.2 Formal Definition of the DSR Property

Infinite sequences of states called paths are generated from a state machine because T is total. Paths are defined as follows.

Definition 8 (Path)**.** A path π of a state machine $M \triangleq \langle S, I, T\rangle$ from a state s_0 is an infinite sequence of states $\pi \triangleq (s_0, s_1, s_2, \dots)$, where $(\forall i \geq 0)((s_i, s_{i+1}) \in T)$. π_i denotes the ith state (i.e., s_i) in π and Π denotes the set of all paths of M.

For a state machine M, $M, \pi \models \text{isReachable}(s_2, s_1)$ if and only if s_2 is reachable from s_1 in a path π in M and then $M \models \text{isReachable}(s_2, s_1)$ if and only if s_2 is reachable from s_1 in M.

Definition 9 (Reachabilty in M)**.** For a state machine $M \triangleq \langle S, I, T\rangle$, for each $\pi \in \Pi$ and each $s_1, s_2 \in S$, $M, \pi \models \text{isReachable}(s_2, s_1)$ if and only if $(\exists i, j \in Nat)\,(i \leq j \wedge s_1 = \pi_i \wedge s_2 = \pi_j)$, and $M \models \text{isReachable}(s_2, s_1)$ if and only if $(\exists \pi \in \Pi)\,(M, \pi \models \text{isReachable}(s_2, s_1))$.

In other words, a state s_2 is said to be reachable from a state s_1 if and only if s_1 can go to s_2 by zero or more state transition steps in the state machine M.

In the informal description of the DSR property, it is checked that CLDSA terminates, and it is checked that some states of a UDS are reachable from some others in the UDS but not the UDS-CLDSA. Accordingly, the property involves two systems, a UDS and the UDS-CLDSA, and hence we need to use two state machines M_{UDS} and M_{CLDSA} to faithfully define the DSR property. Our definition of the DSR property is as follows.

Definition 10 (The DSR Property)**.** For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS}\rangle$, $(\forall s \in S_{CLDSA})\,(M_{CLDSA} \models \text{trmtd}(s) \Rightarrow M_{UDS} \models \text{isReachable}(s_*, s_1) \wedge M_{UDS} \models \text{isReachable}(s_2, s_*))$, where $s_1 = \text{s-state}(s)$, $s_* = \text{snapshot}(s)$ and $s_2 = \text{f-state}(s)$.

5 THE THEOREM ON EQUIVALENCE OF THE TWO DEFINITIONS OF THE DSR PROPERTY

Since our new definition of the DSR property is more likely to faithfully express the informal description of the property, it can be used to more faithfully model check that CLDSA enjoys the property. However, due to involving two state machines, it is not straightforward to directly model check the new definition with an existing model checker. This is because existing temporal logics, such as LTL and CTL, used for model checking, only consider one state machine, more precisely one Kripke structure. Because the existing definition of the DSR property has been model checked in [16], the equivalence of the new definition and the existing one guarantees that we can use the existing model checking approach to model checking for the new definition. Therefore, we prove a theorem saying that our new definition is equivalent to the existing definition, which also confirms the validity of the existing model checking approach.

Although the DSR property is encoded in terms of the Maude search command in the existing study, the existing definition can be represented in terms of state machines. Let us suppose that there are n processes in a UDS and let $p_1, \dots, p_n$

be their identifications, namely that Pid is $\{p_1, \ldots, p_n\}$, where $n \geq 1$. Let ctl be $(\text{prog}[p_1] : \text{notYet}) \ldots (\text{prog}[p_n] : \text{notYet})$ in the rest of the paper. The existing definition of the DSR property is represented in terms of state machines as follows:

> For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$, $(\forall s \in S_{CLDSA})\,(M_{CLDSA} \models \text{trmtd}(s) \Rightarrow M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_*)\,\text{control}(\text{ctl}), \text{base-state}(s_1)\,\text{control}(\text{ctl})) \wedge M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_2)\,\text{control}(\text{ctl}), \text{base-state}(s_*)\,\text{control}(\text{ctl})))$, where $s_1 = \text{s-state}(s)$, $s_* = \text{snapshot}(s)$ and $s_2 = \text{f-state}(s)$.

