

Title	Revisiting Model Checking of Chandy-Lamport Distributed Snapshot Algorithm
Author(s)	Doan, Thi Thu Ha
Citation	
Issue Date	2015-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/12928
Rights	
Description	Supervisor:Kazuhiro Ogata, 情報科学研究科, 修士

Revisiting Model Checking of Chandy-Lamport Distributed Snapshot Algorithm

By DOAN, Ha Thi Thu

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

Revisiting Model Checking of Chandy-Lamport Distributed Snapshot Algorithm

By DOAN, Ha Thi Thu (1310210)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

and approved by
Professor Kazuhiro Ogata
Professor Kunihiko Hiraishi
Associate Professor Toshiaki Aoki

August, 2015 (Submitted)

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to Professor Kazuhiro Ogata for his supervision, encouragement, valuable advices and kindly guidance. I sincerely appreciate his help in both my research and my daily life in JAIST. Without his consistent advices, this thesis could not have been completed. Also, I would like to thank Associate Professor Hoang Xuan Huan for his supervision in the first year of my master's program in Vietnam.

I would like to thank to Professor Kunihiro Hiraishi, Associate Professor Toshiaki Aoki and Associate Professor Xavier Defago for giving me a lot of valuable advices, comments and suggestions on my research.

I would also like thank to Min Zhang and Wenjie Zhang, who were members in Ogata Lab, for their advices and cooperation in my research and their help in my daily life.

Last but not least, I want to give a special thank to my family for their support and encouragement, and to my friends who make my social life in JAIST a pleasure experience.

Glossary

\mathcal{CLDSA} Chandy-Lamport Distributed Snapshot Algorithm.

\mathcal{UDS} Underlying Distributed System.

\mathcal{DSR} Distributed Snapshot Reachability.

$M_{\mathcal{UDS}}$ The State Machine for a \mathcal{UDS} .

$M_{\mathcal{CLDSA}}$ The State Machine for a \mathcal{UDS} Superimposed by \mathcal{CLDSA} .

Contents

1	Introduction	1
1.1	Motivation	1
1.2	The Purpose of the Research and Outcome	2
1.3	Related Work	3
1.4	Organization of the Thesis	4
2	Preliminaries	5
2.1	Underlying Distributed Systems	5
2.2	\mathcal{CLDSA} and the \mathcal{DSR} Property	6
2.3	State Machine	10
2.4	Model Cheking	11
2.5	Maude	12
2.6	Chapter Summary	14
3	Specification and Model Checking of \mathcal{CLDSA} in Maude	15
3.1	Specification of \mathcal{CLDSA}	15
3.2	Model Checking of \mathcal{CLDSA}	17
3.3	Chapter Summary	18
4	Formalizing a UDS and a UDS Superimposed by \mathcal{CLDSA} as State Machines	19
4.1	Modeling a UDS as a State Machine	19
4.1.1	State Expression for a UDS	19
4.1.2	State Transitions for M_{UDS}	21
4.1.3	State Machine M_{UDS}	23
4.2	Modeling a UDS Superimposed by \mathcal{CLDSA} as a State Machine	23
4.2.1	State Expression for $CL(M_{UDS})$	23
4.2.2	State Transitions for $CL(M_{UDS})$	27
4.2.3	State Machine $CL(M_{UDS})$	33
4.3	Chapter Summary	34
5	A More Faithful Definition of the \mathcal{DSR} Property	36
5.1	Some Definitions and Proposition 1	36
5.1.1	Some Definitions on $S_{\mathcal{CLDSA}}$	36

5.1.2	Proposition 1	36
5.2	A More Faithful Definition of the DSR Property	45
5.3	Chapter Summary	46
6	Equivalence of Two Definitions of the DSR Property	47
6.1	The Theorem on Equivalence of the Two Definitions and Some Lemmas . .	47
6.2	Proof of Lemma 2.	48
6.3	Proof of Lemma 3.	50
6.4	Proof of Lemma 1	52
6.4.1	Proof of the “if” part of Lemma 1	52
6.4.2	Proof of the “only if” part of Lemma 1	53
6.5	Proof of Theorem 1	53
6.6	Chapter Summary	54
7	$CLDSA$ does not alter the behaviors of a UDS	55
7.1	A Binary Relation Between Two State Machines M_{UDS} and M_{CLDSA} . . .	55
7.2	Theorem 2 and the Proof of It	56
7.2.1	Simulation from M_{CLDSA} to M_{UDS}	56
7.2.2	Simulation from M_{UDS} to M_{CLDSA}	76
7.3	Chapter Summary	81
8	Future Work	83
8.1	The Future Work	83
8.2	Chapter Summary	88
9	Conclusion	89
Appendix A Specification of $CLDSA$ in Maude		91
Appendix B Model Checking of $CLDSA$ in Maude		110

This dissertation was prepared according to the curriculum for the Collaborative Education Program organized by Japan Advanced Institute of Science and Technology and University of Engineering and Technology, Vietnam National University.

Chapter 1

Introduction

1.1 Motivation

In recent decades, many applications have relied on distributed systems, which involve multiple processes connected by channels. Due to the fact that distributed systems should be fault tolerant because they need to run for a long time, keeping on providing services to human beings, other systems, etc. To make distributed systems fault tolerant, it is necessary to use many non-trivial distributed algorithms, such as snapshot algorithms and checkpointing algorithms. Since many problems on distributed systems, such as recovering from faulty states and detecting stable properties can be cast to taking global snapshots of distributed systems [8], distributed snapshot algorithms are crucial.

A distributed system consists of a finite set of processes and channels. The processes communicate by sending and receiving messages through channels. A global state of a distributed system includes the states of all processes and all channels in the system, where the state of a channel is a sequence of messages sent along the channel, but not been received. Distributed snapshot algorithms help to determine global states of a distributed system (called global snapshots) during a computation. One desired property distributed snapshot algorithms should enjoy is as follows: let s_1 be the state in which a distributed snapshot algorithm concerned initiates, s_2 be the state in which the distributed snapshot algorithm terminates, and s_* be the snapshot taken, and then s_* is reachable from s_1 and s_2 is reachable from s_* , whenever the distributed snapshot algorithm terminates. The property is called the distributed snapshot reachability (*DSR*) property. Note that the snapshot may not appear in the actual computation from the start state to the finish state.

Taking global snapshots of a distributed system is not straightforward because distributed systems are asynchronous and processes do not share clocks and memory. These lead to that all processes cannot record their local states at exactly the same time. What we obtained might be inconsistent global states. Therefore, Chandy and Lamport have proposed a distributed snapshot algorithm [1] called *CLDSA* in this thesis, by which processes can record their own states and the states of communication channels such that the combination of process states and channel states form a consistent global state.

With the advancement of computer and model checking technologies, many model checkers, such as Spin [3] and NuSMV [4] have been developed and applied to formal verification of various kinds of software and hardware systems. To the best of our knowledge, however, application of model checking to formal verification of distributed snapshot algorithms have not been fully investigated. Since distributed snapshot algorithms are non-trivial, the problem to model check that the distributed snapshot algorithms enjoys the DSR property is not straightforward. There is an existing study [2] in which Maude [5] has been used to model check that $CLDSA$ enjoys the DSR property. Maude is a specification and programming language system based on rewriting logic, and equipped with model checking facilities: the LTL model checker and the search command. The search command is used in the existing study. The authors of the existing study, however, do not discuss whether their definition of the DSR property faithfully expresses the property. In addition, we recognize that the informal description of the DSR property involves both an underlying distributed system (UDS) and the UDS superimposed by $CLDSA$, while the DSR property encoded in terms of the Maude search command involves only the UDS superimposed by $CLDSA$ in the existing study. Consequently, we do not think that the existing study provides the good foundation to guarantee that the DSR property has been sufficiently model checked.

Therefore, it is necessary to faithfully express the DSR property, and then consider the similarities between the new formal definition and the existing one. Moreover, one significant characteristic of distributed snapshot algorithms is that distributed snapshot algorithms should run concurrently with but not alter the UDS . Since this property relate to computation and the behaviors of a UDS , however, it is not straightforward to prove that a distributed snapshot algorithm satisfies this property manually. The authors in [1] asserts that $CLDSA$ satisfy this property, but they have not yet proved it. Until now, we have not yet found out any reliable studies, in which this property is proven. In addition, this property is not mentioned in the existing study. But, we should prove that $CLDSA$ does not alter the behaviors of a UDS .

1.2 The Purpose of the Research and Outcome

The purpose of our research is to revisit model checking of $CLDSA$. Carefully investigating the informal description of the DSR property, the way to faithfully express the DSR property has been given in [6]. However, the study has not yet completed. The technical report [6] has not been reviewed. Our research complete [6] giving a formal definition of the DSR property, which is more likely to faithfully express the informal description. Then we prove that the new definition of the DSR property is equivalent to the definition in the existing study to confirm the validity of the model checking approach used in the existing study. Since the new definition involves both a UDS and the UDS superimposed by $CLDSA$, it is not straightforward to directly model check the new definition with any existing model checker. Hence the equality of the two definitions also asserts that we can use the model checking approach used in the existing study to model check the new definition. Moreover, we prove that $CLDSA$ does not alter the behaviors

of a UDS .

Since state machines are suitable to formalize concurrent systems including distributed systems, state machines are used in our research to formalize a UDS and the UDS superimposed by $CLDSA$. To faithfully express the informal description of the DSR property, we first formalize a UDS and the UDS superimposed by $CLDSA$ as state machines M_{UDS} and M_{CLDSA} , respectively. The DSR property is formalized based on them. Maude notation is used to describe state machines. Then we prove Theorem 1 saying that our new definition is equivalent to the definition of the DSR property in the existing study. To prove the Theorem 1, we prove Proposition 1 and Lemma 1. Lemma 1 asserts that reachability is preserved between M_{UDS} and M_{CLDSA} . Proposition 1 says that whenever $CLDSA$ terminates in state s , there is no marker in the start state, the snapshot and the finish state. We prove as Lemma 2 and Lemma 3 that one-step reachability is preserved between M_{UDS} and M_{CLDSA} to prove Lemma 1. The proof of Theorem 1 follows from Proposition 1 and Lemma 1. Furthermore, a binary simulation between M_{UDS} and M_{CLDSA} is used to prove that $CLDSA$ does not alter the behaviors of a UDS . Since $CLDSA$ works by using a special message called marker, $CLDSA$ does not alter the behaviors of a UDS means that excepting for putting markers in a UDS , the algorithm does not change the state of all processes and channels. We propose the binary relation \mathbf{r} between M_{UDS} and M_{CLDSA} saying that for each $s1 \in S_{UDS}$ and each $s2 \in S_{CLDSA}$, $\mathbf{r}(s1, s2)$ if and only if $s1$ is the same as the state obtained by deleting all markers from $s2$. To guarantee that $CLDSA$ does not alter the behaviors of a UDS , we prove Theorem 2 saying that \mathbf{r} is a bi-simulation relation between M_{UDS} and M_{CLDSA} .

Summarizing the research results, we have given a more faithful formal definition of the DSR property. Moreover, we have proved Theorem 1 saying that our formalization of the DSR property is equivalent to the existing one for each M_{UDS} . Theorem 1 guarantees 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 also proved Theorem 2 asserting that M_{CLDSA} simulates M_{UDS} and vice versa to guarantee that $CLDSA$ does not alter the behaviors of the UDS . The research results are depicted in Fig. 1.1.

1.3 Related Work

$CLDSA$ has been originally proposed in [1] as the first, in which the DSR property is described in the informal way. The DSR property is formalized as an invariant in [7], but their definition does not explicitly involve two state machines. $CLDSA$ is modeled in Promela and model checked with Spin [10]. In which, the consistent snapshot is defined, but the DSR property is not mentioned directly.

Andriamiarina, Méry and Singh [9] investigate the correct-by-construction process through which some distributed snapshot algorithms including $CLDSA$ are derived with Event-B and Rodin. Starting with an abstract model in which a consistent cut is instantaneously recorded, several refinement steps are repeated to construct some distributed snapshot algorithms that describe how to obtain consistent cuts by multiple transition steps. A consistent cut is a set of events of a distributed snapshot algorithm that satisfy some con-

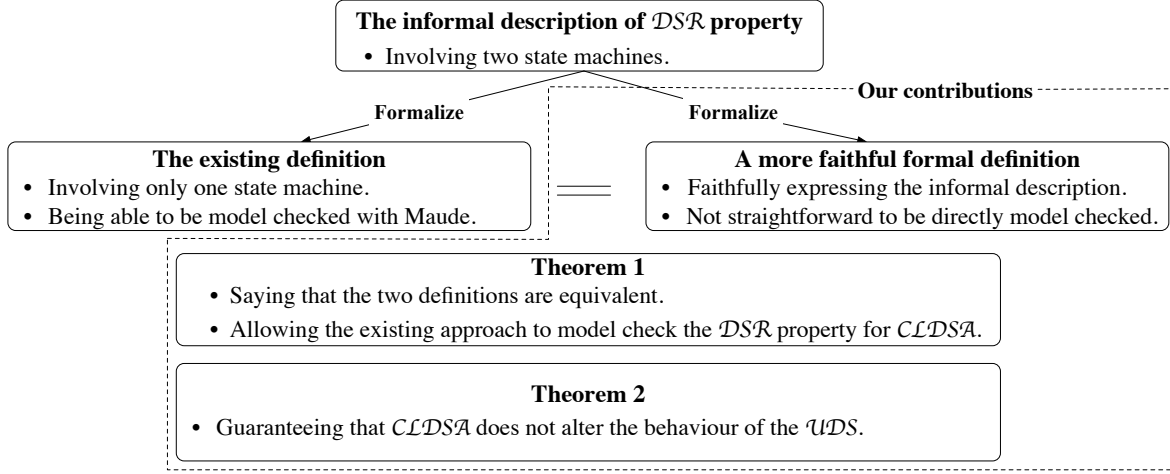


Figure 1.1: The contributions of our research

ditions, and can induce a snapshot. So, recording a consistent cut is regarded as taking a snapshot. Although they use multiple state machines described in Event-B, the relation between their state machines and ours ($M_{\mathcal{UDS}}$ and $CL(M_{\mathcal{UDS}})$) is not clear and must be worth investigating.

1.4 Organization of the Thesis

The thesis is organized as follows:

- Chapter 2 introduces some preliminaries, such as a \mathcal{UDS} , state machines, model checking and \mathcal{CLDSA} , and gives an informal description of the \mathcal{DSR} property.
- Chapter 3 presents the specification and model checking of \mathcal{CLDSA} in Maude.
- Chapter 4 describes how to formalize a \mathcal{UDS} and the \mathcal{UDS} superimposed by \mathcal{CLDSA} as state machines $M_{\mathcal{UDS}}$ and $CL(M_{\mathcal{UDS}})$, where CL is a function that takes $M_{\mathcal{UDS}}$ and returns the state machine of the \mathcal{UDS} superimposed by \mathcal{CLDSA} .
- Chapter 5 gives the more faithful formal definition of the \mathcal{DSR} property.
- Chapter 6 proves the theorem that guarantees the validity of the model checking approach used in the existing study and we can use the model checking approach to model check the new definition of the \mathcal{DSR} property.
- Chapter 7 proves another theorem asserting that $M_{\mathcal{CLDSA}}$ simulates $M_{\mathcal{UDS}}$ and vice versa to guarantee that \mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS} .
- Chapter 8 discusses some future work.
- Chapter 9 concludes the thesis.

Chapter 2

Preliminaries

2.1 Underlying Distributed Systems

As described in [1], a *UDS* consists of a finite set of processes and channels, which can be described as a labeled, directed graph in which the vertices represent the processes and the directed edges represent the channels. An example is shown in Fig. 2.1. The *UDS* consists of three processes p , q , r and four channels $c1$, $c2$, $c3$, $c4$. The channel $c4$ is used to directly send messages to the process p from the process r , but not vice versa. There may be more than one channel from one process to another. There are the two channels $c1$, $c2$ from the process p and the process q in the *UDS*. In addition, channels are assumed to have infinite buffers, to be error-free and to deliver messages in the order sent. The delay experienced by a message is arbitrary but finite. The processes communicate by sending and receiving messages through channels.

The *global state* of a distributed system consists of the states of all processes and all channels in the system, where the state of a channel is characterized by the sequence of messages sent along the channel, excluding the messages received along the channel. Since many problems on distributed systems such as recovering from faulty states and detecting stable properties can be cast to taking global snapshots of distributed systems. Taking a global state of a distributed system is very important. A global state is a consistent global state iff it satisfies that: if in the local state of a sender process p , a message m is recorded as sent, it must be captured in the state of the channel c connected the sender and the receiver or in the local state of the receiver process q and in the collected global state, for every effect, its cause must be present, and if a message m is not recorded as sent in the local state of process p , then it must neither be presented in the state of the channel c nor in the local state of the receiver process q [8]. To recording the meaning a global state of a distributed system, the states of all processes and all channels should be record at exactly the same instant. Due to the asynchrony of distributed systems, the lack of globally shared memory, global clock and unpredictable message delays, however, all processes cannot record their local states at exactly the same time. Then the global states we obtained may be inconsistent. This makes recording consistent global states non-trivial.

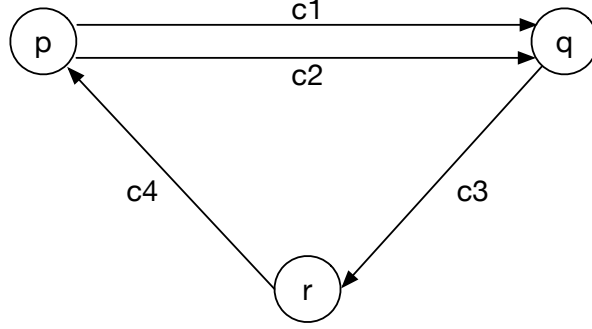


Figure 2.1: A distributed system with processes p, q, r and channels c1, c2, c3, c4.

2.2 \mathcal{CLDSA} and the \mathcal{DSR} Property

Chandy and Lamport have proposed a distributed snapshot algorithm [1] by which processes can record their own states and the states of incoming channels such that the combination of all process states and all channel states forms a consistent global state. \mathcal{CLDSA} guides each process when it should record its own state and the state of each incoming channel by using a special message called marker. \mathcal{CLDSA} runs concurrently with the computation but does not alter the underlying computation. Each process can record its state at any time when it has not yet received any markers from other processes. The following rules must be followed:

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 its incoming channel 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 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 .

\mathcal{CLDSA} can be initiated by one or more processes in a distributed system by executing the “*Marker Sending Rule*” by which each of them records its local state spontaneously without receiving markers from other processes and sends one marker along each of its outgoing channels. A process executes the “*Marker Receiving Rule*” on receiving a marker. If the process has not yet recorded its local state, then it records its local state, records the state of the channel on which the marker is received as the empty sequence, and sends one marker along each of its outgoing channels. Otherwise, it records the state of the channel

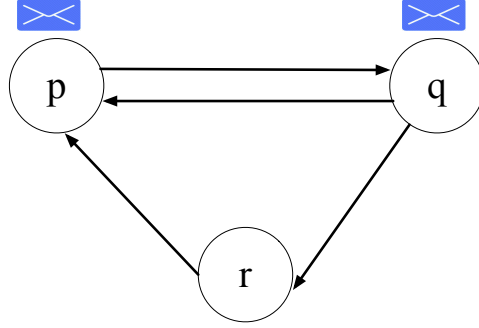


Figure 2.2: The \mathcal{UDS} with processes p, q, r and 4 channels

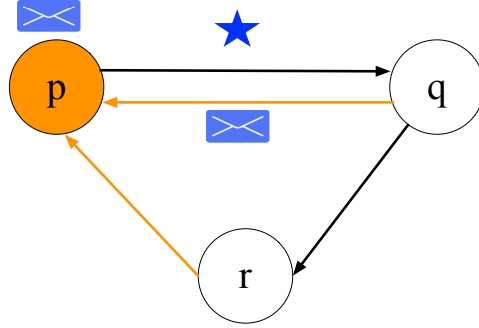


Figure 2.3: The process p initially start the algorithm.

as the sequence of messages received along the channel after its state was recorded and before it received the marker along the channel. When the algorithm terminates, each process has already recorded its state and the states of all its incoming channels. The global snapshot is the combination of those records.

One Scenario of \mathcal{CLDSA} .

Let us consider the \mathcal{UDS} that consists of three processes p, q, r and four channels as shown in Fig .2.2. We assume that there are two tokens in the system and the state of each process depends on the number of tokens it has. In the state shown in Fig .2.2, there are one token in p and one token in q .

Now we assume that p initially starts the algorithm. The start state is same the state of the system as shown in Fig .2.2. There are one token in p and another in q . Since p initially starts the algorithm, it records its state as the state in which there is one token and the state of two its incoming channels as empty sequence. Then, it sends one marker along one its outgoing channel from p to q . Assume that in the same time q sends one token to p . Let see the illustration in Fig .2.3.

Then q receives the marker. Because this is the first time q receives a marker, it records its state as the state in which there is no token and the state of the channel is as empty

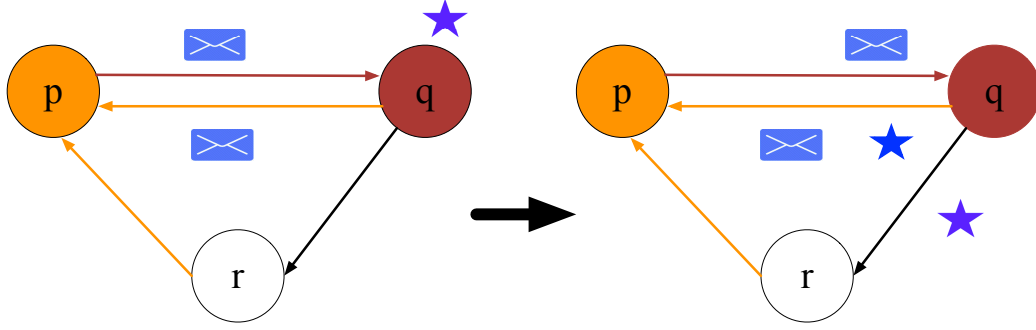


Figure 2.4: The process q receive the marker from p .

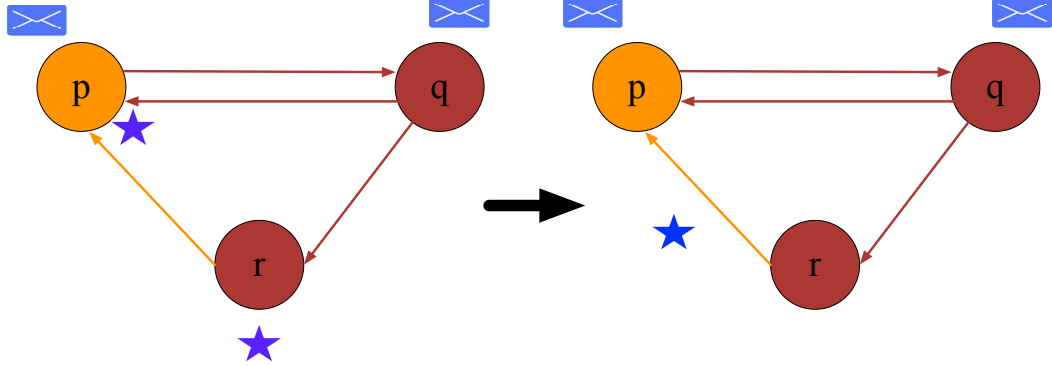


Figure 2.5: The process p and r receive the marker from p .

sequence. Then, it sends two markers along two its outgoing channels. Since q has only one incoming channel, it locally completes the algorithm. Assuming that in the same time, p sends one token to q . Let see the illustration in Fig .2.4.

Because of FIFO channel, p receives the token before it receives the marker from q . When p receives the marker from q , since it has already recored its state then it records the state of the channel from q to p as the sequence in which there is one token. Also, r receives a marker from q . In the same way, r records its state and the state of the channel and then it sends one marker to p . r locally completes the algorithm. Let see the illustration in Fig .2.5.

The algorithm is globally completed when p receives the marker from r . The finish state is shown in Fig .2.6. The snapshot is shown in Fig .2.7 in which there are one token in p and another in the channel from q to p .

The *DSR* Property.

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

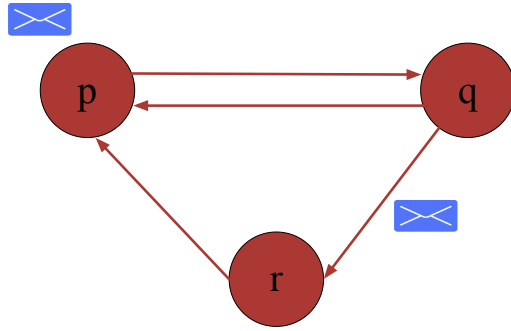


Figure 2.6: The finish state.

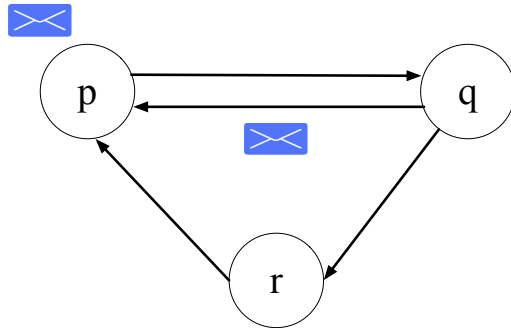


Figure 2.7: The snapshot taking by *CLDSA*.

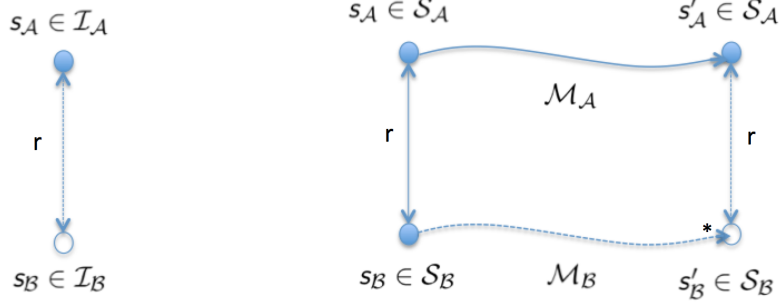


Figure 2.8: Simulation from \mathcal{M}_A to \mathcal{M}_B .

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* superimposed by *CLDSA*.

2.3 State Machine

State machines can be used to formalize distributed systems. It consists of a set of states including a set of initial states and a total 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 over states $T \subseteq S \times S$.

Each element $(s, s') \in T$ is called a (state) transition from s to s' (or just a transition). Since T is total, for each state $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in T$.

Infinite sequences of states called paths are generated from a state machine. Paths are defined as follows:

Definition 2 (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 i -th state (i.e., s_i) in π and Π denotes the set of all paths of M .

Simulation from one state machine to another state machine is defined as follows:

Definition 3 (Simulation from \mathcal{M}_A to \mathcal{M}_B) Given two state machines $\mathcal{M}_A \triangleq \langle \mathcal{S}_A, \mathcal{I}_A, \mathcal{T}_A \rangle$ and $\mathcal{M}_B \triangleq \langle \mathcal{S}_B, \mathcal{I}_B, \mathcal{T}_B \rangle$, $r : \mathcal{S}_A \times \mathcal{S}_B \rightarrow \text{Bool}$ is called a simulation from \mathcal{M}_A to \mathcal{M}_B if it satisfies the following conditions:

1. For each $s_A \in \mathcal{I}_A$, there exists $s_B \in \mathcal{I}_B$ such that $r(s_A, s_B)$.
2. For each $s_A, s'_A \in \mathcal{S}_A$ and $s_B \in \mathcal{S}_B$ such that $r(s_A, s_B)$ and $s_A \rightsquigarrow_{\mathcal{M}_A} s'_A$, there exists $s'_B \in \mathcal{S}_B$ such that $r(s'_A, s'_B)$ and $s_B \rightsquigarrow_{\mathcal{M}_B}^* s'_B$.

Note that $s_A \rightsquigarrow_{\mathcal{M}_A} s'_A$ means that s_A goes to s'_A by one state transition in \mathcal{M}_A and $s_B \rightsquigarrow_{\mathcal{M}_B}^* s'_B$ means that s_B goes to s'_B by zero or more state transitions in \mathcal{M}_B .

Fig. 2.8 shows the diagrams corresponding to the two conditions in the Definition 3.

2.4 Model Cheking

Model checking is an automated technique for verifying finite state systems, such as sequential circuit designs and communication protocols [11]. Given a finite-state model of a system and a formal property, that technique systematically checks whether this property holds for that model. The principle of model checking is: generate all possible states of the system and check that whether its desired property holds in each state. In case, there are errors, the property does not hold, model checking will produce a counterexample that can be used to find the source of the errors. Model-checking problem can be described as follows:

Given a transition system M and a formula f , the model-checking problem is to decide whether $M \models f$ holds or not. If not, the model checker should provide an explanation why, in the form of counterexample.

A model checker is the software tool that performs the model checking in which a model is described in a formal description language and a property is specified in formal way. A model checker generate all possible states of a system, then it automatically checks that whether the system satisfies its desired property. If the property is found to not hold, a counterexample is generated to help to find out why the property does not hold.

The process of model checking can be distinguished into the following different tasks:

- **Modeling:** is to model a system under consideration using the model description language. Models of systems describe the behavior of systems in an accurate and unambiguous way. They are often expressed by using finite-state automata, which consist of a finite set of states and a set of transitions. Transitions describe how a system moves from one state to another.
- **Specification:** is to state the property that a system must satisfy before model checking. The property is specified in a specification language, usually in a logic-based formalism. A model of the system and a formula of the property are considered as inputs to model checking. Completeness is one of the important issue in specification. Model checking provides means for checking that a model of the system satisfies a given specification. However, it is impossible to determine whether the given specification covers all the properties that the system should satisfy [11].

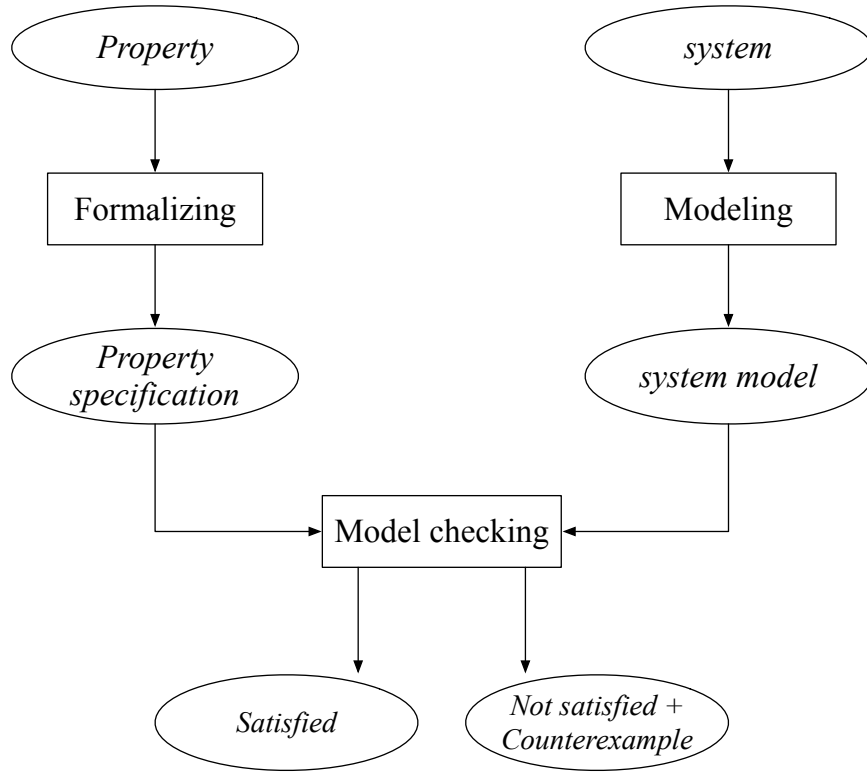


Figure 2.9: Schematic view of the model-checking approach.

- **Verification:** is to check the validity of the property. Running the model checker to automatic check the validity of the property in the system model. There are basically three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

The entire model-checking process should be well organized, well structured, and well planned. One schematic view of the model-checking approach is shown in Fig .2.9.

Model checking has a number of advantages over verification techniques based on automated theorem proving. The main benefit of model checking is highly automatic. However, the main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values.

2.5 Maude

Maude, a specification and programming language, is based on rewriting logic, which contains membership equational logic as its sub-logic. State machines (or transition systems) and their data are specified in rewriting logic and membership equational logic, respectively. States of state machines are expressed as data, such as tuples and associative

commutative collections, and state transition rules are described in rewriting rules. The specifications of state machines are called system specifications. Basic units of Maude specification and programming are modules. A module consists of syntax declaration, which declare for sorts, subsorts, kinds and operators. The syntax declaration part provides appropriate language to describe system and is called a signature. There are two kinds of modules in Core Maude: *functional modules* and *system modules*. Signatures are common for both of them, however,

- A *functional modules* is a equational theory in membership equational logic. *functional modules* admit equations and memberships. *functional modules* are declared with the keyword *fmod ... endfm*, where ‘...’ corresponding to all declaration of submodule importation, sorts, subsorts, operators, variables, equations, and so on.

- A *system modules* is a rewrite theory. *system modules* admit rules, which are used to describe transitions between states. *system modules* are declared with the keyword *mod ... endm*, where ‘...’ corresponding to all declaration of submodule importation, sorts, subsorts, operators, variables, equations, rules and so on.

Some built-in modules are provided in Maude such as BOOL and NAT for Boolean values and natural numbers. The boolean values are denoted as true and false, and natural numbers as 0, 1, 2, ... as usual. The corresponding sorts are Bool and Nat.

In addition, Maude is equipped with many useful functionalities. Among them are model checking facilities and meta-programming facilities. The model checking facilities mainly consists of an LTL model checker and a search command that can be used as a reachability checker. The search command is used in the existing study. the search command is used to model check invariant properties of concurrent systems specified as system modules in Maude.

Given a state s , a state pattern p and an optional condition c , the search command searches the reachable state space from s in a breadth-first manner for all states that match p such that c holds. The syntax of search command is as follows:

search [n , m] in M : $s \Rightarrow p$ such that c .

where M is a module in which the specification of the state machine concerned is described or available, s is the start term, p is the pattern that has to be reached, \Rightarrow is an arrow indicating the form of the rewriting proof from s until p . A rewrite expression $t \Rightarrow t'$ can be used in the optional condition c , that has to be satisfied by the reached state. This checks if t' is reachable from t by zero or more rewrite steps with rewrite rules. \Rightarrow can be:

- $\Rightarrow 1$: means a rewriting proof consisting of exactly one step,
- $\Rightarrow +$: means a rewriting proof consisting of exactly one or more steps.
- $\Rightarrow *$: means a rewriting proof consisting of exactly none, one or more steps, and
- $\Rightarrow !$: indicates that only canonical final states are allowed.

Although the reachable state space is bounded, the whole state space is unbounded. The search command can be given as options the maximum number of solutions and the maximum depth of search, namely n is an optional argument proving a bound on the

number of desired solutions and m is an optional argument for the maximum depth of search.

A metaprogram is a program that takes programs as inputs and performs some useful computation. It may transform one program into another or may analyze such programs. In Maude, metaprogram has a logical, reflective semantics. This reflects the fact that both membership equational logic and rewriting are reflective logics. The Maude meta-programming functionalities can treat Maude specifications as data. The META-LEVEL module, in which key functionality of the universal theory has been implemented, is built. META-LEVEL module can be imported to easily write meta-programming in Maude. It includes the module META-MODULE and META-TERM. In the META-TERM module, Maude terms are metarepresented as elements of a data type *Term* of terms. In the META-MODULE module, Maude modules are metarepresented as elements of a data type *Module* of modules. In the META-LEVEL, there are operations, such as *upModule*, *upTerm* and *downTerm* to allow moving between reflection levels. The process of reducing and rewriting a term using Maude's command is metarepresented by built-in functions *metaReduce* and *metaRewrite*, respectively. The process of applying a rule is metarepresented by built-in function *metaApply*. The process of matching two terms is metarepresented by built-in function *metaMatch* and the process of searching for a term satisfying some conditions is reified by built-in functions *metaSearch* and *metaSearchPath*. The functions *metaReduce*, *metaRewrite*, *metaApply*, *metaSearch* and *metaSearchPath* are called descent functions that allow us to descend levels in the reflective tower.

2.6 Chapter Summary

This chapter introduced some preliminaries that are very necessary. It first described a *UDS* and introduced the concept of the global state of a distributed system. Also, the problems of taking global snapshots of distributed systems were mentioned.

It then introduced *CLDSA* and the *DSR* property. One scenario of *CLDSA* was given to easily imagine how the algorithm works. The informal description of the *DSR* property was given based on the original paper [1] in which the *DSR* property is described as the first.

The chapter then gave the definitions of state machine. Then some other definitions on state machines, such as path and simulation from one state machine to another state machine were given.

The Model checking section presented about model checking in which the principle of model checking was presented and the model-checking problem was described. Also, this section described the process of model checking and the schematic view of the model-checking approach.

The last section in this chapter introduced Maude, in which Maude search command and metaprogram in Maude were introduced in detail.

Chapter 3

Specification and Model Checking of \mathcal{CLDSA} in Maude

\mathcal{CLDSA} has been formally analyzed in the existing study [2] in which Maude has been used to model check that \mathcal{CLDSA} enjoys the \mathcal{DSR} property. The authors describe how to specify \mathcal{CLDSA} in Maude and how to model check that \mathcal{CLDSA} enjoys the \mathcal{DSR} properties with the Maude search command.

3.1 Specification of \mathcal{CLDSA}

\mathcal{CLDSA} has been specified in Maude. What is modeled is actually the \mathcal{UDS} superimposed by \mathcal{CLDSA} . A \mathcal{UDS} consists one or more processes that are connected with directed channels that are unbounded queues. A system is assumed that it may be has only one process, and some processes have no outgoing channels, no incoming channels, or neither of them. There are no self-channel, which from one process to its self. Messages in the system are separated into token for non-marker message and marker for \mathcal{CLDSA} . Tokens can be consumed by processes.

To specify the algorithm, basic data used for identifying processes, tokens and makers are sorts Pid , Tokens and marker , respectively. Sorts EmpChan , NeChan and Chan are used for empty channels, non-empty channels and channels. The state of each process and the state of each channel are considered as observable components, in which the state of a process and the state of a channel are denoted as p-state and c-state, respectively. The state of a process depends on the number of token it has and the state of a channel is as the sequence of messages containing tokens and markers on the channel. Other observable components are for control information, which are used to control behaviour of the algorithm. The state of a \mathcal{UDS} , the start state, the finish state, and the snapshot are expressed as $\text{base-state}(\dots)$, $\text{start-state}(\dots)$, $\text{finish-state}(\dots)$ and $\text{snapshot}(\dots)$, respectively, where ‘ \dots ’ is a soup of p-state and c-state observable components. Those are called meta configuration components and the sort is MCComp . Moreover, the control information is expressed as $\text{control}(\dots)$ that is also meta configuration components, where ‘ \dots ’ is also a soup of observable components of control information. A state of a \mathcal{UDS} superimposed

by \mathcal{CLDSA} is expressed as a soup of meta configuration components, which is called a meta configuration. A meta configuration is a global view of the system (the UDS superimposed by \mathcal{CLDSA}). Although the system specification does not depend on the number of processes or the number of channels, they need to fix those number to model check the DSR property. Fixing those number is described in initial meta configurations.

Each process in the a UDS superimposed by \mathcal{CLDSA} can do the following actions:

1. The process may consume a token owned by it and changes its state accordingly.
2. The process may put a token into one of its outgoing channels if it has some outgoing channels and changes its state accordingly.
3. The process may get a token from one of its nonempty incoming channels if it has some nonempty incoming channels and changes its state accordingly.
4. The process may start the \mathcal{CLDSA} when it has not yet received any markers. It records its state, initializes the states of its incoming channels as empty if any, and puts one marker into each of its outgoing channels if any.
5. The process may get a marker from one of its incoming channels if it has some incoming channel. If it has already started the \mathcal{CLDSA} , it has completed the record of the incoming channel. Moreover, if it has received markers from all the incoming channels, it has locally completed the \mathcal{CLDSA} . If it has not yet started, it records its state and the state of the incoming channel as empty, and initializes the states of the other incoming channels as empty if any. Then, it puts one marker into each of its outgoing channels if any. If it has only one incoming channel, it has locally completed the \mathcal{CLDSA} . Note that the first three describe the UDS part.

Actions of processes in the UDS superimposed by \mathcal{CLDSA} are described by rewriting rules in Maude. Here is an example, in which the consumption of tokens is described by the following rewriting rule:

$$\begin{array}{l}
\text{rl [chgStt]} : \\
\text{base-state}((\text{p-state}[P] : (T \ PS)) \ BC) \ \text{finish-state}(\text{empConfig}) \ \text{control}((\text{consume} : \text{true}) \ CC) \\
\Rightarrow \\
\text{base-state}((\text{p-state}[P] : PS) \ BC) \ \text{finish-state}(\text{empConfig}) \ \text{control}((\text{consume} : \text{true}) \ CC)
\end{array}$$

where BC and CC are variable of sort for a soup of observable components, T is variable of sort for Token, PS is variable of sort for state of processes (as the sequence of Tokens).

For each of basic actions of processes, there are multiple rewriting rules that have been obtained by case analyzes based on predicates that are not locally observable by any process. Case analyzes are to cover all possible situations.

3.2 Model Checking of \mathcal{CLDSA}

The Maude system is equipped with model checking facilities: the search command and the LTL model checker. In the existing study, the model checking invariants through search is used. They model check that \mathcal{CLDSA} satisfies the \mathcal{DSR} property by the Maude search command. The \mathcal{DSR} property can be divided into $\mathcal{RP1}$ and $\mathcal{RP2}$ as follows:

1. s_* is always reachable from s_1 ($\mathcal{RP1}$), and
2. s_2 is always reachable from s_* ($\mathcal{RP2}$).

which can be checked with the Maude search command. The Maude search command is used to search states that satisfies some conditions. To model check \mathcal{DSR} property, they do the following Maude search commands. The following part, SC , FC and SSC are variable of sort for a soup of observable components, MC is variable of sort for meta configuration, b is either true or false, and n is the number of processes in the system.