Both of the definitions are checking the termination of CLDSA in M_{CLDSA}. However, the reachability is checked in the other state machine M_{UDS} in the new definition, while it is checked in the same state machine M_{CLDSA} in the existing definition. This is the key difference between the two definitions. Although the two definitions are seemingly different, we realize that the new one coincides with the existing one [16]. Hence we prove the following theorem saying that two definitions are equivalent.

Theorem 1 (Equivalence of the Two Definitions)**.** For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$, $(\forall s \in S_{CLDSA})\,(M_{CLDSA} \models \text{trmtd}(s) \Rightarrow M_{UDS} \models \text{isReachable}(s_*, s_1) \wedge M_{UDS} \models \text{isReachable}(s_2, s_*)) \Leftrightarrow (M_{CLDSA} \models \text{trmtd}(s) \Rightarrow M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_*)\,\text{control}(\text{ctl}), \text{base-state}(s_1)\,\text{control}(\text{ctl})) \wedge M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_2)\,\text{control}(\text{ctl}), \text{base-state}(s_*)\,\text{control}(\text{ctl})))$, where $s_1 = \text{s-state}(s)$, $s_* = \text{snapshot}(s)$ and $s_2 = \text{f-state}(s)$.

The only difference between the new definition and the existing one is the conclusion part of the implications, in which the different state machines are used to check the reachability in each definition. If we can prove that the conclusion parts are equivalent, then the two definitions are equivalent. The equivalence of the conclusion parts means that reachability is preserved between M_{UDS} and M_{CLDSA}. Therefore, to prove Theorem 1, we prove Lemma 1 on reachability preservation. Lemma 1 asserts that reachability is preserved between M_{UDS} and M_{CLDSA}. The lemma is as follows.

Lemma 1 (Reachability Preservation)**.** For a state machine $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$, $(\forall s_1, s_2 \in S_U DS)\,(M_{UDS} \models \text{isReachable}(s_2, s_1) \Leftrightarrow M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_2)\,\text{control}(\text{ctl}), \text{base-state}(s_1)\,\text{control}(\text{ctl})))$.

We first prove as Lemma 2 and Lemma 3 that one-step reachability is preserved between M_{UDS} and M_{CLDSA} to prove Lemma 1. The two lemmas are as follows.

Lemma 2 (One-step Reachability Preservation from M_{UDS} to M_{CLDSA})**.** $\forall s_1, s_2 \in S_U DS$ such that s_1 goes to s_2 with one state transition step in M_{UDS}, base-state(s_1) control(ctl) goes to base-state(s_2) control(ctl) with one state transition step in M_{CLDSA}.

Lemma 3 (One-step Reachability Preservation from M_{CLDSA} to M_{UDS})**.** $\forall$ s_1, s_2 $\in$ S_UDS such that base-state(s_1) control(ctl) goes to base-state(s_2) control(ctl) with one state transition step in M_{CLDSA}, s_1 goes to s_2 with one state transition step in M_{UDS}.

For each UDS, T_{UDS} is constructed from the three transition rules and T_{CLDSA} is constructed from the 18 transition rules. Therefore, all we have to do is to take into account the three transition rules and the 18 transition rules to discuss T_{UDS} and T_{CLDSA}, respectively. In the following proofs, $p, q \in$ Pid, $ps_1, ps_2 \in$ PState, $cs \in$ MsgQueue, $m \in$ Msg, $bc \in$ Config and $n \in$ Nat are fresh constants of those sorts.

Proof. (Proof Sketch of Lemma 2.) Assume that s_1 goes to s_2 by a state transition t in M_{UDS}. Our proof shows that there exists a state transition t' in M_{CLDSA} that moves base-state(s_1) control(ctl) to base-state(s_2) control(ctl). Let us consider the case in which t is constructed from the transition rule that describes Sending of Message in M_{UDS}. It suffices to consider s_1 as an arbitrary state (p-state[p] : ps_1) (c-state[p, q, n] : cs)bc in S_UDS to which the transition rule can be applied. Therefore, s_2 is (p-state[p] : ps_2) (c-state[p, q, n] : enq(cs, m))bc. Then, base-state(s_1) control(ctl) is base-state((p-state[p] : ps_1)(c-state[p, q, n] : cs)bc) control(ctl), and base-state(s_2) control(ctl) is base-state((p-state[p] : ps_2) (c-state[p, q, n] : enq(cs, m)) bc) control(ctl). The transition rule that describes Sending of Message in M_{CLDSA} can be applied to base-state(s_1) control(ctl) and obtains base-state(s_2) control(ctl). Hence, there exists t'. The case has been discharged. We can deal with the other two cases that correspond to Change of Process State and Receipt of Message, respectively. □

Proof. (Proof Sketch of Lemma 3.) Assume that base-state(s_1) control(ctl) goes to base-state(s_2) control(ctl) by a state transition t in M_{CLDSA}. Because $s_1 \in S_UDS$, there is no marker in s_1. Moreover, ctl is (prog[p_1] : notYet) ... (prog[p_n] : notYet). This is why any of the transition rules that describe Record of Process State and Receipt of Marker in M_{CLDSA} cannot be applied to base-state(s_1) control(ctl). Therefore, t is not a state transition constructed from those transition rules. Any of the transition rules that describe the 2nd, 3rd and 4th sub-cases of Receipt of Message in M_{CLDSA} cannot be applied to base-state(s_1) control(ctl), neither. Therefore, t is not a state transition constructed from those transition rules, neither. Then, all we have to do is to consider the transition rules that describe Change of Process State, Sending of Message and the 1st part of Receipt of Message in M_{CLDSA}. The same proof strategy used in the proof of Lemma 2 can be used to show that there exists a state transition that moves s_1 to s_2 in M_{UDS} for each state transition that moves base-state(s_1) control(ctl) to base-state(s_2) control(ctl) in M_{CLDSA}. □

Proof. (Proof of the "if" part of Lemma 1.) We prove that $\forall s_1, s_2 \in S_UDS$, if $M_{CLDSA} \models$ isReachable(base-state(s_2) control(ctl), base-state(s_1) control(ctl)), then $M_{UDS} \models$ isReachable(s_2, s_1).

Assume that $M_{CLDSA} \models$ isReachable(base-state(s_2) control(ctl), base-state(s_1) control(ctl)) and then there must be a natural number k such that base-state(s_1) control(ctl) goes to base-state(s_2) control(ctl) by k state transition steps in M_{CLDSA}. The proof is done by induction on k.

Base case: Since base-state(s_1) control(ctl) is the same as base-state(s_2) control(ctl) in this case, s_1 is the same as s_2. So, this case is discharged.

Induction case: Suppose that base-state(s_1) control(ctl) moves to base-state(s_2) control(ctl) by $k+1$ transition steps and the $k+1$ transitions taken are $t_1, \ldots, t_{k+1}$. As shown in Figure 5, base-state(s') control(ctl) is the state to which base-state(s_1) control(ctl) moves by the first k transition steps, namely that $M_{CLDSA} \models$ isReachable(base-state(s') control(ctl), base-state(s_1) control(ctl)). From the induction hypothesis, $M_{UDS} \models$ isReachable(s', s_1). Since base-state(s') control(ctl) moves to base-state(s_2) control(ctl) by one transition step in M_{CLDSA}, s' also moves to s_2 by one transition step in M_{UDS} from Lemma 3. Then, this case is also discharged. Figure 5 shows the correspondence between the transitions in M_{CLDSA} and M_{UDS}.

□

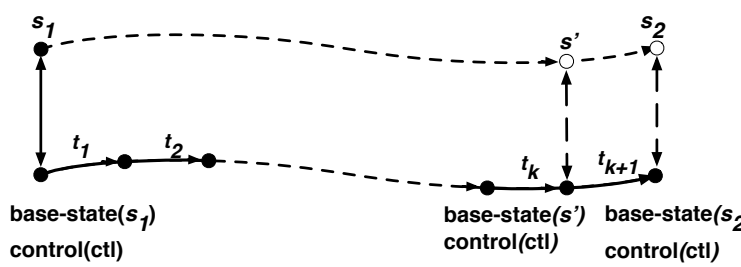


Figure 3. The correspondence between the transitions in M_{CLDSA} and M_{UDS}

Proof. (Proof Sketch of the "only if" part of Lemma 1.) We prove that $\forall s_1, s_2 \in S_U DS$, if $M_{UDS} \models$ isReachable(s_2, s_1), then ($M_{CLDSA} \models$ isReachable(base-state(s_2) control(ctl), base-state(s_1) control(ctl)).