- To find all states in which a snapshot has been taken by following Maude search command:

```
search in EXPERIMENT : imc =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC /= empConfig .
```

- To find all states in which a snapshot has been taken such that the snapshot SSC is reachable from the start state SC by following Maude search command:

```
search in EXPERIMENT :
imc =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC /= empConfig
^ base-state(SC) control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b))
=>
base-state(SSC) control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b)) .
```

- To find all states in which a snapshot has been taken such that the snapshot FC is reachable from the snapshot SSC by following Maude search command:

```
search in EXPERIMENT :
imc01 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC /= empConfig
^ base-state(SSC) control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b))
=>
base-state(FC) control((prog[p(0)]: notYet) ... (prog[p(1)]: notYet) (consume : b)) .
```

Let m_0 , m_1 and m_2 be the number of the solutions to the search for snapshots, the search for $\mathcal{RP1}$ and the search for $\mathcal{RP2}$. The search for snapshots finds all states in

which a snapshot has been taken. The search for $\mathcal{RP1}$ finds all states in which a snapshot has been taken such that the snapshot is reachable from the start state under the \mathcal{UDS} , if m_1 equals m_0 , \mathcal{CLDSA} enjoys $\mathcal{RP1}$. $\mathcal{RP2}$ can be checked likewise. The search for $\mathcal{RP2}$ finds all states in which a snapshot has been taken such that the finish state is reachable from the snapshot under the \mathcal{UDS} , if m_2 equals m_0 , \mathcal{CLDSA} enjoys $\mathcal{RP2}$. The \mathcal{DSR} property ($\mathcal{RP1}$ and $\mathcal{RP2}$) holds if and only if $((m_1 == m_0) \wedge (m_2 == m_0))$. And this is the key point of model checking the \mathcal{DSR} property.

3.3 Chapter Summary

This chapter described how to specify \mathcal{CLDSA} and how to model check that the algorithm enjoy the \mathcal{DSR} property in the existing study. In detail, it described what the system is actually specified and presented how the states of the system is expressed. Then it described how the actions of processes in the \mathcal{UDS} superimposed by \mathcal{CLDSA} are described by rewriting rules in Maude. The way to specify the algorithm in Maude is presented. However, please see fully the specification of \mathcal{CLDSA} in Appendix A.

The chapter then showed how to model check that \mathcal{CLDSA} enjoy the \mathcal{DSR} property by Maude search command in the existing study. It also explained how the \mathcal{DSR} property is encoded in Maude search command. The method to model check that \mathcal{CLDSA} enjoy the \mathcal{DSR} property was presented. However, more experiments conducted are shown in Appendix B.

Chapter 4

Formalizing a \mathcal{UDS} and a \mathcal{UDS} Superimposed by \mathcal{CLDSA} as State Machines

4.1 Modeling a \mathcal{UDS} as a State Machine

To model a \mathcal{UDS} as a state machine $M_{\mathcal{UDS}}$, we consider how to express states and transitions of $M_{\mathcal{UDS}}$.

4.1.1 State Expression for a \mathcal{UDS}

Each process in a \mathcal{UDS} has its own local state, and so does each channel. Each state of $M_{\mathcal{UDS}}$ should consist of the state of each process and the state of each channel in the \mathcal{UDS} . We use *name-value* pairs (called observable components) to express the states of processes and channels, where name may have parameters. For observable components for process states and channel states, the following operators that are constructors as specified with ctor are prepared¹:

```
sort MsgQueue .
op empChan :  $\rightarrow$  MsgQueue [ctor] .
op  $\_[-]$  : Msg MsgSeq  $\rightarrow$  MsgQueue [ctor] .
sort OCom .
op p-state $[_]_-$  : Pid PState  $\rightarrow$  OCom [ctor] .
op c-state $[_]$ ,  $\_[-]$ ,  $\_[-]$  : Pid Pid Nat MsgQueue  $\rightarrow$  OCom [ctor] .
```

where Pid is the sort for process identifiers, PState is the sort for process states, Nat is the sort for natural numbers, MsgQueue is the sort for queues of messages for which the sort Msg is used, and OCom is the sort for observable components. A sort is also used as the set of all ground constructor terms (or values) of the sort. Let $p, q \in \text{Pid}$, $ps \in \text{PState}$,

¹Maude notation is used to describe state machines.

$n \in \text{Nat}$ and $ms \in \text{MsgQueue}$. $(\text{p-state}[p]: ps)$ is an observable component whose name is $\text{p-state}[p]$ 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 . $(\text{c-state}[p, q, n]: ms)$ is an observable component whose name is $\text{c-state}[p, q, n]$ 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 a process p to a process q is ms . Since there may be more than one channel from p to q , a natural number n is used in $(\text{c-state}[p, q, n]: ms)$ to identify the channel.

We use a soup² of process and channel states to represent each state of $M_{\mathcal{UDS}}$. For this end, the following sorts and operators are prepared:

```

sort Config.
subsort OCom < Config .
op empConfig : → Config [ctor] .
op _ _ : Config Config → Config [ctor assoc comm id:empConfig] .

```

Config is the sort for soups of observable components and a super-sort of OCom , which means that each term of OCom is treated as the singleton soup only consisting of the term. empConfig denotes the empty soup of observable components, and the juxtaposition operator $_ _$ is used to construct soups of observable components. For $c_1, c_2 \in \text{Config}$, $c_1 \ c_2 \in \text{Config}$. 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 .

We give the following functions on the states of a \mathcal{UDS} . In the following part, $P, Q \in \text{Pid}$, $PS \in \text{PState}$, $N \in \text{Nat}$, $CS, MS \in \text{MsgQueue}$, $CF, CF1 \in \text{Config}$ and $M \in \text{Msg}$ are variables of those sorts.

- Function chans : to get the set of all channels of a \mathcal{UDS} ;

```

op chans : Config → Config .
eq chans(empConfig) = empConfig .
eq chans((p-state[P]: PS) CF) = chans(CF) .
eq chans((c-state[P, Q, N]: CS) CF) = (c-state[P, Q, N] : CS) chans(CF) .

```

- Function msg : to get the state of a channel;

```

op msg : Ocom → MsgSeq .
eq msg(c-state[P, Q, N] : CS) = CS .
eq msg(p-state[P]: PS) = empChan .

```

- Function enq : to put a message at the end of a sequence of messages;

```

op enq : MsgSeq Msg → MsgSeq .

```

²Associative-commutative collections may be called multisets or bags, but we use soups to refer to such collections according to the nomenclature of the Maude community.

$\text{eq } \text{enq}(\text{empChan}, M2) = M2 \mid \text{empChan} .$
 $\text{eq } \text{enq}(M1 \mid MS, M2) = M1 \mid \text{enq}(MS, M2) .$

Function $\#$: to count the number of occurrences of a process or channel in a state of a \mathcal{UDS} ;

$\text{op } \# : \text{Config Pid} \rightarrow \text{Nat} .$
 $\text{eq } \#(\text{empConfig}, P) = 0 .$
 $\text{eq } \#((\text{p-state}[P1] : PS) CF, P) = \text{if } P = P1 \text{ then } 1 + \#(CF, P) \text{ else } \#(CF, P) \text{ fi} .$
 $\text{eq } \#((\text{c-state}[P1, Q1, N1] : CS) CF, P) = \#(CF, P) .$

$\text{op } \# : \text{Config Pid Pid Nat} \rightarrow \text{Nat} .$
 $\text{eq } \#(\text{empConfig}, P, Q, N) = 0 .$
 $\text{eq } \#((\text{p-state}[P1] : PS) CF, P, Q, N) = \#(CF, P, Q, N) .$
 $\text{eq } \#((\text{c-state}[P1, Q1, N1] : CS) CF, P, Q, N) =$
 $\text{if } (P = P1) \text{ and } (Q = Q1) \text{ and } (N = N1)$
 $\text{then } 1 + \#(CF, P, Q, N) \text{ else } \#(CF, P, Q, N) \text{ fi} .$

Function $=$: to check out the equivalence between the two states of a \mathcal{UDS} ;

$\text{op } = : \text{Config Config} \rightarrow \text{Bool} .$
 $\text{eq } (\text{empConfig} = \text{empConfig}) = \text{true} .$
 $\text{eq } (((\text{p-state}[P] : PS) CF) = ((\text{p-state}[P] : PS) CF1)) = (CF = CF1) .$
 $\text{eq } (((\text{c-state}[P, Q, N] : CS) CF) = ((\text{c-state}[P, Q, N] : CS) CF1)) = (CF = CF1) .$
 $\text{eq } ((OC \ CF) = (OC1 \ CF1)) = \text{false} \text{ [otherwise]} .$

4.1.2 State Transitions for $M_{\mathcal{UDS}}$

Each process in a \mathcal{UDS} may do three kinds of actions:

- i it may change its state without putting any message into any of its outgoing channels nor getting any message from any of its incoming channels,
- ii it may put a message into one of its outgoing channels and may change its state (may not change its state), and
- iii it may get the top message from one of its incoming channels if the channel is not empty and may change its state (may not change its state).

The three kinds of actions i, ii and iii are called Change of Process State, Sending of Message and Receipt of Message, respectively. The actions are described as transition rules. A transition rule is described in form of rewrite rule. In the following part, $P, Q \in \text{Pid}$, $PS1, PS2 \in \text{PState}$, $N \in \text{Nat}$, $CS \in \text{MsgQueue}$ and $M \in \text{Msg}$ are variables of those

sorts.

- Change of Process State is described as the following transition rule³:

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

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

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

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:

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

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

There are only three transition rules. Due to the number of states for each process, the number of channels, the number of states for each channel, etc., however, there may be more than three ground instances of the transition rules. 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 4 ($TR_{\mathcal{UDS}}$) *Let $TR_{\mathcal{UDS}}$ be the set of all ground instances of the three transition rules.*

³The rewrite rule is not executable because the variable $PS2$ does not appear in the left-hand side. 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 \mathcal{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.

4.1.3 State Machine M_{UDS}

A UDS is formalized as $M_{UDS} \triangleq \langle S_{UDS}, I_{UDS}, T_{UDS} \rangle$. S_{UDS} is the set of all ground constructor terms whose sorts are $Config$. 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 5 (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 $(\forall s \in I_{UDS})(\forall c \in chans(s))(msg(c) = empChan)$,
 $(\forall s \in I_{UDS})(\forall p \in Pid)(\#(s, p) \leq 1)$, $(\forall s \in I_{UDS})(\forall p, q \in Pid)(\forall n \in Nat)$
 $(\#(s, p, q, n) \leq 1)$ and $(\forall s \in I_{UDS})(\forall (c\text{-state}[p, q, n] : cs) \in s)((\#(s, p) = 1)$
 $\wedge (\#(s, q) = 1))$;
3. T_{UDS} is the binary relation over S_{UDS} defined as follows:
 $\{(L \text{ CF}, R \text{ CF}) \mid L \Rightarrow R \in TR_{UDS}, CF \in Config\}$.

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 .

4.2 Modeling a UDS Superimposed by \mathcal{CLDSA} as a State Machine

This part considers how to model a UDS superimposed by \mathcal{CLDSA} as a state machine $CL(M_{UDS})$. Let us consider the state expression and the state transition for $CL(M_{UDS})$.

4.2.1 State Expression for $CL(M_{UDS})$

Taking into account a UDS superimposed by \mathcal{CLDSA} , each state of $CL(M_{UDS})$ consists of the local states of all processes and channels, the state (called the start state) when \mathcal{CLDSA} initiates, the snapshot, the state (called the finish state) when \mathcal{CLDSA} terminates and the information to control behaviour of \mathcal{CLDSA} . The local states of all processes and channels, the start state, the snapshot, the finish state and the control information are expressed as $base\text{-}state(bc)$, $start\text{-}state(sc)$, $snapshot(ssc)$, $finish\text{-}state(fc)$ and $control(ctl)$ meta configuration components, respectively, where:

- bc is a soup of process and channel states whose sorts are $BOCom$ and the corresponding sort is $BConfig$. Note that there may be makers in channels of bc . $MMsg$ is the sort for ordinary messages and a marker. $MMsgQueue$ is the sort for queues of $MMsg$. We

prepare the following sorts and operators for them:

```

sort MMsg .
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] .

```

- *sc*, *ssc*, *fc* are ground constructor terms of BConfig .
- *ctl* is a soup of cnt, prog, #ms, and done control observable components that will be described. CtlIOCom is the sort for those control observable components, and CtlConfig is the sort for soups of CtlIOCom. We prepare the following sorts and operators for them:

```

sort Prog .
ops notYet, started, completed : → Prog [ctor] .
op cnt: _ : Nat → CtlIOCom [ctor] .
op prog[_]:_ : Pid Prog → CtlIOCom [ctor] .
op #ms[_]:_ : Pid Nat → CtlIOCom [ctor] .
op done[_ , _ , _]:_ : Pid Pid Nat Bool → CtlIOCom [ctor] .
subsort CtlIOCom < CtlConfig .
op empCtlConfig : → CtlConfig [ctor] .
op _ _ : CtlConfig CtlConfig → CtlConfig [ctor assoc comm id: empCtlConfig] .

```

We need the following control observable components to specify the behaviour of \mathcal{CLDSA} , where $p, q \in \text{Pid}$, $n \in \text{Nat}$, $pg \in \text{Prog}$, $b \in \text{Bool}$:

- ($\text{cnt} : n$): n is the number of processes that have not yet completed \mathcal{CLDSA} .
- ($\text{prog}[p] : pg$): pg is the progress of a process p , indicating that the process has not yet started, has started, or completed \mathcal{CLDSA} .
- ($\text{\#ms}[p] : n$): n is the number of incoming channels to a process p from which markers have not yet been received.
- ($\text{done}[p, q, n] : b$): b is either *true* or *false*. If b is *true*, q has received a marker from the incoming channel identified by n from q . Otherwise, q has not.

Each state of $CL(M_{\mathcal{UDS}})$ is expressed as the soup of the meta configuration components whose sorts are MBCom, which is typically 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 MBConfig. We prepare the following operators for expressing a state of $CL(M_{\mathcal{UDS}})$:

```

sort MBCom .
op base-state: BConfig → MBCom [ctor] .
ops start-state, snapshot, finish-state : BConfig → MBCom [ctor].
op control : CtlConfig → MBCom [ctor] .
sort MBConfig .
subsort MBCom < MBConfig [ctor] .
op _ : MBConfig MBConfig → MBConfig [ctor assoc comm].

```

Initially, *bc* is an initial state of $M_{\mathcal{UDS}}$ and all of *sc*, *ssc* and *fc* are empBConfig. The number of processes that have not yet completed \mathcal{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.

We give the following functions on the state of a $M_{\mathcal{CLDSA}}$. In the following part, $P, Q \in \text{Pid}$, $PS \in \text{PState}$, $N \in \text{Nat}$, $CS, MS \in \text{MMsgQueue}$, $CF, CF1 \in \text{Config}$, $BCF \in \text{BConfig}$ and $M \in \text{MMsg}$ are variables of those sorts.

- Function cntN: to get the number of processes in a \mathcal{UDS} ;

```

op cntN : BConfig → Nat .
eq cntN(empConfig) = 0 .
eq cntN((p-state[P]: PS) CF) = 1 + cntN(CF) .
eq cntN((c-state[P, Q, N]: CS) CF) = cntN(CF) .

```

- Function msN: to get the number of incoming channel of a process;

```

op msN : Pid BConfig → Nat .
eq msN(P, empConfig) = 0 .
eq msN(P, (p-state[P1]: PS) CF) = msN(P, CF) .
eq msN(P, (c-state[P1, Q1, N]: CS) CF) =
  if (P = Q1) then 1 + msN(P, CF) else msN(P, CF) fi.

```

- Function mkCnt: the number of processes that have not yet completed \mathcal{CLDSA} is equal to the number of processes in the system;

```

op mkCnt : BConfig → CtlConfig .
eq mkCnt(CF) = (cnt: cntN(CF)) .

```


- Function mkProg : the progress of all processes in a system are notYet;

```

op mkProg : BConfig → CtlConfig .
eq mkProg(empConfig) = empCtlConfig .
eq mkProg((p-state[P]: PS) CF) = (prog[P]: notYet) mkProg(CF) .
eq mkProg((c-state[P, Q, N]: MS) CF) = mkProg(CF) .

```

- Function mkDone : all processes have not yet received any markers from any their incoming channels;

```

op mkDone : BConfig → CtlConfig .
eq mkDone(empConfig) = empCtlConfig .
eq mkDone((p-state[P]: PS) CF) = mkDone(CF) .
eq mkDone((c-state[P, Q, N]: CS) CF) = (done[P, Q, N]: false) mkDone(CF) .

```

- Function mkMs : the number of channels of a process that have not yet received a marker is equal to the number of channels of the process;

```

op mkMs : BConfig BConfig → CtlConfig .
eq mkMs(empConfig, CF1) = empCtlConfig .
eq mkMs((p-state[P]: PS) CF, CF1) = (#ms[P]: msN(P, CF1)) mkMs(CF, CF1) .
eq mkMs((c-state[P, Q, N]: CS) CF, CF1) = mkMs(CF, CF1) .

```

- Function InitCtlConfig;

```

op InitCtlConfig : BConfig → CtlConfig .
eq InitCtlConfig(BC) = mkCnt(BC) mkMs(BC, BC)
mkDone(BC) mkDone(BC) mkProg(BC) .

```

Functions InitCtlConfig, which call sub-functions mkCnt, mkProg, mkDone and mkMs, is to initial control meta configuration component of an initial state of M_{UDS} such that the number of processes that have not yet completed \mathcal{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.

- Function delMchan: to delete all markers in the sequence of message. Note that there is almost one marker in a channel in a state of a UDS superimposed by \mathcal{CLDSA} , but this function is defined for general case, means that there may have more than one marker in a channel.

```

op delMchan : MMsgSeq → MsgSeq .
eq delMchan(empChan) = empChan .
eq delMchan(M | MMS) = if M = marker then delMchan(MMS)
else M | delMchan(MMS) fi.

```

- Function delM : to delete all markers in all channels in a soup of process and channel states in a \mathcal{UDS} superimposed by \mathcal{CLDSA} ;

$\text{op delM: BConfig} \rightarrow \text{Config} .$
 $\text{eq delM}(\text{empBConfig}) = \text{empConfig} .$
 $\text{eq delM}((\text{p-state}[P]: PS) BCF) = (\text{p-state}[P]: PS) \text{delM}(BCF) .$
 $\text{eq delM}((\text{c-state}[P, Q, N]: MMS) BCF) = (\text{c-state}[P, Q, N]: \text{delMchan}(MMS)) \text{delM}(BCF) .$

- Function bcast : to puts markers in all the outgoing channels from a process P ;

$\text{op bcast : BConfig Pid MMsg} \rightarrow \text{BConfig} .$
 $\text{eq bcast}(\text{empBConfig}, P, \text{marker}) = \text{empBConfig} .$
 $\text{eq bcast}((\text{c-state}[P, Q, N]: MMS) BCF, P, \text{marker}) =$
 $(\text{c-state}[P, Q, N]: \text{put}(MMS, \text{marker})) \text{bcast}(BCF, P, \text{marker}) .$
 $\text{eq bcast}(OCBCF, P, M1) = OC \text{bcast}(BCF, P, \text{marker}) [\text{owise}] .$

- Function inchans : to initialize the states of a process's all incoming channels as empty channel;

$\text{op inchans : Config Pid} \rightarrow \text{Config} .$
 $\text{eq inchans}(\text{empConfig}, P) = \text{empConfig} .$
 $\text{eq inchans}((\text{c-state}[P, Q, N]: CS) CF, P) = (\text{c-state}[P, Q, N]: \text{empChan}) \text{inchans}(CF, P) .$
 $\text{eq inchans}(OCCF, P) = \text{inchans}(CF, P) [\text{owise}] .$

4.2.2 State Transitions for $CL(M_{UDS})$

Each process in the system may do five kinds of actions. The three kinds of actions i, ii and iii are almost the same as those for a \mathcal{UDS} and the two more kinds of actions iv and v are as follows:

- iv it may record its state and put markers into its outgoing channels, and
- v it may get a marker from one of its incoming channels.

The two kinds of actions iv and v are called Record of Process State and Receipt of Marker, respectively. In the following part, $P, Q \in \text{Pid}$, $PS, PS1, PS2 \in \text{PState}$, $BC, SC, FC, SSC \in \text{BConfig}$, $MMS, MMS' \in \text{MMsgQueue}$, $CC \in \text{CtlConfig}$, $N, NzN, NzN' \in \text{NzNat}$ and $M \in \text{Msg}$ 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

base-state((p-state[P] : $PS2$) BC)

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

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

- Receipt of Message is split into four subcases:

1. The process has not yet started \mathcal{CLDSA} .

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

2. The process has completed \mathcal{CLDSA} .

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

3. The process has started \mathcal{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] : $M \mid MMS$) 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 \mathcal{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)
⇒
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 \mathcal{CLDSA} . This case is further split into three subcases:

- (a) The \mathcal{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)
⇒
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)
⇒
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)
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 return $x - y$ 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)
⇒
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)

```

where `inchans(BC, P)` initialises the states of all incoming channels of P as `empChan`.

2. The process does not globally initiates \mathcal{CLDSA} . This case is further split into three subcases:

- (a) The process does not have any incoming channels, and there are no processes except for the process that has not completed \mathcal{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)
⇒
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 ≠ empBConfig) .

```

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

```

base-state((p-state[P] : PS) BC)
start-state(SC)

```

```

snapshot(SSC)
control((prog[P] : notYet) (cnt : NzN) (#ms[P] : 0) CC)
⇒
base-state((p-state[P] : PS) bcast(BC, P, marker))
start-state(SC)
snapshot(SSC)
control((prog[P] : completed) (cnt : sd(NzN,1)) (#ms[P] : 0) CC)
if (SC ≠ empBConfig) ∧ (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)
⇒
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 ≠ empBConfig)

```

- Receipt of Marker is split into two subcases:

1. The process has not yet started \mathcal{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 \mathcal{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)
⇒
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 \mathcal{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)
⇒
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 channel.

```

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)
⇒
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 \mathcal{CLDSA} . This case is further split into three sub-cases:

- (a) There are no incoming channels from which markers have not been received except for the incoming channel, and there are no processes that have not yet completed \mathcal{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)
⇒
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 \mathcal{CLDSA} .

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

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

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

The 18 transition rules can be classified into three parts: UDS , $UDS \& \mathcal{CLDSA}$, and \mathcal{CLDSA} . The UDS part consists of the transition rules describing actions 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 $CL(M_{UDS})$. The $UDS \& \mathcal{CLDSA}$ 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 $CL(M_{UDS})$ as well. Three transition rules describing actions iii-2, iii-3 and iii-4 are in the $UDS \& \mathcal{CLDSA}$ part. The \mathcal{CLDSA} part is independent from the UDS concerned, can be constructed regardless of any UDS s, and does not change the base-state meta configuration component of a state of $CL(M_{UDS})$. The transition rules describing two kinds of actions iv and v are in the \mathcal{CLDSA} part.

Definition 6 ($TR_{\mathcal{CLDSA}}$) *Let $TR_{\mathcal{CLDSA}}$ be the set of all ground instances of the 18 transition rules.*

4.2.3 State Machine $CL(M_{UDS})$

In this part, we propose the function CL that takes a state machine $M_{UDS} \triangleq \langle S_{UDS}, I_{UDS}, T_{UDS} \rangle$ and returns another state machine $M_{\mathcal{CLDSA}} \triangleq \langle S_{\mathcal{CLDSA}}, I_{\mathcal{CLDSA}}, T_{\mathcal{CLDSA}} \rangle$. Note that M_{UDS} is a state machine of a UDS and $M_{\mathcal{CLDSA}}$ is a state machine of the UDS superimposed by \mathcal{CLDSA} . The definition of the function CL is as follows:

Definition 7 ($CL(M_{UDS})$) *For a state machine $M_{UDS} \triangleq \langle S_{UDS}, I_{UDS}, T_{UDS} \rangle$ formalizing a UDS , CL is the function that takes M_{UDS} and returns $CL(M_{UDS}) \triangleq \langle CL_{State}(S_{UDS}),$*

$CL_{Init}(I_{UDS}), CL_{Trans}(T_{UDS})\rangle$, where

1. $CL_{State}(S_{UDS})$ is the set of all ground constructor terms of sort $MBConfig$;
2. $CL_{Init}(I_{UDS})$ is
 $\{base-state(bc) \ start-state(empBConfig)$
 $snapshot(empBConfig) \ finish-state(empBConfig)$
 $control(ctl) \mid bc \in I_{UDS}, \ ctl = InitCtlConfig(bc)\}$;
3. $CL_{Trans}(T_{UDS}) \subseteq CL_{State}(S_{UDS}) \times CL_{State}(S_{UDS})$ is $\{(L \ MCF, R \ MCF) \mid L \Rightarrow R$
 $\in TR_{CLDSA}, \ MCF \in MBConfig\}$.

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

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

4.3 Chapter Summary

This chapter described how to formalize a UDS and the UDS superimposed by $CLDSA$ as state machines M_{UDS} and M_{CLDSA} .

The first section of the chapter described how to formalize a UDS as state machines M_{UDS} . It first showed how to express a state of a UDS . The sorts Pid , $PState$, Msg and $MsgQueue$ are used for process identifiers, process states, messages and queues of messages, respectively. The *name-value* pairs (called observable components) are used to express the states of processes and channels, where name may have parameters. The corresponding operators for observable components for process states and channel states were given. The sort $OCom$ is used for observable components. A soup of process and channel states is used to represent each state of M_{UDS} . The corresponding sort is $Config$. Also, the functions on the states of a UDS were given. It then showed how the actions of a UDS are described as transition rules and a transition rule is described in form of rewrite rule. At the end of this section, the definition of state machine M_{UDS} was given.

The chapter then described how to formalize a UDS superimposed by $CLDSA$ as a state machines $CL(M_{UDS})$. Each state of $CL(M_{UDS})$ consists of the local states of all processes and channels, the start state, the snapshot, the finish state and the information to control behaviour of $CLDSA$. The local states of all processes and channels, the start state, the snapshot, the finish state and the control information are expressed as $base-state(bc)$, $start-state(sc)$, $snapshot(ssc)$, $finish-state(fc)$ and $control(ctl)$ meta configuration components, respectively. Each state of $CL(M_{UDS})$ is expressed as the soup of the meta configuration components whose sorts are $MBCom$, which is typically 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 $MBConfig$. The functions on

the states of a $CL(M_{\mathcal{UDS}})$ were presented. Then state transitions for $CL(M_{\mathcal{UDS}})$ was described.

The chapter then defined the function CL that takes a state machine $M_{\mathcal{UDS}}$ and returns the state machine $CL(M_{\mathcal{UDS}})$. Let $M_{\mathcal{CLDSA}}$ is $CL(M_{\mathcal{UDS}})$. $M_{\mathcal{CLDSA}}$ is the state machine of the \mathcal{UDS} superimposed by \mathcal{CLDSA} .

Chapter 5

A More Faithful Definition of the DSR Property

5.1 Some Definitions and Proposition 1

5.1.1 Some Definitions on $S_{\mathcal{CLDSA}}$

Some functions on $S_{\mathcal{CLDSA}}$ are defined as follows:

Definition 8 (*b-state, s-state, snapshot, f-state, finished*) For each $s \in S_{\mathcal{CLDSA}}$, $b\text{-state}(s)$ is bc if there exists exactly one occurrence of the base-state(bc) meta configuration component in s and $empBConfig$ otherwise, $s\text{-state}(s)$ 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, $f\text{-state}(s)$ 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 $f\text{-state}(s)$ is $empBConfig$ and true otherwise.

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

Definition 9 ($M_{\mathcal{CLDSA}} \models terminated(s)$) For a state machine $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$, for each $s \in S_{\mathcal{CLDSA}}$, $M_{\mathcal{CLDSA}} \models terminated(s)$ iff $finished(s)$.

5.1.2 Proposition 1

We have the following proposition on $M_{\mathcal{CLDSA}}$:

Proposition 1 (No marker in s-state, snapshot and f-state) For each $s \in S_{\mathcal{CLDSA}}$, if $M_{\mathcal{CLDSA}} \models terminated(s)$, then there is no marker in $s\text{-state}(s)$, $snapshot(s)$ and $f\text{-state}(s)$, equivalently that the sorts of $s\text{-state}(s)$, $snapshot(s)$ and $f\text{-state}(s)$ are $Config$.

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

Proof of Proposition 1. We will separate Proposition 1 into three parts, then prove them independently.

1. For each $s \in S_{\mathcal{CLDSA}}$, if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, there is no maker in $\text{s-state}(s)$.

It suffices to consider $s \in S_{\mathcal{CLDSA}}$ that can be reach from any initial state in $S_{\mathcal{CLDSA}}$. Then s is in form $\text{base-state}(bc) \text{ start-state}(sc) \text{ snapshot}(ssc) \text{ finish-state}(fc) \text{ control}(ctl)$. if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, $\text{s-state}(s)$ is sc . We will prove that there is no marker in sc . To prove, we consider all transition rules that change the start-state meta configuration component of a state of $M_{\mathcal{CLDSA}}$. Our proof shows that there is no marker in any the start-state meta configuration component in the right hand side of any those transition rules. This means that there is no marker in sc .

All transition rules that change the start-state meta configuration component of a state of $M_{\mathcal{CLDSA}}$ are the following transition rules:

- Record of Process State: the process globally initiates \mathcal{CLDSA} . This case is further split into three subcases:

1. The \mathcal{UDS} only consists of the process.

$$\begin{aligned}
& \text{base-state}((\text{p-state}[P] : PS)) \\
& \text{start-state}(\text{empBConfig}) \\
& \text{snapshot}(\text{empBConfig}) \\
& \text{finish-state}(\text{empBConfig}) \\
& \text{control}((\text{prog}[P] : \text{notYet}) (\text{cnt} : 1) (\#ms[P] : 0) CC) \\
& \Rightarrow \\
& \text{base-state}((\text{p-state}[P] : PS)) \\
& \text{start-state}((\text{p-state}[P] : PS)) \\
& \text{snapshot}((\text{p-state}[P] : PS)) \\
& \text{finish-state}((\text{p-state}[P] : PS)) \\
& \text{control}((\text{prog}[P] : \text{completed}) (\text{cnt} : 0) (\#ms[P] : 0) CC)
\end{aligned}$$

Since the process globally initiates \mathcal{CLDSA} , there is no-marker in $(\text{p-state}[P] : PS)$. Therefore, there is no marker in $\text{start-state}((\text{p-state}[P] : PS))$.

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

$$\text{base-state}((\text{p-state}[P] : PS) BC)$$

```

start-state(empBConfig)
snapshot(empBConfig)
finish-state(empBConfig)
control((prog[P] : notYet) (cnt : NzN) (#ms[P] : 0) CC)
⇒
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)
if NzN > 1

```

Since the process globally initiates \mathcal{CLDSA} , there is no-marker in (p-state[P] : PS) BC. Therefore, there is no marker in start-state((p-state[P] : PS) BC).

3. 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)
⇒
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)

```

Since the process globally initiates \mathcal{CLDSA} , there is no-marker in (p-state[P] : PS) BC. Therefore, there is no marker in start-state((p-state[P] : PS) BC).

Our proof have considered all transition rules that change the start-state meta configuration component of a state of $M_{\mathcal{CLDSA}}$ and showed that there is no marker in the start-state meta configuration component after apply those transition rules. This case is discharged.

2. For each $s \in S_{\mathcal{CLDSA}}$, if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, there is no maker in snapshot(s).

It suffices to consider $s \in S_{\mathcal{CLDSA}}$ that can be reach from any initial state in $S_{\mathcal{CLDSA}}$. Then s is in form base-state(bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl). if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, snapshot(s) is ssc . We will prove that there is no marker in ssc . To prove, we consider all transition rules that change the snapshot meta configuration component of a state of $M_{\mathcal{CLDSA}}$. Our proof shows that those transition rules do not put

any marker into snapshot meta configuration component of a state of $M_{\mathcal{CLDSA}}$ when they change the snapshot meta configuration component, this means that there is no marker in the snapshot.

All the transition rules that change the snapshot meta configuration component of a state of $M_{\mathcal{CLDSA}}$ are the following transition rules:

- **Receipt of Message** is split into four subcases. The subcase that change the snapshot meta configuration component of a state of $M_{\mathcal{CLDSA}}$ is as follows:

1. The process has started \mathcal{CLDSA} , not yet completed it, and not yet received a marker from the incoming channel.

$$\begin{aligned}
& \text{base-state}((\text{p-state}[P] : PS1) (\text{c-state}[Q, P, N] : M \mid MMS) BC) \\
& \text{snapshot}((\text{c-state}[Q, P, N] : MMS') SSC) \\
& \text{control}((\text{prog}[P] : \text{started}) (\text{done}[Q, P, N] : \text{false}) CC) \\
& \Rightarrow \\
& \text{base-state}((\text{p-state}[P] : PS2) (\text{c-state}[Q, P, N] : MMS) BC) \\
& \text{snapshot}((\text{c-state}[Q, P, N] : \text{enq}(MMS', M)) SSC) \\
& \text{control}((\text{prog}[P] : \text{started}) (\text{done}[Q, P, N] : \text{false}) CC)
\end{aligned}$$

where $M \in \text{Msg}$. Because M is a message, the transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}((\text{c-state}[Q, P, N] : MMS') SSC) \Rightarrow \text{snapshot}((\text{c-state}[Q, P, N] : \text{enq}(MMS', M)) SSC)$

- **Record of Process State** is split into two subcases:

1. The process globally initiates \mathcal{CLDSA} . This case is further split into three subcases:
 - (a) The \mathcal{UDS} only consists of the process.

$$\begin{aligned}
& \text{base-state}((\text{p-state}[P] : PS)) \\
& \text{start-state}(\text{empBConfig}) \\
& \text{snapshot}(\text{empBConfig}) \\
& \text{finish-state}(\text{empBConfig}) \\
& \text{control}((\text{prog}[P] : \text{notYet}) (\text{cnt} : 1) (\#ms[P] : 0) CC) \\
& \Rightarrow \\
& \text{base-state}((\text{p-state}[P] : PS)) \\
& \text{start-state}((\text{p-state}[P] : PS)) \\
& \text{snapshot}((\text{p-state}[P] : PS)) \\
& \text{finish-state}((\text{p-state}[P] : PS)) \\
& \text{control}((\text{prog}[P] : \text{completed}) (\text{cnt} : 0) (\#ms[P] : 0) CC)
\end{aligned}$$

Since the process globally initiates \mathcal{CLDSA} , there is no marker in $(\text{p-state}[P] :$

PS). The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(\text{empBConfig}) \Rightarrow \text{snapshot}((\text{p-state}[P] : PS))$.

- (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)
⇒
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)
if NzN > 1

```

Since the process globally initiates \mathcal{CLDSA} , there is no marker in $(\text{p-state}[P] : PS) BC$. The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(\text{empBConfig}) \Rightarrow \text{snapshot}((\text{p-state}[P] : PS) BC)$.

- (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)
⇒
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)

```

where $\text{inchans}(BC, P)$ initialises the states of all incoming channels of P as empChan . The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(\text{empBConfig}) \Rightarrow \text{snapshot}((\text{p-state}[P] : PS) \text{inchans}(BC, P))$.

2. The process does not globally initiates \mathcal{CLDSA} . This case is further split into three subcases:

- (a) The process does not have any incoming channels, and there are no processes

except for the process that has not completed \mathcal{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)
⇒
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 ≠ empBConfig) .

```

It is clear to see that the transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(SSC) \Rightarrow \text{snapshot}((p\text{-state}[P] : PS) SSC)$.

(b) 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)
⇒
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 ≠ empBConfig)

```

It is clear to see that the transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(SSC) \Rightarrow \text{snapshot}((p\text{-state}[P] : PS) \text{inchans}(BC, P) SSC)$.

- **Receipt of Marker** is split into two subcases:

1. The process has not yet started \mathcal{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 \mathcal{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)

Since there is no marker in (c-state[Q, P, N] : empChan). The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(SSC) \Rightarrow \text{snapshot}((p\text{-state}[P] : PS) (c\text{-state}[Q, P, N] : \text{empChan}) SSC)$.

- (b) The process has only one incoming channel, and there are some other processes that have not yet completed \mathcal{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

Since there is no marker in (c-state[Q, P, N] : empChan). The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(SSC) \Rightarrow \text{snapshot}((p\text{-state}[P] : PS) (c\text{-state}[Q, P, N] : \text{empChan}) SSC)$.

- (c) The process has more than one incoming channel.

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

Since there is no marker in (c-state[Q, P, N] : empChan) inchans(BC, P).

The transition rule does not put any marker in the the snapshot meta configuration component: $\text{snapshot}(SSC) \Rightarrow \text{snapshot}((\text{p-state}[P] : PS) \text{ (c-state}[Q, P, N] : \text{empChan}) \text{ inchans}(BC, P) SSC)$.

3. For each $s \in S_{\mathcal{CLDSA}}$, if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, there is no marker in $\text{f-state}(s)$.

It suffices to consider $s \in S_{\mathcal{CLDSA}}$ that can be reach from any initial state in $S_{\mathcal{CLDSA}}$. Then s is in form $\text{base-state}(bc) \text{ start-state}(sc) \text{ snapshot}(ssc) \text{ finish-state}(fc) \text{ control}(ctl)$. if $M_{\mathcal{CLDSA}} \models \text{terminated}(s)$, $\text{f-state}(s)$ is fc . We will prove that there is no marker in fc . To prove, we consider all transition rules that change the finish-state meta configuration component of a state of $M_{\mathcal{CLDSA}}$. Our proof shows that there is no marker in any the finish-state meta configuration component in the right hand side of any those transition rules. This means that there is no marker in fc .

All transition rules that change the finish state meta configuration component of a state of $M_{\mathcal{CLDSA}}$ are the following transition rules:

- **Record of Process State** is split into two subcases:

1. The process globally initiates \mathcal{CLDSA} . This case is further split into three subcases. The subcase that change the snapshot meta configuration component of a state of $M_{\mathcal{CLDSA}}$ is as follows:

(a) The \mathcal{UDS} only consists of the process.

$$\begin{aligned} & \text{base-state}((\text{p-state}[P] : PS)) \\ & \text{start-state}(\text{empBConfig}) \\ & \text{snapshot}(\text{empBConfig}) \\ & \text{finish-state}(\text{empBConfig}) \\ & \text{control}((\text{prog}[P] : \text{notYet}) \text{ (cnt : 1) (\#ms}[P] : 0) CC) \\ \Rightarrow & \\ & \text{base-state}((\text{p-state}[P] : PS)) \\ & \text{start-state}((\text{p-state}[P] : PS)) \\ & \text{snapshot}((\text{p-state}[P] : PS)) \\ & \text{finish-state}((\text{p-state}[P] : PS)) \\ & \text{control}((\text{prog}[P] : \text{completed}) \text{ (cnt : 0) (\#ms}[P] : 0) CC) \end{aligned}$$

Since there is no marker in $(\text{p-state}[P] : PS)$. There is no marker in $\text{finish-state}((\text{p-state}[P] : PS))$. Therefore, there is no marker in the finish-state meta configuration component in the right hand side of any the transition rule: $\text{finish-state}(\text{empBConfig}) \Rightarrow \text{finish-state}((\text{p-state}[P] : PS))$.

2. The process does not globally initiates \mathcal{CLDSA} . This case is further split into three subcases:

- (a) The process does not have any incoming channels, and there are no processes except for the process that has not completed \mathcal{CLDSA} .

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

Since there is no marker in $(\text{p-state}[P] : PS) BC$. There is no marker in $\text{finish-state}((\text{p-state}[P] : PS))$. Therefore, there is no marker in the finish-state meta configuration component in the right hand side of any the transition rule: $\text{finish-state}(\text{empBConfig}) \Rightarrow \text{finish-state}((\text{p-state}[P] : PS) BC)$.

- **Receipt of Marker** is split into two subcases:

- (a) The process has not yet started \mathcal{CLDSA} . This case is further split into three subcases:
- i. The process has only one incoming channel, and there are no processes that have not yet completed \mathcal{CLDSA} except for the process.

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

Because the process has only one incoming channel, and there are no processes that have not yet completed \mathcal{CLDSA} except for the process, there is no marker in BC . Then there is no marker in $\text{finish-state}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : MMS) BC)$.

- (b) The process has already started \mathcal{CLDSA} . This case is further split into three subcases:

- i. There are no incoming channels from which markers have not been received except for the incoming channel, and there are no processes that have not yet completed $\mathcal{CLDS}\mathcal{A}$ except for the process.

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

Because There are no incoming channels from which markers have not been received except for the incoming channel, and there is no process that has not yet completed $\mathcal{CLDS}\mathcal{A}$ except for the process, there is no marker in BC . Then there is no marker in $\text{finish-state}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : MMS) BC)$.

The case has been discharged.

From the three parts are proven, Proposition 1 is proven. QED.

We have already given the proof of Proposition 1.

5.2 A More Faithful Definition of the \mathcal{DSR} Property

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

Definition 10 (Reachabilty in M) For a state machine $M \triangleq \langle S, I, T \rangle$, for each $\pi \in \Pi$ and each $s_1, s_2 \in S$,
 $M, \pi \models \text{isReachable}(s_2, s_1)$ iff $(\exists i, j \in \text{Nat}) (i \leq j \wedge s_1 = \pi_i \wedge s_2 = \pi_j)$, and $M \models \text{isReachable}(s_2, s_1)$ iff $(\exists \pi \in \Pi) (M, \pi \models \text{isReachable}(s_2, s_1))$.

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 \mathcal{DSR} property, it is checked that $\mathcal{CLDS}\mathcal{A}$ terminates, and it is checked that some states of a \mathcal{UDS} are reachable from some others in the \mathcal{UDS} but not the \mathcal{UDS} superimposed by $\mathcal{CLDS}\mathcal{A}$. Accordingly, the property involves two systems, a \mathcal{UDS} and the \mathcal{UDS} superimposed by $\mathcal{CLDS}\mathcal{A}$, and hence we need to use two state machines $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDS}\mathcal{A}}$ to faithfully define the \mathcal{DSR} property. Our definition of the \mathcal{DSR} property is as follows:

Definition 11 (*DSR Property*) For a state machine $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$,
 $(\forall s \in S_{\mathcal{CLDSA}})$
 $(M_{\mathcal{CLDSA}} \models \text{terminated}(s) \Rightarrow M_{\mathcal{UDS}} \models \text{isReachable}(s_*, s_1) \wedge M_{\mathcal{UDS}} \models \text{isReachable}(s_2, s_*))$,
 where $s_1 = s\text{-state}(s)$, $s_* = \text{snapshot}(s)$ and $s_2 = f\text{-state}(s)$.

5.3 Chapter Summary

This chapter gave the more faithful formal definition of the \mathcal{DSR} property. It first gave the definitions of functions b-state, s-state, snapshot and f-state corresponding to the local states of all processes and channels, the start-state, the snapshot, the finish-state of $M_{\mathcal{CLDSA}}$, and function finished for termination of \mathcal{CLDSA} . It then gave Proposition 1 saying that whenever \mathcal{CLDSA} terminates in state s , there is no marker in the start state, the snapshot and the finish state.