Assume that $M_{UDS} \models$ isReachable(s_2, s_1) and then there must exist a natural number k such that s_1 goes to s_2 by k state transition steps in M_{UDS}. The proof is done by induction on k. Note that Lemma 2 is used in this proof. □

Proof. (Proof of Theorem 1.) Proof of Theorem 1 follows from Proposition 1 and Lemma 1. □

6 CLDSA DOES NOT ALTER THE BEHAVIORS OF A UDS

As control algorithms [21], DSAs should run concurrently but not interfere with the behaviors of a UDS. The behaviors of the algorithms are transparent to a UDS. It is

necessary to prove that CLDSA does not alter the behaviors of a UDS to guarantee the correctness of the algorithm. We will prove that any original actions of each process of a UDS, namely sending a message, receiving a message and changing its state, are preserved by CLDSA.

If we prove that M_{CLDSA} simulates M_{UDS} and vice versa, we can state that the behaviors of a UDS are preserved by CLDSA. We propose a binary relation **r** between M_{UDS} and M_{CLDSA}, and then prove Theorem 1 saying that **r** is a bi-simulation relation between M_{UDS} and M_{CLDSA}. In the following part, for all states s, s' in state machine M, $s \rightsquigarrow_M s'$ denotes that state s moves to states s' by one state transition of M, and $s \rightsquigarrow_M^* s'$ denotes that state s moves to state s' by zero or more state transitions of M. Simulation from one state machine to another is defined as follows:

Definition 11 (Simulation from M_A to M_B)**.** Given two state machines $M_A \triangleq \langle S_A, I_A, T_A \rangle$ and $M_B \triangleq \langle S_B, I_B, T_B \rangle$, $r : S_A\ S_B \rightarrow$ Bool is called a *simulation* from M_A to M_B if it satisfies the following conditions:

1. For each $s_A \in I_A$ there exists $s_B \in I_B$ such that $r(s_A, s_B)$.
2. For each $s_A, s'_A \in S_A$ and $s_B \in S_B$ such that $r(s_A, s_B)$ and $s_A \rightsquigarrow_{M_A} s'_A$, there exists $s'_B \in S_B$ such that $r(s'_A, s'_B)$ and $s_B \rightsquigarrow_{M_B}^* s'_B$.

r is a bi-simulation if and only if it is a simulation from M_A to M_B and vice versa.

We recognize that with the exception of putting markers into the channels of a UDS, the algorithm does not change any original behavior of processes in the system. We propose a binary relation **r** between M_{UDS} and M_{CLDSA} saying that for each $s_1 \in S_U DS$ and each $s_2 \in S_{CLDSA}$, $\mathbf{r}(s_1, s_2)$ if and only if s_1 is the same as the state obtained by deleting all markers from s_2. The functions to delete all markers from one state of a UDS on which CLDSA is superimposed is implemented as the function delM as follows:

op delM: BConfig $\rightarrow$ Config.

eq delM(empBConfig) = empConfig.

eq delM((p-state$[P] : PS)$ BCF) = (p-state$[P] : PS)$delM(BCF).

eq delM((c-state$[P, Q, N] : MMS)BCF) =$

(c-state$[P, Q, N]$: delMchan(MMS)) delM(BCF).

Where function delMchan deletes all markers in a sequence of messages.

The binary relation **r** is defined as follows:

Definition 12 (Binary relation **r**)**.** Given two state machines $M_{UDS} \triangleq \langle S_U DS, I_{UDS}, T_{UDS} \rangle$ and $M_{CLDSA} \triangleq \langle S_{CLDSA}, S_{CLDSA}, T_{CLDSA} \rangle, \forall s_{UDS} \in S_U DS$ and $\forall s_{CLDSA} \in S_{CLDSA}$, the binary relation $\mathbf{r} : S_U DS S_{CLDSA} \rightarrow$ Bool is defined as follows:

$$\mathbf{r}(s_{UDS}, s_{CLDSA}) \triangleq (s_{UDS} = delM(\text{b-state}(s_{CLDSA}))).$$

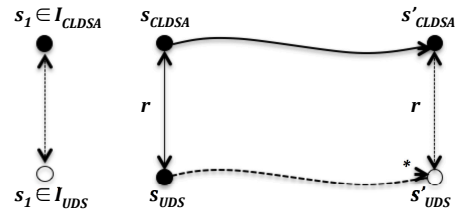


Figure 4. The binary relation **r** is a simulation from M_{CLDSA} to M_{UDS}

Theorem 2 (Bi-simulation relation **r**)**.** Binary relation **r** is a bi-simulation relation between M_{UDS} and M_{CLDSA}.