The chapter then gave the proof of Proposition 1, which is separated into three parts: whenever \mathcal{CLDSA} terminates in state s , there is no marker in the start state, whenever \mathcal{CLDSA} terminates in state s , there is no marker in the snapshot and whenever \mathcal{CLDSA} terminates in state s , there is no marker in the finish state.

At the end of the chapter, the more faithful formal definition of the \mathcal{DSR} property was given, which involves two state machines $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$. The definition more faithfully expresses the informal description of the \mathcal{DSR} property.

Chapter 6

Equivalence of Two Definitions of the DSR Property

6.1 The Theorem on Equivalence of the Two Definitions and Some Lemmas

Since our new definition of the DSR property involves two state machines, it is not straightforward to directly model check the new definition of the DSR property for \mathcal{CLDSA} with any existing model checker. If the new definition is equivalent to the definition used in the existing study for each UDS , we can use the model checking approach used in the existing study to model check the new definition of the DSR property for \mathcal{CLDSA} . Therefore, we prove a theorem saying that our new definition is equivalent to the definition in the existing study, which also guarantees the validity of the model checking approach used in the existing study.

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 $\{p_1, \dots, p_n\}$, where $n \geq 1$. Let ctl be $(prog[p_1] : notYet) \dots (prog[p_n] : notYet)$ in the rest of the paper.

Although the DSR property is encoded in terms of the Maude search command, the definition of DSR property in the existing study can be expressed as follows:

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

Checked are the termination of \mathcal{CLDSA} and then the reachability from the snapshot and the final state to the start state and the snapshot, respectively, in a UDS superimposed by \mathcal{CLDSA} but not in the UDS . Their definition of the DSR property only involves $M_{\mathcal{CLDSA}}$, while the new definition involves M_{UDS} and $M_{\mathcal{CLDSA}}$. Hence, both definitions are seemingly different. However, we reveal that the new definition coincides with theirs by proving a theorem in this section. The theorem on the equivalence of two definitions

is as follows:

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

Only the difference between the new definition of the \mathcal{DSR} property and the definition used in the existing study is the conclusion parts of the implications. If the conclusion parts are equivalent, then the two definitions are equivalent. The equivalence of the conclusion parts means that reachability is preserved between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDS}_A}$. The lemma on reachability preservation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDS}_A}$ is as follows:

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

We first prove two more lemmas, which say that one-step reachability is preserved between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDS}_A}$, to prove Lemma 1. The two lemmas are as follows:

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

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

For each \mathcal{UDS} , $T_{\mathcal{UDS}}$ is constructed from the three transition rules and $T_{\mathcal{CLDS}_A}$ 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 $T_{\mathcal{UDS}}$ and $T_{\mathcal{CLDS}_A}$, respectively. In the following proofs, $p, q \in \text{Pid}$, $ps_1, ps_2 \in \text{PState}$, $cs \in \text{MsgQueue}$, $m \in \text{Msg}$, $bc \in \text{Config}$ and $n \in \text{Nat}$ are fresh constants of those sorts.

6.2 Proof of Lemma 2.

Assume that s_1 goes to s_2 by a state transition t in $M_{\mathcal{UDS}}$. Our proof shows that there exists a state transition t' in $M_{\mathcal{CLDS}_A}$ that moves $\text{base-state}(s_1) \text{ control}(\text{ctl})$ to $\text{base-state}(s_2) \text{ control}(\text{ctl})$. In our proof, we consider all possible state transition t in $M_{\mathcal{UDS}}$ that can move s_1 to s_2 . As we mentioned above, it suffices to take into account the three transition rules that describes Change of Process State, Sending of Message and Receipt of Message

in $M_{\mathcal{UDS}}$.

1. Change of Process State. Let us consider the case in which t is constructed from the transition rule that describes Change of Process State in $M_{\mathcal{UDS}}$.

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

It suffices to consider s_1 as an arbitrary state $(\text{p-state}[p] : ps_1) \text{ bc}$ in $S_{\mathcal{UDS}}$ to which the transition rule can be applied. Therefore, s_2 is $(\text{p-state}[p] : ps_2) \text{ bc}$. Then, $\text{base-state}(s_1) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_1) \text{ bc}) \text{ control}(ctl)$, and $\text{base-state}(s_2) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_2) \text{ bc}) \text{ control}(ctl)$. The transition rule that describes Change of Process State in $M_{\mathcal{CLDS}_A}$ can be applied to $\text{base-state}(s_1) \text{ control}(ctl)$ and obtains $\text{base-state}(s_2) \text{ control}(ctl)$.

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

Hence, there exists t' . The case has been discharged.

2. Sending of Message. Let us consider the case in which t is constructed from the transition rule that describes Sending of Message in $M_{\mathcal{UDS}}$.

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

It suffices to consider s_1 as an arbitrary state $(\text{p-state}[p] : ps_1) (\text{c-state}[p, q, n] : cs) \text{ bc}$ in $S_{\mathcal{UDS}}$ to which the transition rule can be applied. Therefore, s_2 is $(\text{p-state}[p] : ps_2) (\text{c-state}[p, q, n] : \text{enq}(cs, m)) \text{ bc}$. Then, $\text{base-state}(s_1) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_1) (\text{c-state}[p, q, n] : cs) \text{ bc}) \text{ control}(ctl)$, and $\text{base-state}(s_2) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_2) (\text{c-state}[p, q, n] : \text{enq}(cs, m)) \text{ bc}) \text{ control}(ctl)$. The transition rule that describes Sending of Message in $M_{\mathcal{CLDS}_A}$ can be applied to $\text{base-state}(s_1) \text{ control}(ctl)$ and obtains $\text{base-state}(s_2) \text{ control}(ctl)$.

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

Hence, there exists t' . The case has been discharged.

3. Receipt of Message. Let us consider the case in which t is constructed from the transition rule that describes Receipt of Message in $M_{\mathcal{UDS}}$.

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

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

It suffices to consider s_1 as an arbitrary state $(\text{p-state}[p] : ps_1) (\text{c-state}[q, p, n] : m \mid cs) bc$ in S_{UDS} to which the transition rule can be applied. Therefore, s_2 is $(\text{p-state}[p] : ps_2) (\text{c-state}[q, p, n] : cs) bc$. Then, $\text{base-state}(s_1) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_1) (\text{c-state}[p, q, n] : m \mid cs) bc) \text{ control}(ctl)$, and $\text{base-state}(s_2) \text{ control}(ctl)$ is $\text{base-state}((\text{p-state}[p] : ps_2) (\text{c-state}[p, q, n] : cs) bc) \text{ control}(ctl)$. The transition rule that describes Receipt of Message in M_{CLDS_A} can be applied to $\text{base-state}(s_1) \text{ control}(ctl)$ and obtains $\text{base-state}(s_2) \text{ control}(ctl)$.

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

Hence, there exists t' . The case has been discharged.

We have considered all possible state transition t in M_{UDS} that can move s_1 to s_2 . All cases has been discharged. QED

6.3 Proof of Lemma 3.

Assume that $\text{base-state}(s_1) \text{ control}(ctl)$ goes to $\text{base-state}(s_2) \text{ control}(ctl)$ by a state transition t in M_{CLDS_A} . Because $s_1 \in S_{UDS}$, there is no marker in s_1 . This is why any of the transition rules that describe Record of Process State and Receipt of Marker in M_{CLDS_A} cannot be applied to $\text{base-state}(s_1) \text{ control}(ctl)$. 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_{CLDS_A} cannot be applied to $\text{base-state}(s_1) \text{ control}(ctl)$, either. Therefore, t is not a state transition constructed from those transition rules, either. 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_{CLDS_A} . 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 $\text{base-state}(s_1) \text{ control}(ctl)$ to $\text{base-state}(s_2) \text{ control}(ctl)$ in M_{CLDS_A} . The following proof considers all the possible transition rules as what we mentioned above.

1. Change of Process State. Let us consider the case in which t is constructed from the transition rule that describes Change of Process State in M_{CLDS_A} .

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

It suffices to consider $\text{base-state}(s_1) \text{ control}(ctl)$ as an arbitrary state $\text{base-state}((\text{p-state}[p]$

: ps_1) bc) $control(ctl)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied. Therefore, $base-state(s_2) control(ctl)$ is $base-state((p-state[p]: ps_2) bc) control(ctl)$. Then, s_1 is $(p-state[p]: ps_1) bc$, and s_2 is $(p-state[p]: ps_2) bc$. The transition rule that describes Change of Process State in $M_{\mathcal{UDS}}$ can be applied to s_1 and obtains s_2 .

$$(p-state[P] : PS1) \Rightarrow (p-state[P] : PS2)$$

Hence, there exists t' . The case has been discharged.

2. Sending of Message. Let us consider the case in which t is constructed from the transition rule that describes Sending of Message in $M_{\mathcal{CLDSA}}$.

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

It suffices to consider $base-state(s_1) control(ctl)$ as an arbitrary state $base-state((p-state[p] : ps_1) (c-state[p, q, n]: cs) bc) control(ctl)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied. Therefore, $base-state(s_2) control(ctl)$ is $base-state((p-state[p]: ps_2) (c-state[p, q, n]: enq(cs, m)) bc) control(ctl)$. Then, s_1 is $(p-state[p]: ps_1) (c-state[p, q, n]: cs) bc$, and s_2 is $(p-state[p]: ps_2) (c-state[p, q, n]: enq(cs, m)) bc$. The transition rule that describes Sending of Message in $M_{\mathcal{UDS}}$ can be applied to s_1 and obtains s_2 .

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

Hence, there exists t' . The case has been discharged.

3. Receipt of Message.

a. The process has not yet started the CLDSA. Let us consider the case in which t is constructed from the transition rule that describes 1st case of Receipt of Message in $M_{\mathcal{CLDSA}}$.

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

It suffices to consider $base-state(s_1) control(ctl)$ as an arbitrary state $base-state((p-state[p] : ps_1) (c-state[p, q, n]: m \mid cs) bc) control(ctl)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied. Therefore, $base-state(s_2) control(ctl)$ is $base-state((p-state[p]: ps_2) (c-state[p, q, n]: cs) bc) control(ctl)$. Note that because ctl is $(prog[p_1] : notYet) \dots (prog[p_n] : notYet)$, \mathcal{CLDSA} has not yet started. There is no marker in cs . Then, s_1 is $(p-state[p]:$

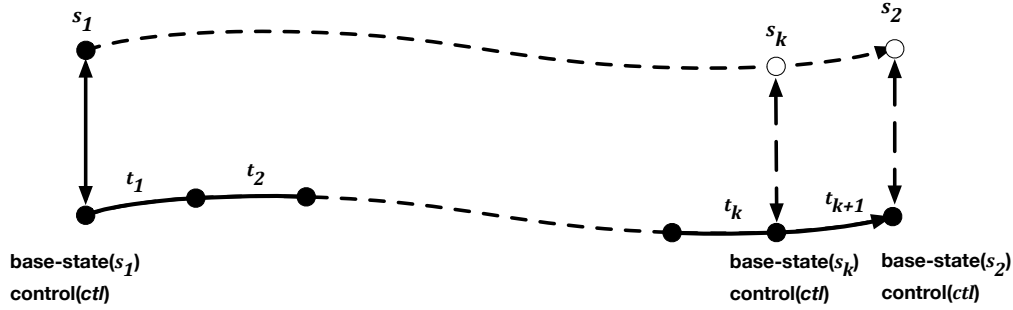


Figure 6.1: The correspondence from the transitions in M_{CCDS_A} to the transitions in M_{UDS} .

ps_1) (c-state[q, p, n]: $m \mid cs$) bc , and s_2 is (p-state[p]: ps_2) (c-state[q, p, n]: cs) bc . The transition rule that describes Receipt of Message in M_{CCDS_A} can be applied to s_1 and obtains s_2 .

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

\Rightarrow

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

Hence, there exists t' . The case has been discharged.

We have considered all possible state transition t in M_{UDS} that can move $\text{base-state}(s_1) \text{ control}(ctl)$ to $\text{base-state}(s_2) \text{ control}(ctl)$. All cases has been discharged. QED

6.4 Proof of Lemma 1

6.4.1 Proof of the “if” part of Lemma 1

We prove that $\forall s_1, s_2 \in S_{UDS}$, if $M_{CCDS_A} \models \text{isReachable}(\text{base-state}(s_2) \text{ control}(ctl), \text{base-state}(s_1) \text{ control}(ctl))$, then $M_{UDS} \models \text{isReachable}(s_2, s_1)$.

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

- Base case: since $\text{base-state}(s_1) \text{ control}(ctl)$ is the same as $\text{base-state}(s_2) \text{ control}(ctl)$ in this case, s_1 is the same as s_2 . So, this case is discharged.

- Induction case: suppose that $\text{base-state}(s_1) \text{ control}(ctl)$ moves to $\text{base-state}(s_2) \text{ control}(ctl)$ by $k+1$ transition steps and the $k+1$ transitions taken are t_1, \dots, t_{k+1} . As shown in Fig. 3, $\text{base-state}(s_k) \text{ control}(ctl)$ is the state to which $\text{base-state}(s_1) \text{ control}(ctl)$ moves by the first k transition steps, namely that $M_{CCDS_A} \models \text{isReachable}(\text{base-state}(s_k) \text{ control}(ctl), \text{base-state}(s_1) \text{ control}(ctl))$. From the induction hypothesis, $M_{UDS} \models \text{isReachable}(s_k, s_1)$.

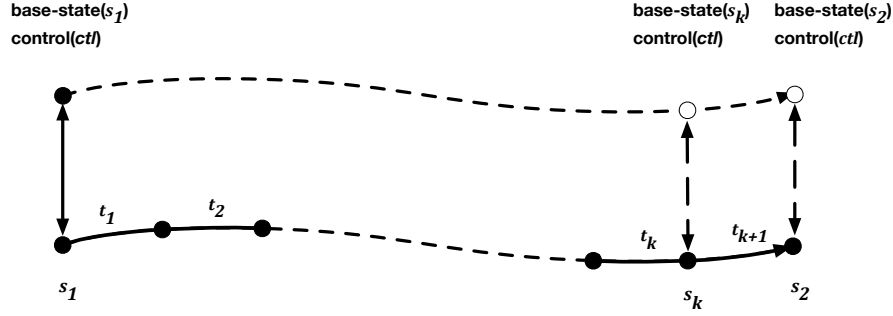


Figure 6.2: The correspondence from the transitions in M_{UDS} to the transitions in M_{CLDSA} .

Since $\text{base-state}(s_k) \text{ control}(ctrl)$ moves to $\text{base-state}(s_2) \text{ control}(ctrl)$ by one transition step in M_{CLDSA} , s_k also moves to s_2 by one transition step in M_{UDS} from Lemma 3. Then, this case is also discharged. Fig. 6.1 shows the correspondence from the transitions in M_{CLDSA} to transitions in M_{UDS} . QED

6.4.2 Proof of the “only if” part of Lemma 1

We prove that $\forall s_1, s_2 \in S_{UDS}$, if $M_{UDS} \models \text{isReachable}(s_2, s_1)$, then $(M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_2) \text{ control}(ctrl), \text{base-state}(s_1) \text{ control}(ctrl)))$.

Assume that $M_{UDS} \models \text{isReachable}(s_2, s_1)$ and then there must be 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 .

- Base case: since s_1 is the same as s_2 in this case, $\text{base-state}(s_1) \text{ control}(ctrl)$ is the same as $\text{base-state}(s_2) \text{ control}(ctrl)$. So, this case is discharged.

- Induction case: suppose that s_1 moves to s_2 by $k+1$ transition steps and the $k+1$ transitions taken are t_1, \dots, t_{k+1} . As shown in Fig. 4, s_k is the state to which s_1 moves by the first k transition steps, namely that $M_{UDS} \models \text{isReachable}(s_k, s_1)$. From the induction hypothesis, $M_{CLDSA} \models \text{isReachable}(\text{base-state}(s_k) \text{ control}(ctrl), \text{base-state}(s_2) \text{ control}(ctrl))$. Since s_k moves to s_2 by one transition step in M_{CLDSA} , $\text{base-state}(s_k) \text{ control}(ctrl)$ also moves to $\text{base-state}(s_2) \text{ control}(ctrl)$ by one transition step in M_{CLDSA} from Lemma 2. Then, this case is also discharged. Fig. 6.2 shows the correspondence from the transitions in M_{UDS} to transitions in M_{CLDSA} . QED

6.5 Proof of Theorem 1

Since Proposition 1 says that whenever $CLDSA$ terminates in state s , there is no marker in the start state, the snapshot and the finish state, and Lemma 1 asserts that reachability is preserved between M_{UDS} and M_{CLDSA} . Proof of Theorem 1 follows from Proposition 1 and Lemma 1. QED

6.6 Chapter Summary

This chapter gave the theorem on equivalence of the two definitions, the more faithful formal definition and the definition in the existing study, and some lemmas. Also, the proofs of them were given.

The chapter first represented the definition of \mathcal{DSR} property in the existing study, which is encoded in terms of the Maude search command in the existing study. Then the analysis of the similarities between the new definition of the \mathcal{DSR} property and the existing definition was conducted.

The chapter then gave Theorem 1 saying that new definition is equivalent to the definition of the \mathcal{DSR} property in the existing study. To prove the Theorem 1, it proved Proposition 1 and Lemma 1. Lemma 1 asserts that reachability is preserved between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$. Proposition 1 says that whenever \mathcal{CLDSA} terminates in state s , there is no marker in the start state, the snapshot and the finish state. We prove as Lemma 2 and Lemma 3 that one-step reachability is preserved between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ to prove Lemma 1.

At the end of the chapter, The proof of Theorem 1 was given following from Proposition 1 and Lemma 1.

Chapter 7

\mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS}

7.1 A Binary Relation Between Two State Machines $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$

Since \mathcal{CLDSA} works by using a special message called marker, \mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS} means that excepting for putting markers in a \mathcal{UDS} , the algorithm does not change the state of all processes and channels. We propose a binary relation \mathbf{r} between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ saying that for each $s1 \in S_{\mathcal{UDS}}$ and each $s2 \in S_{\mathcal{CLDSA}}$, $\mathbf{r}(s1, s2)$ if and only if $s1$ is the same as the state obtained by deleting all markers from $s2$.

Definition 12 (Binary relation \mathbf{r}) Given two state machines $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$ and $M_{\mathcal{CLDSA}} \triangleq \langle S_{\mathcal{CLDSA}}, I_{\mathcal{CLDSA}}, T_{\mathcal{CLDSA}} \rangle$, $\forall s_{\mathcal{UDS}} \in S_{\mathcal{UDS}}$ and $\forall s_{\mathcal{CLDSA}} \in S_{\mathcal{CLDSA}}$: A binary relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ $\mathbf{r} : S_{\mathcal{UDS}} \times S_{\mathcal{CLDSA}} \rightarrow \text{Bool}$.

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

Note that: For $\forall s_{\mathcal{UDS}} \in S_{\mathcal{UDS}}$, $s_{\mathcal{UDS}} = \text{delM}(s_{\mathcal{UDS}})$ because there is no marker in $s_{\mathcal{UDS}}$.

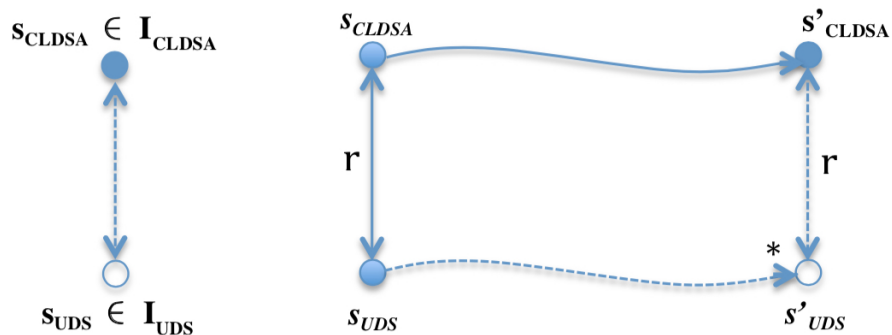


Figure 7.1: The binary relation \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$.

7.2 Theorem 2 and the Proof of It

To guarantee that \mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS} , we prove that $M_{\mathcal{CLDSA}}$ simulates $M_{\mathcal{UDS}}$ and vice versa. To prove that we prove that exists a bi-simulation relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$. Therefore, we prove Theorem 2 saying that \mathbf{r} is a bi-simulation relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$.

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

To prove Theorem 2, we shall prove that \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$ and it is also a simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$.

Since, binary relation \mathbf{r} is considered as a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$ ($M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$). For two state machines, we only consider states what can be reached from any initial states, which has specific configuration and the configuration is never changed by any state transition. This is used to assume the configuration of states of $M_{\mathcal{CLDSA}}$ and states of $M_{\mathcal{UDS}}$ in the following proof.

7.2.1 Simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$.

To prove that \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$, We will prove that \mathbf{r} satisfies the following two conditions. Fig. 7.1 shows the diagrams corresponding to the two conditions.

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

Prove:

For each $s_{\mathcal{CLDSA}} \in CL_{Init}(I_{\mathcal{UDS}})$, from the definition of CL , we have:

$s_{\mathcal{CLDSA}} = \text{base-state}(bc) \text{ start-state}(\text{empConfig}) \text{ snapshot}(\text{empConfig})$
 $\text{finsh-state}(\text{empConfig}) \text{ control}(\text{ctl})$, where $bc \in I_{\mathcal{UDS}}$, $\text{ctl} = \text{InitCtlConfig}(bc)$.

We have $\text{b-state}(s_{\mathcal{CLDSA}}) = bc$. Let $s_{\mathcal{UDS}} = bc \in I_{\mathcal{UDS}}$. Since, $s_{\mathcal{UDS}} = bc$ and $\text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}(s_{\mathcal{UDS}}) = bc$, $s_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s_{\mathcal{CLDSA}}))$. Then $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$.

Therefore, for each $s_{\mathcal{CLDSA}} \in I_{\mathcal{CLDSA}}$ there exists $s_{\mathcal{UDS}} \in I_{\mathcal{UDS}}$, such that $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$. This condition is satisfied.

2. For each $s_{\mathcal{CLDSA}}, s'_{\mathcal{CLDSA}} \in S_{\mathcal{CLDSA}}$ and $s_{\mathcal{UDS}} \in S_{\mathcal{UDS}}$ such that $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$ and $s_{\mathcal{CLDSA}} \rightsquigarrow_{M_{\mathcal{CLDSA}}} s'_{\mathcal{CLDSA}}$ by one state transition of the $M_{\mathcal{CLDSA}}$, there exists $s'_{\mathcal{UDS}}$ such that $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{UDS}} \rightsquigarrow^*_{M_{\mathcal{UDS}}} s'_{\mathcal{UDS}}$ by zero or more state transitions of the $M_{\mathcal{UDS}}$.

Prove:

In the following proofs, $p, q \in \text{Pid}$, $ps_1, ps_2 \in \text{PState}$, $cs \in \text{MMsgQueue}$, $m \in \text{Msg}$, $bc, sc, ssc, fc \in \text{BConfig}$, $ctl \in \text{CtlConfig}$ and $n \in \text{Nat}$ are fresh constants of those sorts. As we said before, for two state machines, we only consider states what can be reached from any initial states. The configuration of a state of $M_{\mathcal{CLDSA}}$ is as following:

$s_{\mathcal{CLDSA}} = \text{base-state}(bc) \text{ start-state}(sc) \text{ snapshot}(ssc) \text{ finish-state}(fc) \text{ control}(ctl)$, where $bc, sc, ssc, fc \in \text{BConfig}$ and $ctl \in \text{CtlConfig}$.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}(bc)$.

Assume that $s_{\mathcal{CLDSA}} \rightsquigarrow_{M_{\mathcal{CLDSA}}} s'_{\mathcal{CLDSA}}$ by state transition t .

The same as what we have mentioned before. For each \mathcal{UDS} , $T_{\mathcal{UDS}}$ is constructed from the three transition rules and $T_{\mathcal{CLDSA}}$ 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 $T_{\mathcal{UDS}}$ and $T_{\mathcal{CLDSA}}$, respectively. Since, the 18 transition rules can be classified into three parts: \mathcal{UDS} , $\mathcal{UDS} \& \mathcal{CLDSA}$, and \mathcal{CLDSA} . The \mathcal{UDS} part consists of the transition rules describing actions i, ii and iii-1. Three transition rules describing actions iii-2, iii-3 and iii-4 are in the $\mathcal{UDS} \& \mathcal{CLDSA}$ part. The transition rules describing two kinds of actions iv and v are in the \mathcal{CLDSA} part. No loss of generality, in our proof, state transition t is separated into three parts: \mathcal{UDS} , $\mathcal{UDS} \& \mathcal{CLDSA}$, and \mathcal{CLDSA} . In the following proofs, $p, q \in \text{Pid}$, $ps_1, ps_2 \in \text{PState}$, $cs \in \text{MsgQueue}$, $m \in \text{Msg}$, $bc, sc, ssc \in \text{BConfig}$, $ctl \in \text{CtlConfig}$ and $n \in \text{Nat}$ are fresh constants of those sorts. To prove that this condition is satisfied, we will separate the proof in two parts depended on the transition rules.

The \mathcal{UDS} and $\mathcal{UDS} \& \mathcal{CLDSA}$ part: the transition t constructed from the transition rules in the \mathcal{UDS} part and the $\mathcal{UDS} \& \mathcal{CLDSA}$ part.

Since, the \mathcal{UDS} part depends on the \mathcal{UDS} concerned, can be constructed from the three transition rules of the \mathcal{UDS} and changes the base-state meta configuration component of a state of $M_{\mathcal{CLDSA}}$. The $\mathcal{UDS} \& \mathcal{CLDSA}$ part also depends on the \mathcal{UDS} concerned and can be constructed from the three transition rules of the \mathcal{UDS} , but changes the other meta configuration components of a state of $CL(M_{\mathcal{UDS}})$ as well. For any state transition t in \mathcal{UDS} and $\mathcal{UDS} \& \mathcal{CLDSA}$ part that moves state $s_{\mathcal{CLDSA}}$ to state $s'_{\mathcal{CLDSA}}$ in $M_{\mathcal{CLDSA}}$. We can find state transition t' in $M_{\mathcal{UDS}}$ that can moves state $s_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s_{\mathcal{CLDSA}}))$ to state $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}}))$ in $M_{\mathcal{UDS}}$. This will be proven by the following. Existing t' in $M_{\mathcal{UDS}}$ corresponding t is shown in Fig .7.2.

1. **Change of Process State.** Let us consider the case in which t is constructed from the transition rule that describes Change of Process State in $M_{\mathcal{CLDSA}}$.

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

\Rightarrow

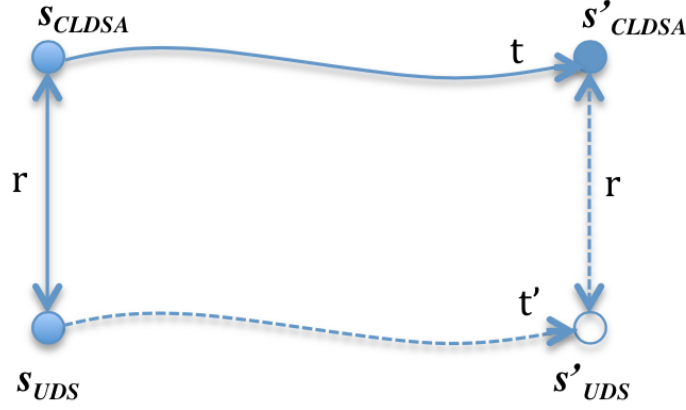


Figure 7.2: Existing t' in M_{UDS} corresponding t .

base-state((p-state[P] : $PS2$) BC)

It suffices to consider s_{CLDSA} as base-state((p-state[p] : $ps1$) bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl) in S_{CLDSA} to which the transition rule can be applied.

Because of $\mathbf{r}(s_{UDS}, s_{CLDSA})$, s_{UDS} is $\text{delM}((\text{p-state}[p] : ps1) bc) = (\text{p-state}[p] : ps1) \text{delM}(bc)$ from definition of function delM . s_{CLDSA} goes to s'_{CLDSA} by state transition t , then s'_{CLDSA} is base-state((p-state[p] : $ps2$) bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl).

Let us choose the state transition t' constructed from the transition rule that describes Change of Process State in M_{UDS} .

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

Let s'_{UDS} be $(\text{p-state}[p] : ps2) \text{delM}(bc)$. Then s_{UDS} can go to s'_{UDS} by state transition t' .

We have:

$$s'_{UDS} = (\text{p-state}[p] : ps2) \text{delM}(bc) \quad (*) \text{ and,} \\ \text{delM}(\text{b-state}(s'_{CLDSA})) = \text{delM}((\text{p-state}[p] : ps2) bc) = (\text{p-state}[p] : ps2) \text{delM}(bc) \quad (**),$$

From $(*)$ and $(**)$, $s'_{UDS} = \text{delM}(\text{b-state}(s'_{CLDSA}))$.

Therefore, $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and $s_{UDS} \rightsquigarrow_{M_{UDS}} s'_{UDS}$ by state transition t' .

2. **Sending of Message.** Let us consider the case in which t is constructed from the transition rule that describes Sending of Message in M_{CLDSA} .

base-state((p-state[P] : $PS1$) (c-state[P, Q, N] : MMS))

$BC)$

\Rightarrow

base-state((p-state[P] : $PS2$) (c-state[P, Q, N] : enq(MMS, M))

$BC)$

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : $ps1$) (c-state[p, q, n] : mms) bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl) in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}} = \text{delM}((\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : mms) bc) = (\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : \text{delChan}(mms)) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : $ps2$) (c-state[p, q, n] : enq(mms, m)) bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl).

Let us choose the state transition t' is constructed from the transition rule that describes Sending of Message in $M_{\mathcal{UDS}}$.

(p-state[P] : $PS1$) (c-state[P, Q, N] : CS)

\Rightarrow

(p-state[P] : $PS2$) (c-state[P, Q, N] : enq(CS, M))

Let $s'_{\mathcal{UDS}}$ be (p-state[p] : $ps2$) (c-state[p, q, n] : enq($\text{delChan}(mms), m$)) $\text{delM}(bc)$. Then $s_{\mathcal{UDS}}$ can goes to $s'_{\mathcal{UDS}}$ by state transition t' .

We have:

$s'_{\mathcal{UDS}} = (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : \text{enq}(\text{delChan}(mms), m)) \text{delM}(bc) (*)$
and, because m is a message, $\text{delChan}(\text{enq}(mms, m)) = \text{enq}(\text{delChan}(mms), m)$.

Then, $\text{delM}(\text{base-sate}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps2)$
(c-state[p, q, n] : $\text{delChan}(\text{enq}(mms, m))$) $\text{delM}(bc) = (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n]$
: $\text{enq}(\text{delChan}(mms), m)) \text{delM}(bc) (**).$

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}}))$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{UDS}} \rightsquigarrow_{M_{\mathcal{UDS}}} s'_{\mathcal{UDS}}$ by state transition t' .

3. **Receipt of Message:** can be separated into four subcases

- (a) **The process has not yet started the \mathcal{CLDSA} .** Let us consider the case in which t is constructed from the transition rule that describes 1st case of Receipt of Message in $M_{\mathcal{CLDSA}}$.

base-state((p-state[P] : $PS1$) (c-state[Q, P, N] : $M \mid MMS$) BC)

control((prog[P] : notYet) CC)

\Rightarrow

base-state((p-state[P] : $PS2$) (c-state[Q, P, N] : MMS) BC)
control((prog[P] : notYet) CC)

It suffices to consider $s_{\mathcal{CLDS}_A}$ as base-state((p-state[p] : $ps1$) (c-state[q, p, n] : $m \mid mms$) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : notYet) cc) in $S_{\mathcal{CLDS}_A}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDS}_A})$, $s_{\mathcal{UDS}}$ is $\text{delM}(\text{b-state}(s_{\mathcal{CLDS}_A})) = \text{delM}((\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid mms) bc) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(m \mid mms)) \text{delM}(bc) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid \text{delChan}(mms)) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDS}_A}$ goes to $s'_{\mathcal{CLDS}_A}$ by state transition t , then $s'_{\mathcal{CLDS}_A}$ is base-state((p-state[p] : $ps2$) (c-state[q, p, n] : mms) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : notYet) cc).

Let us choose the state transition t' is constructed from the transition rule that describes Receipt of Message in $M_{\mathcal{UDS}}$.

(p-state[P] : $PS1$) (c-state[Q, P, N] : $M \mid CS$)

\Rightarrow

(p-state[P] : $PS2$) (c-state[Q, P, N] : CS)

Let $s'_{\mathcal{UDS}}$ is (p-state[p] : $ps2$) (c-state[q, p, n] : $\text{delChan}(mms)$) $\text{delM}(bc)$. Then $s_{\mathcal{UDS}}$ can goes to $s'_{\mathcal{UDS}}$ by state transition t' .

We have:

$s'_{\mathcal{UDS}}$ is (p-state[p] : $ps2$) (c-state[q, p, n] : $\text{delChan}(mms)$) $\text{delM}(bc)$ (*) and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) = \text{delM}((\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : mms) bc) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ from definition of function $\text{delM}(**)$,

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A}))$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDS}_A})$ and $s_{\mathcal{UDS}} \rightsquigarrow_{M_{\mathcal{UDS}}} s'_{\mathcal{UDS}}$ by state transition t' .

- (b) **The process has completed \mathcal{CLDS}_A .** Let us consider the case in which t is constructed from the transition rule that describes 2st case of Receipt of Message in $M_{\mathcal{CLDS}_A}$.

base-state((p-state[P] : $PS1$) (c-state[Q, P, N] : $M \mid MMS$) BC)

control((prog[P] : completed) CC)

\Rightarrow

base-state((p-state[P] : $PS2$) (c-state[Q, P, N] : MMS) BC)

control((prog[P] : completed) CC)

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps1) (c-state[q, p, n] : m | mms) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : completed) cc) in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : \text{ps1}) (\text{c-state}[q, p, n] : m \mid mms) bc)) = (\text{p-state}[p] : \text{ps1}) (\text{c-state}[q, p, n] : \text{delChan}(m \mid mms)) \text{delM}(bc) = (\text{p-state}[p] : \text{ps1}) (\text{c-state}[q, p, n] : m \mid \text{delChan}(mms)) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : ps2) (c-state[q, p, n] : mms) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : (prog[p] : completed)) cc).

Let us choose the state transition t' is constructed from the transition rule that describes Receipt of Message in $M_{\mathcal{UDS}}$.

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

\Rightarrow

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

Let $s'_{\mathcal{UDS}}$ is (p-state[p] : ps2) (c-state[q, p, n] : delChan(mms)) delM(bc). Then $s_{\mathcal{UDS}}$ can goes to $s'_{\mathcal{UDS}}$ by state transition t' .

We have:

$s'_{\mathcal{UDS}}$ is (p-state[p] : ps2) (c-state[q, p, n] : delChan(mms)) delM(bc) (*) and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : \text{ps2}) (\text{c-state}[q, p, n] : mms) bc)) = (\text{p-state}[p] : \text{ps1}) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ from definition of function $\text{delM}(**)$,

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}}))$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{UDS}} \rightsquigarrow_{M_{\mathcal{UDS}}} s'_{\mathcal{UDS}}$ by state transition t' .

- (c) **The process has started \mathcal{CLDSA} , not yet completed it, and has not yet received a marker from the incoming channel.** Let us consider the case in which t is constructed from the transition rule that describes 3 st case of Receipt of Message in $M_{\mathcal{CLDSA}}$.

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)

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : $ps1$) (c-state[q, p, n] : $m \mid mms$) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : started) (done[q, p, n] : false) cc). in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid mms) bc)) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(m \mid mms)) \text{delM}(bc) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid \text{delChan}(mms)) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : $ps2$) (c-state[q, p, n] : mms) bc) start-state(sc) snapshot((c-state[q, p, n] : enq(ms, m)) ssc) finish-state(fc) control((prog[p] : started) (done[q, p, n] : false) cc).

Let us choose the state transition t' is constructed from the transition rule that describes Receipt of Message in $M_{\mathcal{UDS}}$.

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

Let $s'_{\mathcal{UDS}}$ is $(\text{p-state}[P] : ps2) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$. Then $s_{\mathcal{UDS}}$ can goes to $s'_{\mathcal{UDS}}$ by state transition t' .

We have:

$s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ (*) and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : mms) bc)) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ from definition of function $\text{delM}(**)$,

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}}))$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{UDS}} \rightsquigarrow_{M_{\mathcal{UDS}}} s'_{\mathcal{UDS}}$ by state transition t' .

- (d) **The process has started \mathcal{CLDSA} , not yet completed it, and has already received a marker from the incoming channel.** Let us consider the case in which t is constructed from the transition rule that describes 4 st case of Receipt of Message in $M_{\mathcal{CLDSA}}$.

base-state((p-state[P] : $PS1$) (c-state[Q, P, N] : $M \mid MMS$) BC)

$\text{control}((\text{prog}[P] : \text{started}) (\text{done}[Q, P, N] : \text{true}) CC)$
 \Rightarrow
 $\text{base-state}((\text{p-state}[P] : PS2) (\text{c-state}[Q, P, N] : MMS) BC)$
 $\text{control}((\text{prog}[P] : \text{started}) (\text{done}[Q, P, N] : \text{true}) CC)$

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid mms) bc)$ $\text{start-state}(sc)$ $\text{snapshot}(ssc)$ $\text{finish-state}(fc)$ $\text{control}((\text{prog}[p] : \text{started}) (\text{done}[q, p, n] : \text{true}) cc)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{UDS}, s_{\mathcal{CLDSA}})$, s_{UDS} is $\text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid mms) bc)) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(m \mid mms)) \text{delM}(bc) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : m \mid \text{delChan}(mms)) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : mms) bc)$ $\text{start-state}(sc)$ $\text{snapshot}(ssc)$ $\text{finish-state}(fc)$ $\text{control}((\text{prog}[p] : \text{started}) (\text{done}[q, p, n] : \text{true}) cc)$.

Let us choose the state transition t' is constructed from the transition rule that describes Receipt of Message in M_{UDS} .

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

Let s'_{UDS} is $(\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$. Then s_{UDS} can goes to s'_{UDS} by state transition t' .

We have:

s'_{UDS} is $(\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ (*) and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : mms) bc)) = (\text{p-state}[p] : ps1) (\text{c-state}[q, p, n] : \text{delChan}(mms)) \text{delM}(bc)$ from definition of function $\text{delM}(**)$,

From (*) and (**), $s'_{UDS} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}}))$.

Therefore, $\mathbf{r}(s'_{UDS}, s'_{\mathcal{CLDSA}})$ and $s_{UDS} \rightsquigarrow_{M_{UDS}} s'_{UDS}$ by state transition t' .

The \mathcal{CLDSA} part: state transition t is constructed from the transition rule in the \mathcal{CLDSA} part

Since, the \mathcal{CLDSA} part is independent from the UDS concerned, can be constructed regardless of any UDS s, and does not change the base-state meta configuration component of a state of $CL(M_{UDS})$. Our proof show that we can choose s'_{UDS} the same as s_{UDS} then s_{UDS} goes to s'_{UDS} by zero step and $\mathbf{r}(s'_{UDS}, s'_{\mathcal{CLDSA}})$. This is shown in Fig. 7.3. The following prove will consider transition rules in \mathcal{CLDSA} part.

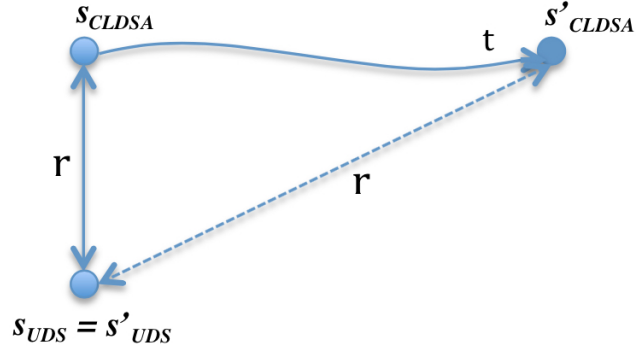


Figure 7.3: s'_{UDS} is the same as s_{UDS} when state transition t in the \mathcal{CLDSA} part.

- **Record of Process State** is split into two subcases:

1. **The process globally initiates \mathcal{CLDSA} .** This case is further split into three subcases:
 - (a) The \mathcal{UDS} only consists of the process.

Let us consider the case in which t is constructed from the transition rule that describes this case of Record of Process State in $M_{\mathcal{CLDSA}}$.

```

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

```

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps))$
 $\text{start-state}(\text{empConfig})$ $\text{snapshot}(\text{empConfig})$ $\text{finish-state}(\text{empConfig})$
 $\text{control}((\text{prog}[p] : \text{notYet}) (\text{cnt} : 1) (\#ms[p] : 0) cc)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{UDS}, s_{\mathcal{CLDSA}})$, s_{UDS} is $\text{delM}(\text{p-state}[p] : ps) = (\text{p-state}[p] : ps)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps)) \text{ start-state}((\text{p-state}[p] : ps)) \text{ snapshot}((\text{p-state}[p] : ps)) \text{ finish-state}((\text{p-state}[p] : ps)) \text{ control}((\text{prog}[p] : \text{completed}) (\text{cnt} : 0) (\#ms[p] : 0) cc)$.

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps)$ (*).

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}(\text{p-state}[p] : ps) = (\text{p-state}[p] : ps)$ (**).

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

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

Let us consider the case in which t is constructed from the transition rule that describes this case of Record of Process State in $M_{\mathcal{CLDSA}}$.

```

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)
⇒
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)
if NzN > 1

```

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps) bc)$ $\text{start-state}(\text{empConfig}) \text{ snapshot}(\text{empConfig}) \text{ finish-state}(fc) \text{ control}((\text{prog}[p] : \text{notYet}) (\text{cnt} : nzn') (\#ms[p] : 0) cc)$ such that $nzn' > 1$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{ delM}(bc)$ from definition of function delM

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps) \text{ bcast}(bc, p, \text{marker})) \text{ start-state}((\text{p-state}[p] : ps)) \text{ snapshot}((\text{p-state}[p] : ps)) \text{ finish-state}(fc) \text{ control}((\text{prog}[p] : \text{completed}) (\text{cnt} : 0) (\#ms[p] : 0) cc)$.

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) \text{ delM}(bc)$ (*).

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) = \text{delM}((\text{p-state}[p] : ps) \mathbf{bcast}(bc, p, \text{marker}))$
 $= (\text{p-state}[p] : ps) \text{delM}(\mathbf{bcast}(bc, p, \text{marker}))$ from definition of function delM .

Function $\mathbf{bcast}(bc, p, \text{marker})$ to puts markers in all the outgoing channels from a process p , then $\text{delM}(\mathbf{bcast}(bc, p, \text{marker})) = \text{delM}(bc)$.

Then $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) = (\text{p-state}[p] : ps) \text{delM}(bc) \text{ (**)}$.