We will prove that **r** is a simulation from M_{CLDSA} to M_{UDS} and vice versa. It suffices to only consider states reachable from the initial states in the proof. Initial states have a specific form of configuration and no observable component will be added to and/or deleted from initial states by any transition. The proof uses this fact.

Simulation from M_{CLDSA} to M_{UDS}. We will prove that **r** satisfies the following conditions. Figure 6 shows the diagrams corresponding to the two conditions.

Condition 1. For each $s_{CLDSA} \in S_{CLDSA}$ there exists $s_{UDS} \in I_{UDS}$ such that $\mathbf{r}(s_{UDS}, s_{CLDSA})$.

Proof. For each $s_{CLDSA} \in \mathrm{CL}_{Init}(I_{UDS})$, according to the definition of CL, s_{CLDSA} is in form of base-state(bc) start-state(empBConfig) snapshot(empBConfig) finsh-state(empBConfig) control(ctl), where $bc \in I_{UDS}$. Since b-state(s_{CLDSA}) = bc and $bc \in I_{UDS}$, let us choose s_{UDS} is bc. Because delM(b-state(s_{CLDSA})) = bc, $s_{UDS} = delM(b - state(s_{CLDSA}))$. We have $\mathbf{r}(s_{UDS}, s_{CLDSA})$. This condition is satisfied. □

Condition 2. For each s_{CLDSA}, $s'_{CLDSA} \in S_{CLDSA}$ and $s_{UDS} \in S_UDS$ such that $\mathbf{r}(s_{UDS}, s_{CLDSA})$ and $s_{CLDSA} \rightsquigarrow_{M_{CLDSA}} s'_{CLDSA}$, there exists s'_{UDS} such that $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and $s_{UDS} \rightsquigarrow^*_{M_{UDS}} s'_{UDS}$.

Proof. (Proof sketch.) The configuration of a state of M_{CLDSA} is as follows.

base-state(bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl),

where $bc, sc, ssc, fc \in$ BConfig and ctl $\in$ CtlConFigure.

Because of $\mathbf{r}(s_{UDS}, s_{CLDSA})$, s_{UDS} = delM(b-state(s_{CLDSA})) = delM(bc). Let us assume that $s_{CLDSA} \rightsquigarrow_{M_{CLDSA}} s'_{CLDSA}$ by state transition t. The same as what we have mentioned above, we only take into account the three transition rules and the 18 transition rules to discuss T_{UDS} and T_{CLDSA}, respectively. Because the 18 transition rules are classified into three parts: UDS, UDS & CLDSA, and CLDSA, the state transitions can be also classified into the three parts. In what follows,

$p, q \in$ Pid, $ps_1, ps_2 \in$ PState, $cs \in$ MsgQueue, $m \in$ Msg, $bc, sc, ssc \in$ BConfig, $ctl \in$ CtlConfig and $n \in$ Nat are fresh constants of those sorts.

1. Let us consider the first case in which t is in the UDS part and the UDS & CLDSA part.

   Our proof shows that for any state transition t in the UDS and the UDS & CLDSA parts that moves s_{CLDSA} to s'_{CLDSA}, there exists s'_{UDS} to which s_{UDS} moves by one state transition such that $\mathbf{r}(s'_{CLDSA}, s'_{UDS})$ holds. We can find a state transition t' in M_{UDS} that can move $s_{UDS} = \text{delM}(\text{b-state}(s_{CLDSA}))$ to $s'_{UDS} = \text{delM}(\text{b-state}(s'_{CLDSA}))$. The existence of t' corresponding to t is shown in Figure 5 a).