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) = (\text{p-state}[p] : ps) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDS}_A})$

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

Let us consider the case in which t is constructed from the transition rule that describes this case of Record of Process State in $M_{\mathcal{CLDS}_A}$.

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)

It suffices to consider $s_{\mathcal{CLDS}_A}$ as base-state((p-state[p] : ps) bc) start-state(empConfig) snapshot(empConfig) finish-state(fc) control((prog[p] : notYet) (#ms[p] : nzn) cc) in $S_{\mathcal{CLDS}_A}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDS}_A})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function delM .

Since $s_{\mathcal{CLDS}_A}$ goes to $s'_{\mathcal{CLDS}_A}$ by state transition t , $s'_{\mathcal{CLDS}_A}$ is 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)) finish-state(fc) control((prog[p] : started) (cnt : 0) (#ms[p] : 0) cc).

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) \text{delM}(bc)$ (*).

Function $\mathbf{bcast}(bc, p, \text{marker})$ to put markers in all the outgoing channels of the process p , $\text{delM}(\mathbf{bcast}(bc, p, \text{marker})) = \text{delM}(bc)$.

Then we have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc)$ (**).

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

2. The process does not globally initiates \mathcal{CLDSA} . This case is further split into three subcases:

- (a) The process does not have any incoming channels, and there are no processes except for the process that has not completed \mathcal{CLDSA} .

Let us consider the case in which t is constructed from the transition rule that describes this case of Record of Process State in $M_{\mathcal{CLDSA}}$.

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

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps) bc) \text{start-state}(sc) \text{snapshot}(ssc) \text{finish-state}(\text{empConfig}) \text{control}((\text{prog}[p] : \text{notYet}) (\text{cnt} : 1) (\#ms[p] : 0) cc)$, such that $(sc \neq \text{empConfig})$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function delM .

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps) bc) \text{start-state}(sc) \text{snapshot}((\text{p-state}[p] : ps) ssc) \text{finish-state}((\text{p-state}[p] : ps) fc) \text{control}((\text{prog}[p] : \text{completed}) (\text{cnt} : 0) (\#ms[p] : 0) cc)$.

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps)$

$\text{delM}(bc) (*)$.

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function $\text{delM}(**)$.

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

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

Let us consider the case in which t is constructed from the transition rule that describes 5 st case of Record of Process State in $M_{\mathcal{CLDSA}}$.

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(SSC)
control((prog[P] : completed) (cnt : sd(NzN ,1)) (#ms[P] : 0) CC)
if ($SC \neq \text{empBConfig} \wedge (NzN > 1)$)

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : notYet) (cnt : nzn) (#ms[p] : 0) cc), such that ($nzn > 1$) and ($sc \neq \text{empConfig}$) in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function delM .

$s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , then $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : ps) **bcast**(bc , p , marker)) start-state(sc) snapshot((p-state[p] : ps) ssc) finish-state(fc) control((prog[p] : completed) (cnt : sd(nzn ,1)) cc) .

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) \text{delM}(bc)(*)$.

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) \mathbf{bcast}(bc, p, marker)) = (\text{p-state}[p] : ps) \text{delM}(\mathbf{bcast}(bc, p, marker))$ from definition of function delMand and $\text{delM}(\mathbf{bcast}(bc, p, marker))) = \text{delM}(bc)$.

Then $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc) \text{ (**)}.$

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc).$

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

(c) The process has some incoming channels.

Let us consider the case in which t is constructed from the transition rule that describes 6 st case of Record of Process State in $M_{\mathcal{CLDSA}}$.

```

base-state((p-state[P] : PS) BC)
start-state(SC)
snapshot(SSC)
control((prog[P] : notYet) (#ms[P] : NzN') CC)
⇒
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 ≠ empBConfig)

```

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps) bc) \text{start-state}(sc) \text{snapshot}(ssc) \text{finish-state}(fc) \text{control}((\text{prog}[p] : \text{notYet}) (\#ms[p] : nzn') cc)$ such that $sc \neq \text{empConfig}$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) bc) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function delM .

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps) \mathbf{bcast}(bc, p, \text{marker})) \text{start-state}(sc) \text{snapshot}((\text{p-state}[p] : ps) \text{inchans}(bc, p) ssc) \text{finish-state}(fc) \text{control}((\text{prog}[p] : \text{started}) (\#ms[p] : nzn') cc)$.

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) \text{delM}(bc) \text{ (*)}.$

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) \mathbf{bcast}(bc, p, \text{marker})) = (\text{p-state}[p] : ps) \text{delM}(bc)$ from definition of function delM (**).

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) \text{delM}(bc).$

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

- **Receipt of Marker** is split into two subcases:

1. The process has not yet started \mathcal{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 \mathcal{CLDSA} except for the process.

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in $M_{\mathcal{CLDSA}}$.

$$\begin{aligned}
& \text{base-state}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : \text{marker} \mid MMS) BC) \\
& \text{snapshot}(SSC) \\
& \text{finish-state}(\text{empBConfig}) \\
& \text{control}((\text{prog}[P] : \text{notYet}) (\text{cnt} : 1) (\#ms[P] : 1) (\text{done}[Q, P, N] : \text{false}) CC) \\
& \Rightarrow \\
& \text{base-state}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : MMS) BC) \\
& \text{snapshot}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : \text{empChan}) SSC) \\
& \text{finish-state}((\text{p-state}[P] : PS) (\text{c-state}[Q, P, N] : MMS) BC) \\
& \text{control}((\text{prog}[P] : \text{completed}) (\text{cnt} : 0) (\#ms[P] : 0) (\text{done}[Q, P, N] : \text{true}) CC)
\end{aligned}$$

It suffices to consider $s_{\mathcal{CLDSA}}$ as $\text{base-state}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{marker} \mid ms) bc)$ $\text{start-state}(sc)$ $\text{snapshot}(ssc)$ $\text{finish-state}(\text{empConfig})$ $\text{control}((\text{prog}[p] : \text{notYet}) (\#ms[p] : 1) (\text{cnt} : 1) (\text{done}[q, p, n] : \text{false}) cc)$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{marker} \mid ms) bc)$
 $= (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(\text{marker} \mid ms)) \text{delM}(bc)$
 $= (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM .

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is $\text{base-state}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc)$ $\text{start-state}(sc)$ $\text{snapshot}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{empChan}) ssc)$ $\text{finish-state}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc)$ $\text{control}((\text{prog}[p] : \text{completed}) (\#ms[p] : 0) (\text{cnt} : 0) (\text{done}[q, p, n] : \text{true}) cc)$

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)(*)$.

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc)$
 $= (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM .

From (*) and (**), $s'_{UDS} = \text{delM}(\text{b-state}(s'_{CLDSA})) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$

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

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in M_{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 $NzN > 1$

It suffices to consider s_{CLDSA} as base-state((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc) start-state(sc) snapshot(ssc) finish-state(empConfig)
 control((prog[p] : notYet) (#ms[p] : 1) (cnt : nzn) (done[q, p, n] : false) cc),
 where $nzn > 1$, in S_{CLDSA} to which the transition rule can be applied.

Because of $\mathbf{r}(s_{UDS}, s_{CLDSA})$, s_{UDS} is $\text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{marker} | ms) bc)$
 $= (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM .

Since s_{CLDSA} goes to s'_{CLDSA} by state transition t , s'_{CLDSA} is base-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) start-state(sc) snapshot((p-state[p] : ps) (c-state[q, p, n] : empChan) ssc) finish-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) control((prog[p] : completed) (#ms[p] : 0) (cnt : sd($nzn, 1$)) (done[q, p, n] : true) cc)

Let we choose s'_{UDS} is the same s_{UDS} , means that s'_{UDS} is $(\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ (*).

We have $\text{delM}(\text{b-state}(s'_{CLDSA})) = \text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM .

From (*) and (**), $s'_{UDS} = \text{delM}(\text{b-state}(s'_{CLDSA})) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$.

$\text{state}[q, p, n] : \text{delChan}(ms)) \text{ delM}(bc).$

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

- (c) The process has more than one incoming channel.

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in $M_{\mathcal{CLDSA}}$.

base-state((p-state[P] : PS) (c-state[Q, P, N] : marker | MMS) BC)
 snapshot(SSC)
 control((prog[P] : notYet) (cnt : NzN) (#ms[P] : 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 $NzN' > 1$

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc) start-state(sc) snapshot(ssc) finish-state(fc) control((prog[p] : notYet) (#ms[p] : nzn') (cnt : nzn) (done[q, p, n] : false) cc), such that $nzn' > 1$, in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is delM((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc)
 $=$ (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of function delM.

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) start-state(sc) snapshot((p-state[p] : ps) (c-state[q, p, n] : empChan) inchans(bc, p) ssc) finish-state(fc) control((prog[p] : started) (#ms[p] : sd($nzn', 1$)) (cnt : nzn) (done[q, p, n] : true) cc)

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is (*).

We have delM(b-state($s'_{\mathcal{CLDSA}}$)) = (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc)

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc)$
 $=$ (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of function delM.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

2. The process has already started \mathcal{CLDSA} . This case is further split into three sub-cases:

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

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in $M_{\mathcal{CLDSA}}$.

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)

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc) start-state(sc) snapshot(ssc) finish-state(empConfig)
 control((prog[p] : started) (#ms[p] : 1) (cnt : 1) (done[q, p, n] : false) cc) in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is delM((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc)
 $=$ (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of function delM.

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) start-state(sc) snapshot((p-state[p] : ps) (c-state[q, p, n] : empChan) ssc) finish-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) control((prog[p] : completed) (#ms[p] : 0) (cnt : 0) (done[q, p, n] : true) cc) (*).

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc)

We have delM(b-state($s'_{\mathcal{CLDSA}}$)) = delM((p-state[p] : ps) (c-state[q, p, n] : ms) bc)
 $=$ (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of function delM(**).

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

- (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 \mathcal{CLDSA} .

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in $M_{\mathcal{CLDSA}}$.

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 $NzN > 1$

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps) (c-state[q, p, n] : marker | ms) bc) start-state(sc) snapshot(ssc) finish-state(fc)
control((prog[p] : started) (#ms[p] : 1) (cnt : nzn) (done[q, p, n] : false) cc)
such that $nzn > 1$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is $\text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{marker} | ms) bc)$
 $= (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM .

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : ps) (c-state[q, p, n] : ms) bc) start-state(sc) snapshot((p-state[p] : ps) (c-state[q, p, n] : empChan) ssc) finish-state(fc) control((prog[p] : completed) (#ms[p] : 0) (cnt : 0) (done[q, p, n] : true) cc)

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is $(\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ (*).

We have $\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : ms) bc) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$ from definition of function delM

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps) (\text{c-state}[q, p, n] : \text{delChan}(ms)) \text{delM}(bc)$.

Therefore, $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$

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

Let us consider the case in which t is constructed from the transition rule that describes this case of Receipt of Marker in $M_{\mathcal{CLDSA}}$.

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 $NzN' > 1$

It suffices to consider $s_{\mathcal{CLDSA}}$ as base-state((p-state[p] : ps) (c-state[q, p, n] :
marker | ms) bc) start-state(sc) snapshot(ssc) finish-state(empConfig)
control((prog[p] : started) (#ms[p] : nzn') (cnt : nzn) (done[q, p, n] : false)
 cc) such that $nzn' > 1$ in $S_{\mathcal{CLDSA}}$ to which the transition rule can be applied.

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}}$ is delM((p-state[p] : ps) (c-state[q, p, n] :
marker | ms) bc)
= (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of
function delM.

Since $s_{\mathcal{CLDSA}}$ goes to $s'_{\mathcal{CLDSA}}$ by state transition t , $s'_{\mathcal{CLDSA}}$ is base-state((p-
state[p] : ps) (c-state[q, p, n] : ms) bc) start-state(sc) snapshot((p-state[p] : ps)
(c-state[q, p, n] : empChan) ssc) finish-state((p-state[p] : ps) (c-state[q, p, n] :
 ms) bc) control((prog[p] : started) (#ms[p] : sd($nzn', 1$)) (cnt : nzn) (done[q, p, n]
: true) cc)

Let we choose $s'_{\mathcal{UDS}}$ is the same $s_{\mathcal{UDS}}$, means that $s'_{\mathcal{UDS}}$ is (p-state[p] : ps)
(c-state[q, p, n] : delChan(ms)) delM(bc) (*).

We have delM(b-state($s'_{\mathcal{CLDSA}}$)) = delM((p-state[p] : ps) (c-state[q, p, n] : ms)
 bc)
= (p-state[p] : ps) (c-state[q, p, n] : delChan(ms)) delM(bc) from definition of
function delM(**).

From (*) and (**), $s'_{\mathcal{UDS}}$ = delM(b-state($s'_{\mathcal{CLDSA}}$)) = p-state[p] : ps) (c-state[q, p, n]
: delChan(ms)) delM(bc) .

Therefore, $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$

We have considered all transition rules in M_{CLDSA} and prove that for each $s_{CLDSA}, s'_{CLDSA} \in S_{CLDSA}$ and $s_{UDS} \in S_{UDS}$ such that $\mathbf{r}(s_{UDS}, s_{CLDSA})$ and $s_{CLDSA} \rightsquigarrow_{M_{CLDSA}} s'_{CLDSA}$ by one state transition of the M_{CLDSA} , there exists s'_{UDS} such that $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and $s_{UDS} \rightsquigarrow_{M_{UDS}}^* s'_{UDS}$ by zero or more. Therefore, this condition is hold.

From what have been proved above, we can see that binary relation \mathbf{r} satisfies that for all state s in I_{CLDSA} , exist s' in I_{UDS} such that s is simulated by s' and for all s_1, s_2 in S_{CLDSA} and s_1 moves to s_2 by state transition t , s'_1 in S_{UDS} such that s_1 is simulated by s'_1 , then there exist s'_2 such that s_2 moves to s'_2 by zero or more state transitions and s_2 is simulated by s'_2 . This mean that for any state s in S_{CLDSA} which can be moved from any state in I_{CLDSA} , we can find a state s' in S_{UDS} , such that s is simulated by s' . Therefore, binary relation \mathbf{r} is simulation from M_{CLDSA} to M_{UDS} . QED

7.2.2 Simulation from M_{UDS} to M_{CLDSA}

To prove that \mathbf{r} is a simulation from M_{UDS} to M_{CLDSA} , we shall prove that \mathbf{r} satisfies the following two conditions. Fig. 7.2 shows the diagrams corresponding to the two conditions.

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

For each $s_{UDS} \in I_{UDS}$, from the definition of CL , we can find:

$s_{CLDSA} = \text{base-state}(s_{UDS}) \text{ start-state}(\text{empConfig}) \text{ snapshot}(\text{empConfig})$
 $\text{finsh-state}(\text{empConfig}) \text{ control}(ctl) \text{ where } ctl = \text{InitCtlConfig}(s_{UDS}).$

Because of $\text{b-state}(s_{CLDSA}) = s_{UDS}$, $\text{delM}(\text{b-state}(s_{CLDSA})) = \text{delM}(s_{UDS}) = s_{UDS}$ means that $s_{UDS} = \text{delM}(\text{b-state}(s_{CLDSA}))$ then $\mathbf{r}(s_{UDS}, s_{CLDSA})$.

Therefore, for each $s_{UDS} \in I_{UDS}$ there exists $s_{CLDSA} \in I_{CLDSA}$, such that $\mathbf{r}(s_{UDS}, s_{CLDSA})$.

2. For each $s_{UDS}, s'_{UDS} \in S_{UDS}$ and $s_{CLDSA} \in S_{CLDSA}$ such that $\mathbf{r}(s_{UDS}, s_{CLDSA})$ and $s_{UDS} \rightsquigarrow_{M_{UDS}} s'_{UDS}$ by one state transition of the M_{UDS} , there exists s'_{CLDSA} such that $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and $s_{CLDSA} \rightsquigarrow_{M_{CLDSA}}^* s'_{CLDSA}$ by zero or more state transitions of the M_{CLDSA} .

Prove:

Assume that $s_{UDS} \rightsquigarrow_{M_{UDS}} s'_{UDS}$ by state transition t ,

To prove that \mathbf{r} is simulation from M_{UDS} to M_{CLDSA} , it suffices to take into account the three transition rules. It means that we only need to consider state transitions, which

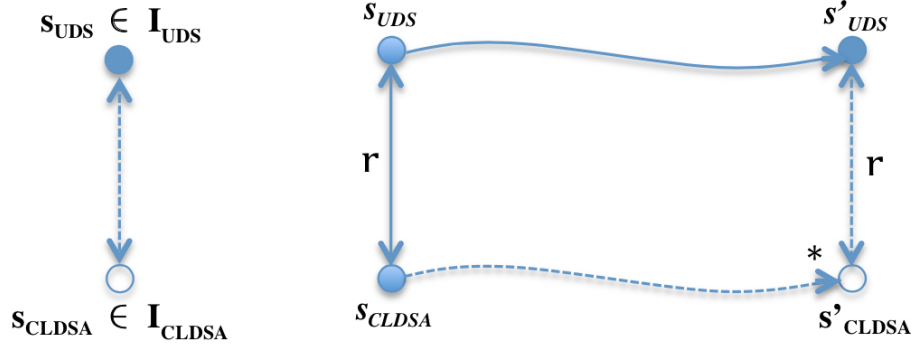


Figure 7.4: The binary relation \mathbf{r} is a simulation from M_{UDS} to M_{CLDSA} .

are defined by the three transition rules.

We also know that for any transition rule in M_{UDS} , there exists a transition rule in the UDS part in M_{CLDSA} , which can be constructed from the transition rules of the UDS and changes the base-state meta configuration component of a state of M_{CLDSA} .

More specifically in this proof, for any state transition t in M_{UDS} that moves state s_{UDS} to state s'_{UDS} in M_{UDS} , there exists s'_{CLDSA} such that $\mathbf{r}(s'_{UDS}, s'_{CLDSA})$ and we can find state transition t' in M_{CLDSA} such that t' can move state s_{CLDSA} to state s'_{CLDSA} in M_{CLDSA} . This will be proven by the following. The following proof will consider all the three transition rules in M_{UDS} . In the following proof: $p, q \in \text{Pid}$, $ps1, ps2 \in \text{PState}$, $n \in \text{Nat}$, $m \in \text{Msg}$, $cs \in \text{MsgSeq}$, $cs' \in \text{MMsgSeq}$, $cf \in \text{Config}$, $bc, bc', sc, ssc, fc \in \text{BConfig}$ and $ctl, cc \in \text{CtlConfig}$ are fresh constants of those sorts.

1. Change of Process State.

Let us consider the case in which t is constructed from the transition rule that describes Change of Process State in M_{UDS} .

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

It suffices to consider s_{UDS} as to which the transition rule can be applied:

$$- s_{UDS} = ((\text{p-state}[p] : ps1) \text{ cf})$$

s_{CLDSA} is in form $\text{base-state}(bc) \text{ start-state}(sc) \text{ snapshot}(ssc) \text{ finish-state}(fc) \text{ control}(ctl)$

Because of $\mathbf{r}(s_{UDS}, s_{CLDSA})$,

$$s_{UDS} = \text{delM}(\text{b-state}(s_{CLDSA})) = \text{delM}(bc), \text{delM}(bc) = (\text{p-state}[p] : ps1) \text{ cf}.$$

Then $bc = ((\text{p-state}[p] : ps1) \text{ bc}')$, where $\text{delM}(bc') = \text{cf}$.

$s_{\mathcal{CLDSA}}$ is base-state((p-state[p] : $ps1$) bc') start-state(sc) snapshot(ssc) finish-state(fc) control(ctl), where $\text{delM}(bc') = cf$.

Since $s_{\mathcal{UDS}}$ goes to $s'_{\mathcal{UDS}}$ by state transition t , $s'_{\mathcal{UDS}} = ((\text{p-state}[p] : ps2) cf)$

Let us choose the state transition t' is constructed from the transition rule that describes Change of Process State in $M_{\mathcal{CLDSA}}$.

base-state((p-state[P] : $PS1$) BC)

\Rightarrow

base-state((p-state[P] : $PS2$) BC)

Let us choose $s'_{\mathcal{CLDSA}}$ is base-state((p-state[p] : $ps2$) bc') start-state(sc) snapshot(ssc) finish-state(fc) control(ctl).

We have $s'_{\mathcal{UDS}} = ((\text{p-state}[p] : ps2) cf) (*)$ and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = \text{delM}((\text{p-state}[p] : ps2) bc') = (\text{p-state}[p] : ps2) \text{delM}(bc')$
 $= (\text{p-state}[p] : ps2) cf$ from $\text{delM}(bc') = cf. (**)$

From (*) and (**), $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) = (\text{p-state}[p] : ps2) cf$.

Therefore $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{CLDSA}} \rightsquigarrow_{M_{\mathcal{CLDSA}}} s'_{\mathcal{CLDSA}}$ by state transition t' .

2. Sending of Message.

Let us consider the case in which t is constructed from the transition rule that describes Sending of Message in $M_{\mathcal{UDS}}$.

(p-state[P] : $PS1$) (c-state[P, Q, N] : CS)

\Rightarrow

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

It suffices to consider $s_{\mathcal{UDS}}$ as to which the transition rule can be applied is $s_{\mathcal{UDS}}$ is (p-state[p] : $ps1$) (c-state[p, q, n] : cs) cf .

$s_{\mathcal{CLDSA}}$ is in form : base-state(bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl)

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$, $s_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}(bc)$.

So $\text{delM}(bc) = (\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : cs) cf$,

Then $bc = ((\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : cs') bc'), where $\text{delM}(bc') = cf$ and$

$\text{delChan}(cs') = cs$ from definition of function delM .