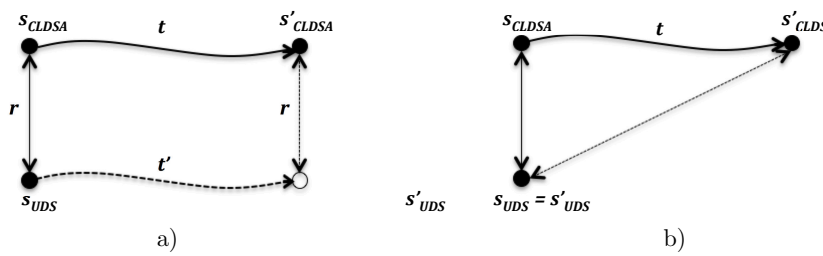


Figure 5. Existing t' in M_{UDS} corresponding to t

   Let us consider the case in which t is constructed from the transition rule that describes Change of Process State in M_{CLDSA}. It suffices to consider s_{CLDSA} as base-state((p-state[p] : $ps1$)bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl) to which the transition rule can be applied. Because of $\mathbf{r}(s_{UDS}, s_{CLDSA})$, $s_{UDS} = \text{delM}((\text{p-state}[p] : ps1)bc) = (\text{p-state}[p] : ps1)\ \text{delM}(bc)$ from the definition of function delM. Since s_{CLDSA} goes to s'_{CLDSA} by t, s'_{CLDSA} is base-state((p-state[p] : $ps2$)bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl). Let t' be the state transition that is constructed from the transition rule that describes Change of Process State in M_{UDS}. Let s'_{UDS} be (p-state[p] : $ps2$) delM(bc). Then s_{UDS} can move to s'_{UDS} by t'. Because $s'_{UDS} = (\text{p-state}[p] : ps2)\ \text{delM}(bc)$ and $\text{delM}(\text{b-state}(s'_{CLDSA})) = (\text{p-state}[p] : ps2)\ \text{delM}(bc)$, $s'_{UDS} = \text{delM}(\text{b-state}(s'_{CLDSA}))$. Therefore, $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and $s_{UDS} \rightsquigarrow_{M_{UDS}} s'_{UDS}$ by t'. The case has been discharged. We can deal with the other two cases that correspond to Sending of Message and the 1st of Receipt of Message, respectively, likewise.

2. The last case in which t is constructed from the transition rule in the CLDSA part.

   Since CLDSA part does not change the base-state meta configuration component of a state of M_{CLDSA}. Our proof shows that we can choose as s'_{UDS} the same as s_{UDS} then s_{UDS} goes to s'_{UDS} by zero step and $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$. This is shown in Figure 5 b).

From what have been proved above, we can see that relation **r** satisfies the two conditions of simulation from M_{CLDSA} to M_{UDS}. Therefore, **r** is a simulation relation from M_{CLDSA} to M_{UDS}. □

Simulation from M_{UDS} to M_{CLDSA} We will prove that **r** is a simulation from M_{UDS} to M_{CLDSA}.

Proof. (Proof Sketch.) Our proof shows that **r** satisfies the two conditions of simulation from M_{CLDSA} to M_{UDS}. It is straightforward to show that for each s in I_{UDS} there exists s' in I_{CLDSA} such that $\mathbf{r}(s, s')$ holds. To show the second condition, we need to consider the three (kinds of) transition rules of M_{UDS} and then it suffices to consider the rules of the UDS part and some rules of the UDS & CLDSA part in M_{CLDSA}. The proof can be conducted like we have done for the proof of simulation from M_{CLDSA} to M_{UDS}. □

7 RELATED WORK

Many researches [10, 11, 12] have been conducted to formally verify various distributed systems. Among them, [10] concentrates on model checking for distributed systems. The main contribution of the research is the design and implementation of the fair linear temporal logic of rewriting (LTLR) model checker, a model checker under localized fairness assumptions for Maude system. LTLR is an extension of LTL. So the Fair LTLR model checker is basically an extension of Maude LTL model checker dealing with fairness assumptions. Although the model checker tries to deal with several distributed algorithms, it has not yet considered DSAs. The authors in [11] deal with the problem of verification of asynchronous consensus algorithms, a fault-tolerant distributed algorithm. The challenge of the problem is that the state space is huge. Dealing with this problem, they have proposed a semi-automatic verification approach based on model checking technique. In their approach the problem of verification of asynchronous consensus algorithms is reduced to small model checking problems, namely the set of bounded model checking problems that can be solved efficiently by using bounded model checking with an SMT (Satisfiability Modulo Theories) solver. In detail, they adopt a round-based model called the Heard-Of (HO) model [20] to alleviate the problem. Their method can be used to model check several consensus algorithms up to around 10 processes. However, the method can only be applied to some consensus algorithms but not to DSAs.

CLDSA and its desired properties were initially introduced in [13]. In this, the DSR property is given in an informal way. Several studies are motivated by verification of snapshot algorithms. Among them, [7, 12] consider directly CLDSA. In [7], CLDSA is modelled in PROMELA, and then the model is simplified to be verifiable. However, only the UDS-CLDSA is modelled, and a property that is different from the DSR property is model checked for CLDSA. The authors in [12] focus on developing snapshot algorithms with formal proofs that guarantee the correctness

of the algorithms. Some existing snapshot algorithms, such as CLDSA and Lai-Yang, are re-developed by using the Event B framework and refinement. Starting with a model providing an abstract view of a system and its behaviors, the model then is enriched more concretely by many refinement steps to derive the algorithms. To capture the complete and desired behaviors of snapshot algorithms, each refinement step must preserve essential desired properties ensuring a consistent cut. The properties are implemented as invariant conditions. This is also to ensure that the snapshot recorded by the deriving algorithm is consistent. Their experiments are conducted on fixed networks. Moreover, any of the properties they have considered are not necessarily the same as the DSR property.

8 CONCLUSION

The authenticity of a model checking relies on the faithfulness of the specifications of desired properties. It is expected that the desired properties are faithfully expressed in the specifications. Attempting to more faithfully model check the DSR property for CLDSA, we have given a more faithful formal definition of the DSR property. Our definition involves two state machines in which the termination is checked in the state machine M_{CLDSA} formalizing the UDS-CLDSA and the reachability is checked in the other state machine M_{UDS} formalizing a UDS. The checking of the reachability is different between the new definition and the existing definition. To guarantee that it suffices to model check the definition used in the existing study for CLDSA and the existing model checking approach can be used for this end, we have proved Theorem 1 saying that our formalization of the DSR property is equivalent to the existing one for each M_{UDS}. Moreover, we have proved Theorem 2 saying that M_{CLDSA} simulates M_{UDS} and vice versa to guarantee that CLDSA does not alter the behaviors of a UDS.

In the existing work [16], it is necessary to specify the UDS on which CLDSA is superimposed for each UDS in Maude to model check that the UDS on which CLDSA is superimposed enjoys the DSR property with the Maude search command. Moreover, the way to model check means to compare the numbers of solutions obtained by three search experiments. Therefore, it is not straightforward to construct a counterexample when the property is not fulfilled. The specification techniques described in the paper make it possible to specify CLDSA as a meta-program in Maude. Such a meta-program as a specification of CLDSA takes a concrete UDS as an actual parameter and generates the specification of the UDS-CLDSA. It is also possible to directly model check the faithful formalization of the DSR property based on the specification of the UDS-CLDSA and construct a counterexample if the property is not fulfilled.

Acknowledgement

This work was partially supported by Kakenhi 23220002, 26240008, 30272991. The authors are grateful to the anonymous reviewers who carefully read an earlier version

of the paper and gave us useful comments that made it possible for us to complete the present paper.

REFERENCES

[1] Coulouris, G. F.—Dollimore, J.—Kindberg, T.—Blair, G.: Distributed Systems: Concepts and Design. 5th Edition. Addison-Wesley, 2011.

[2] Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer, 2013, doi: 10.1007/978-3-642-38123-2.

[3] Rittinghouse, J. W.—Ransome, J. F.: Cloud Computing: Implementation, Management, and Security. CRC Press, 2009.

[4] Spink, A.—Zimmer, M. (Eds.): Web Search: Multidisciplinary Perspectives. Springer, Information Science and Knowledge Management, Vol. 14, 2008, doi: 10.1007/978-3-540-75829-7.

[5] Baier, C.—Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008.

[6] Clarke, E. M.—Grumberg, O.—Peled, D. A.: Model Checking. MIT Press, 1999.

[7] Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.

[8] Cimatti, A.—Clarke, E.—Giunchiglia, E.—Giunchiglia, F.—Pistore, M.—Roveri, M.—Sebastiani, R.—Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K. G. (Eds.): Computer Aided Verification (CAV 2002). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2404, 2002, pp. 359–364, doi: 10.1007/3-540-45657-0_29.

[9] Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional, 2002.

[10] Konnov, I.—Veith, H.—Widder J.: On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. In: Baldan, P., Gorla, D. (Eds.): CONCUR 2014 – Concurrency Theory. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 8704, 2014, pp. 125–140, doi: 10.1007/978-3-662-44584-6_10.

[11] Tsuchiya, T.—Schiper, A.: Verification of Consensus Algorithms Using Satisfiability Solving. Distributed Computing, Vol. 23, 2011, No. 5-6, pp. 341–358, doi: 10.1007/s00446-010-0123-3.

[12] Andriamiarina, M. B.—Méry, D.—Singh, N. K.: Revisiting Snapshot Algorithms by Refinement-Based Techniques. Computer Science and Information Systems, Vol. 11, 2014, No. 1, pp. 251–270, doi: 10.2298/CSIS130122007A.

[13] Chandy, K. M.—Lamport, L.: Distributed Snapshots: Determining Global States of Distributed System. ACM Transactions on Computer Systems (TOCS), Vol. 3, 1985, No. 1, pp. 63–75, doi: 10.1145/214451.214456.

[14] Spezialetti, M.—Kearns, P.: Efficient Distributed Snapshots. Proceeding of the 6th International Conference on Distributed Computing Systems (ICDCS 1986), 1986, pp. 382–388.

[15] Venkatesan, S.: Message-Optimal Incremental Snapshots. The Journal of Computer and Software Engineering, Vol. 27, 1993, pp. 211–231.

[16] Ogata, K.–Huyen, T. T. P.: Specification and Model Checking of the Chandy and Lamport Distributed Snapshot Algorithm in Rewriting Logic. In: Aoki, T., Taguchi, K. (Eds.): Formal Methods and Software Engineering (ICFEM 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7635, 2012, pp. 87-102, doi: 10.1007/978-3-642-34281-3_9.

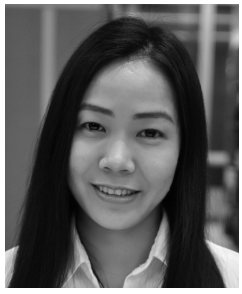
[17] Clavel, M.—Durán, F.—Eker, S.—Lincoln, P.—Martí-Oliet, N.—Meseguer, J.—Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4350, 2007, doi: 10.1007/978-3-540-71999-1.

[18] Zhang, W.—Ogata, K.—Zhang, M.: A Consideration on How to Model Check Distributed Snapshot Reachability Property. IEICE Technical Report, Vol. 114, 2015, No. 416, pp. 49–54. ISSN 0913-5685.

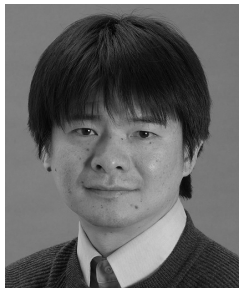
[19] Doan, H. T. T.—Zhang, W.—Zhang, M.—Ogata, K.: Model Checking Chandy-Lamport Distributed Snapshot Algorithm Revisited. Proceeding of the 2nd International Symposium on Dependable Computing and Internet of Things (DCIT), IEEE, 2015, pp. 30–39, doi: 10.1109/DCIT.2015.13.

[20] Charron-Bost, B.—Schiper, A.: Harmful Dogmas in Fault Tolerant Distributed Computing. ACM SIGACT News, Vol. 38, 2007, No. 1, pp. 53–61, doi: 10.1145/1233481.1233496.

[21] Kshemkalyani, A. D.—Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2008.



Ha Thi Thu Doan got her Ph.D. degree from the Japan Advanced Institute of Science and Technology (JAIST) in 2019. She was Lecturer at the Vietnam National University of Agriculture. She received her M.Sc. degree from the Information Science School, JAIST. Her research interests include formal methods, distributed systems, especially formal verification of distributed systems. She has been currently working on formal verification of distributed snapshot algorithms and distributed mobile robot algorithms.



Kazuhiro Ogata is Professor at the School of Information Science, Japan Advanced Institute of Science and Technology. He got his doctoral degree of engineering from the Graduate School of Science and Technology, Keio University, in 1995. Among his interesting research topics are formal methods and their application to systems, such as distributed systems.