Therefore $s_{\mathcal{CLDS}_A}$ is base-state((p-state[p] : $ps1$) (c-state[p, q, n] : cs') bc') start-state(sc) snapshot(ssc) finish-state(fc) control(ctl), where $\text{delM}(bc') = cf$ and $\text{delChan}(cs') = cs$.

Because $s_{\mathcal{UDS}}$ goes to $s'_{\mathcal{UDS}}$ by state transition t , $s'_{\mathcal{UDS}} = (\text{p-state}[p]: ps2) (\text{c-state}[p, q, n]: \text{enq}(cs, m))$

Let us choose the state transition t' is constructed from the transition rule that describes Sending of Message in $M_{\mathcal{CLDS}_A}$.

base-state((p-state[P] : $PS1$) (c-state[P, Q, N] : MMS)

BC)

\Rightarrow

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

BC)

Let us choose $s'_{\mathcal{CLDS}_A}$ is base-state((p-state[p] : $ps1$) (c-state[p, q, n] : $\text{enq}(cs', m)$) bc')

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

We have $s'_{\mathcal{UDS}} = (\text{p-state}[p]: ps2) (\text{c-state}[p, q, n]: \text{enq}(cs, m)) bc (*)$ and,

$\text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) = \text{delM}((\text{p-state}[p] : ps2)$
 $(\text{c-state}[p, q, n] : \text{enq}(cs', m)) bc)$
 $= (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n]: \text{delChan}(\text{enq}(cs', m)) \text{delM}(bc')))$

Because m is message, $\text{delChan}(\text{enq}(cs', m)) = \text{enq}(\text{delChan}(cs'), m)$, then

$(\text{p-state}[p] : ps2) (\text{c-state}[p, q, n]: \text{delChan}(\text{enq}(cs', m)) \text{delM}(bc')))$
 $= (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n]: \text{enq}(\text{delChan}(cs'), m) \text{delM}(bc')))$
 $= (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n]: \text{enq}(cs, m)) cf)$
 $\{\text{delM}(bc') = cf \text{ and } \text{delChan}(cs') = cs\}.$

Then $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDS}_A})) (**).$

From (*) and (**), $r(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDS}_A})$ and $s_{\mathcal{CLDS}_A} \rightsquigarrow_{M_{\mathcal{CLDS}_A}} s'_{\mathcal{CLDS}_A}$ by state transition t' .

3. Receipt of Message.

Let us consider the case in which t is constructed from the transition rule that describes Receipt of Message in $M_{\mathcal{UDS}}$.

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

It suffices to consider an arbitrary state $s_{\mathcal{UDS}}$ in $S_{\mathcal{UDS}}$ to which the rule can be applied is $s_{\mathcal{UDS}} = (\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : m \mid cs) cf$.

$s_{\mathcal{CLDSA}}$ is in form: base-state(bc) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl).

Because of $\mathbf{r}(s_{\mathcal{UDS}}, s_{\mathcal{CLDSA}})$,

$s_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s_{\mathcal{CLDSA}})) = \text{delM}(bc)$ and $\text{delM}(bc) = (\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : m \mid cs) cf$.

Then $bc = (\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : m \mid cs') bc'$, where $\text{delM}(bc') = cf$ and $\text{delChan}(cs') = cs$.

Therefore, $s_{\mathcal{CLDSA}}$ is base-state($(\text{p-state}[p] : ps1) (\text{c-state}[p, q, n] : m \mid cs') bc'$) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl), where $\text{delM}(bc') = cf$ and $\text{delM}(cs') = cs$.

Since, $s_{\mathcal{UDS}}$ goes to $s'_{\mathcal{UDS}}$ by state transition t , $s'_{\mathcal{UDS}} = (\text{p-state}[p] : ps2) (\text{c-state}[q, p, n] : cs) cf$

Although, there are four transition rules and $s_{\mathcal{CLDSA}}$ can apply one of them. However, No loss of generality, we can choose the following transition rule to consider.

Let us choose the state transition t' is constructed from the transition rule that describes the 1st of Receipt of Message in $M_{\mathcal{CLDSA}}$.

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

Let us choose $s'_{\mathcal{CLDSA}}$ is base-state($(\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : cs') bc'$) start-state(sc) snapshot(ssc) finish-state(fc) control(ctl').

We have $s'_{\mathcal{UDS}} = ((\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : m \mid cs) cf (*)$ and,

$$\begin{aligned} \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) &= \text{delM}((\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : cs') bc') \\ &= (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : \text{delChan}(cs')) \text{delM}(bc') \\ &= (\text{p-state}[p] : ps2) (\text{c-state}[p, q, n] : cs) cf \{ \text{delM}(bc') = cf \text{ and } \text{delChan}(cs') = cs \}. \end{aligned}$$

Then $s'_{\mathcal{UDS}} = \text{delM}(\text{b-state}(s'_{\mathcal{CLDSA}})) (**)$.

From (*) and (**) $\mathbf{r}(s'_{\mathcal{UDS}}, s'_{\mathcal{CLDSA}})$ and $s_{\mathcal{CLDSA}} \rightsquigarrow_{M_{\mathcal{CLDSA}}} s'_{\mathcal{CLDSA}}$ by state transition t' .

From what have been proved above, we can see that binary relation \mathbf{r} satisfies that for all state s in $I_{\mathcal{UDS}}$, exist s' in $I_{\mathcal{CLDSA}}$ such that s is simulated by s' and for all s_1, s_2 in $S_{\mathcal{UDS}}$ and s_1 moves to s_2 by state transition t , s'_1 in $S_{\mathcal{CLDSA}}$ such that s_1 is simulated by s'_1 , then there exists s'_2 in $S_{\mathcal{CLDSA}}$ such that s'_1 can moves to s'_2 by zero or more state transitions and s_2 is simulated by s'_2 . This means that for any state s in $S_{\mathcal{UDS}}$, which can be moved from any state in $I_{\mathcal{UDS}}$, we can find a state s' in $S_{\mathcal{CLDSA}}$, such that s is simulated by s' . Therefore, binary relation \mathbf{r} is simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$. QED

We have proven that \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$ and it is also a simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$ to prove Theorem 2. QED

7.3 Chapter Summary

This chapter proposed a binary relation \mathbf{r} between two state machines $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ by which for each $s1 \in S_{\mathcal{UDS}}$ and each $s2 \in S_{\mathcal{CLDSA}}$, $\mathbf{r}(s1, s2)$ if and only if $s1$ is the same as the state obtained by deleting all markers from $s2$. To guarantee that \mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS} , it then proved that $M_{\mathcal{CLDSA}}$ simulates $M_{\mathcal{UDS}}$ and vice versa. Namely it proved Theorem 2 saying that \mathbf{r} is a bi-simulation relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$. \mathbf{r} is a bi-simulation relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ if and only if \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$ and it is also a simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$.

To prove that \mathbf{r} is a simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$, the chapter then proven that \mathbf{r} satisfies the two conditions as follows: for all state s in $I_{\mathcal{CLDSA}}$, exist s' in $I_{\mathcal{UDS}}$ such that s is simulated by s' and for all s_1, s_2 in $S_{\mathcal{CLDSA}}$ and s_1 moves to s_2 by state transition t , s'_1 in $S_{\mathcal{UDS}}$ such that s_1 is simulated by s'_1 , then there exist s'_2 such that s_2 moves to s'_2 by zero or more state transitions and s_2 is simulated by s'_2 . This mean that for any state s in $S_{\mathcal{CLDSA}}$ which can be moved from any state in $I_{\mathcal{CLDSA}}$, we can find a state s' in $S_{\mathcal{UDS}}$, such that s is simulated by s' .

To prove that \mathbf{r} is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$, it then proven that \mathbf{r} satisfies the two conditions as follows: for all state s in $I_{\mathcal{UDS}}$, exist s' in $I_{\mathcal{CLDSA}}$ such that s is simulated by s' and for all s_1, s_2 in $S_{\mathcal{UDS}}$ and s_1 moves to s_2 by state transition t , s'_1 in $S_{\mathcal{CLDSA}}$ such that s_1 is simulated by s'_1 , then there exists s'_2 in $S_{\mathcal{CLDSA}}$ such that s'_1 can

moves to s'_2 by zero or more state transitions and s_2 is simulated by s'_2 . This means that for any state s in $S_{\mathcal{UDS}}$, which can be moved from any state in $I_{\mathcal{UDS}}$, we can find a state s' in $S_{\mathcal{CLDSA}}$, such that s is simulated by s' . Therefore, binary relation \mathbf{r} is simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$. Since \mathbf{r} was proven that it is a simulation from $M_{\mathcal{CLDSA}}$ to $M_{\mathcal{UDS}}$ and also a simulation from $M_{\mathcal{UDS}}$ to $M_{\mathcal{CLDSA}}$, Theorem 2 was proven.

Theorem 2 was proven asserting that \mathcal{CLDSA} does not alter the behaviors of a \mathcal{UDS} .

Chapter 8

Future Work

8.1 The Future Work

In recent decades, key software systems on which human beings heavily rely on are in the form of distributed systems, which consist of multiple nodes (or processes) connected with networks (or channels). Such systems should be fault tolerant because they need to run for a long time, keeping on providing services to human beings, other systems, etc. To make distributed systems fault tolerant, it is necessary to use many non-trivial distributed algorithms, such as snapshot algorithms, checkpointing algorithms and self-stabilizing algorithms. One common characteristic of these algorithms is that distributed systems are superimposed by them and may be regarded and treated as data by them. Therefore, such distributed algorithms can be called meta-distributed algorithms (MDAs) (see Fig. 8.1).

Since meta-distributed algorithms play the core part in our software-centric society and will become more and more important in our future highly advanced software-centric society, it is necessary as well as demanding to verify that such algorithms enjoy their desired properties. However, model checking that meta-distributed algorithms enjoy their desired properties has not been fully investigated because of the following reasons. It is challenging to specify meta-distributed algorithms in an existing specification language, such as PROMELA because it is necessary to specify distributed systems as computational targets or data that are dealt with by meta-distributed algorithms while an existing specification language, such as PROMELA is designed to fit to specify distributed systems

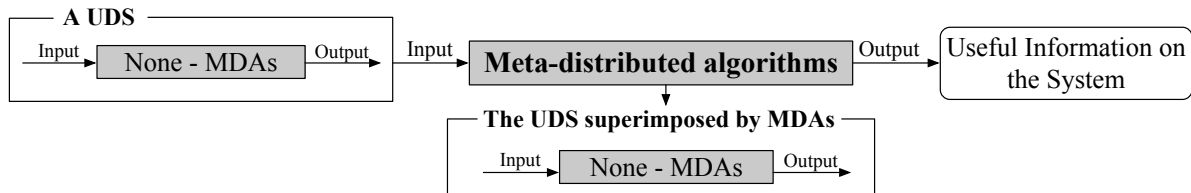


Figure 8.1: Meta-distributed algorithms.

but not computations of distributed systems (sequences of states that satisfy some conditions) as data, and to specify their desired properties in an existing temporal logic such as LTL because it is necessary to take into account computations of distributed systems. Therefore, it is also challenging to model check that meta-distributed algorithms enjoy their desired properties with an existing model checker, such as Spin [3]. It is therefore necessary to investigate how meta-distributed algorithms are formally analyzed.

Therefore, what we intend to do as the future work is to come up with how to specify meta-distributed algorithms, how to specify their desired properties, how to model check that meta-distributed algorithms enjoy their desired properties, and design and implement a model checker for meta-distributed algorithms.

It is necessary to deal with states of a distributed system, its computations and/or even a distributed system itself as data so as to specify an meta-distributed algorithm and its desired properties. From what we have conducted the current research on model checking of \mathcal{CLDSA} revisited, we have learned that Maude is one promising specification language for the purpose. Maude is a specification and programming language and system based on rewriting logic. Rewriting logic is a logic designed to formally deal with concurrent systems including distributed systems, and then Maude is suited to formally deal with distributed systems. Maude is equipped with many useful functionalities. Among them are model checking facilities and meta-programming facilities. The model checking facilities mainly consists of an LTL model checker and a search command that can be used as a reachability checker. Specifying a distributed system in Maude, its states are expressed as terms. Therefore, states of a distributed system can be treated as data. Although computations of a distributed system as they are cannot be directly treated as data in Maude, the Maude search command is able to analyze the computations, checking if a given state is reachable from another given state. The Maude meta-programming functionalities can treat Maude specifications as data, and then can deal with distributed systems as data. MDAs are typically used by superimposing distributed systems, and then the Maude meta-programming facilities make it possible to describe generic specifications of meta-distributed algorithms as meta-programs that take specifications of UDSs and generate specifications of the UDSs superimposed by an meta-distributed algorithm.

To achieve the research goal, the first to do is to write a meta-program in Maude that corresponds to the function CL . The meta-program takes a specification of a UDS in Maude and generates a specification of the UDS superimposed by \mathcal{CLDSA} in Maude. The next to do is to implement a model checker as a meta-program in Maude that can directly deal with the more faithful formalization of the DSR property. Then, some more concrete meta-distributed algorithms, such as Koo-Toueg Checkpointing Algorithm, will be taken and similar case studies will be repeated, coming up with a generic way to specify meta-distributed algorithms, their desired properties and model check that such algorithms enjoy such properties.

We will first specify a UDS as a program in Maude. The specification of a UDS corresponds to state machine M_{UDS} . The specification of a UDS corresponds to state machine M_{UDS} . The system is specified as the module UDS in Maude. In which, the sorts Pid for process identifiers, $PState$ for process states, Msg for messages, $MsgQueue$ for

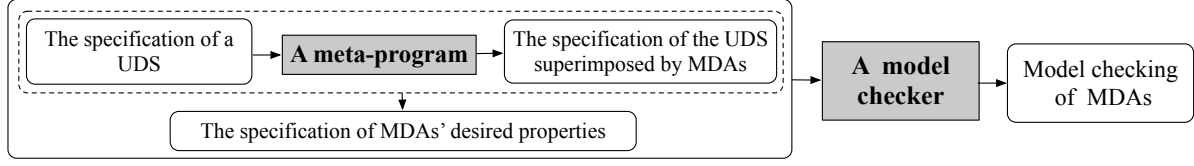


Figure 8.2: Meta-distributed algorithms.

queues of messages, OCom for observable components and Config for soups of observable components will be declared. Sub-sorts will also declared.

```

sorts Pid Msg PState .
sort MsgSeq .
sort OCom .
sort Config .

subsort OCom < Config .

```

Then the operator to constructor the sort are defined.

```

op empChan : -> MsgSeq [ctor] .
op _|_ : Msg MsgSeq -> MsgSeq [ctor] .

op p-state[_]:_ : Pid PState -> OCom [ctor] .
op c-state[_ , _ , _]:_ : Pid Pid Nat MsgSeq -> OCom [ctor] .

op empConfig : -> Config [ctor] .
op _ _ : Config Config -> Config [ctor assoc comm id: empConfig] .

```

Variables will be declared.

```

vars P P1 Q Q1 : Pid .
vars M1 M2 : Msg .
var N N1 : Nat .
var MS : MsgSeq .
var PS PS1 PS2 : PState .
var CS : MsgSeq .
vars OC OC1 : OCom .
vars CF CF1 : Config .

```

The operators and equations for the functions on the state of a *UDS*, such as chans, msg, enq, # and = are defined.

```

op chans : Config -> Config .
op msg : OCom -> MsgSeq .
op enq : MsgSeq Msg -> MsgSeq .
op # : Config Pid -> Nat .

eq chans(empConfig) = empConfig .
eq chans((p-state[P]: PS) CF) = chans(CF) .
eq chans((c-state[P, Q, N]: CS) CF) = (c-state[P, Q, N] : CS) chans(CF) .

eq msg(c-state[P, Q, N] : CS) = CS .
eq msg(p-state[P]: PS) = empChan .

eq enq(empChan, M2) = M2 | empChan .
eq enq(M1 | MS, M2) = M1 | enq(MS, M2) .

eq #(empConfig, P) = 0 .
eq #((p-state[P1]: PS) CF, P) = if P == P1 then 1 + #(CF, P)
  else #(CF, P) fi .
eq #((c-state[P1, Q1, N1]: CS) CF, P) = #(CF, P) .
op # : Config Pid Pid Nat -> Nat .

eq #(empConfig, P, Q, N) = 0 .
eq #((p-state[P1] : PS) CF, P, Q, N) = #(CF, P, Q, N) .
eq #((c-state[P1, Q1, N1]: CS) CF, P, Q, N) = if (P == P1)
  and (Q == Q1) and (N == N1) then 1 + #(CF, P, Q, N) else #(CF, P, Q, N) fi .

```

The three transition rules will be written.

```

rl [chgStt] :
(p-state[P]: PS1)
=>
(p-state[P]: PS2) .

rl [sndMsg] :
(p-state[P]: PS1) (c-state[P,Q,N]: CS)
=>
(p-state[P]: PS2) (c-state[P,Q,N]: enq(CS,M)) .

rl [recMsg] :
(p-state[P]: PS1) (c-state[Q,P,N]: M | CS)
=>
(p-state[P]: PS2) (c-state[Q,P,N]: CS) .

```

Next, the meta-program corresponding to the function CL will be written, in which the specification of a UDS is treated as input data of the meta-program. Taking look at the meta-programming facilities in Maude, META-LEVEL module, in which terms and modules can be meta-represented, can be imported to use to generate the specification of the UDS superimposed by $CLDSA$ from the specification of a UDS . The meta-program corresponding to the function CL will be written in the module CLDSA in Maude that import the function module META-LEVEL, which is imported to write meta-programs in Maude. Module CLDSA define the meta-program corresponding to the function CL as the function CL that takes a module, namely module UDS, as an its argument and return another module.

```
mod CLDSA is
pr META-LEVEL .

op CL : Module -> Module .
```

The function CL will call three other functions CLstate, CLinit and CLtrans corresponding to CL_{State} , CL_{Init} and CL_{Trans} , respectively.

```
op CLstate : Module -> Module .
op CLinit : Module -> Module .
op CLtrans : ReluSet -> ReluSet .
```

Function CLstate use other functions to takes a part of the specifications of M_{UDS} and return a part of the specification of M_{CLDSA} .

```
op modImportList : ImportList -> ImportList .
op modSortSet : SortSet -> SortSet .
op modSubsortDeclSet : SubsortDeclSet -> SubsortDeclSet .
op modOpDeclSet : OpDeclSet -> OpDeclSet .
op modEquationSet : EquationSet -> EquationSet .
```

Function CLtrans use other functions to takes the specification of the transition rules of M_{UDS} and return the specification of the transition rules of M_{CLDSA} .

```
op getReluSet : Module -> ReluSet .
op modReluSet : ReluSet -> ReluSet .
op addReluSet : ReluSet ReluSet -> Module .
```

Beside, the more faithful formalization of the DSR property will be specified based on the two specifications. After that, we will design and implement a model checker as a meta-program in Maude. To implement the model checker, metalevel-operations in

META-LEVEL module such as metaRewrite and metaSearch, which can be used to handle a specification, can be used to analyze computations of distributed systems. Continuously, we will conduct some similar studies on other meta-distributed algorithms such as Dijkstra-Scholten algorithm, Chandy-Misra-Haas algorithm and Koo-Toueg coordinated checkpointing to come up with a generic way to specify meta-distributed algorithms, their desired properties and model check that these algorithms enjoy such properties. The process to model check that meta-distributed algorithms enjoy desired properties is depicted in Fig. 8.2.

The concept “meta-distributed algorithm” is brand-new. One possible important achievement of the future work is to clarify meta-distributed algorithms and identify a class of meta-distributed algorithms and the main achievements of the research are model checking techniques and a model checker suited for the class of meta-distributed algorithms that cannot be reasonably well tackled by any existing model checkers.

8.2 Chapter Summary

This chapter mentioned the future work. As what is mentioned, the future work is to come up with how to specify meta-distributed algorithms, how to specify their desired properties, how to model check that meta-distributed algorithms enjoy their desired properties, and design and implement a model checker for meta-distributed algorithms. In detail, the chapter gave the concept of meta-distributed algorithms, then discussed the problems to resolve.

The chapter then discussed the procedures and the contents of the future work. The plan were also mentioned by which the first to do is to write a meta-program in Maude that corresponds to the function CL , The next to do is to implement a model checker as a meta-program in Maude that can directly deal with the more faithful formalization of the DSR property and similar case studies will be repeated, coming up with a generic way to specify meta-distributed algorithms, their desired properties and model check that such algorithms enjoy such properties.

At the end of the chapter, the desired achievements of the future work were discussed by which the main desired achievements of the research are model checking techniques and a model checker suited for the class of meta-distributed algorithms that cannot be reasonably well tackled by any existing model checkers.

Chapter 9

Conclusion

The Contribution of the Research

Carefully taking into account an informal description of the \mathcal{DSR} property, we have given a new formal definition of the property, which more faithfully express the informal description of the \mathcal{DSR} property.

Since the definition of the \mathcal{DSR} property in the existing study involves only one state machine $M_{\mathcal{CLDSA}}$ and our definition involves two state machines $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$, our definition is seemingly different from the one used in the existing study. It is checked that \mathcal{CLDSA} terminates in $M_{\mathcal{CLDSA}}$ and then the snapshot and the final state are reachable from the start state and the snapshot, respectively, in $M_{\mathcal{UDS}}$ in our definition, while it is checked that the snapshot and the final state are reachable from the start state and the snapshot, respectively, in $M_{\mathcal{CLDSA}}$ in their definition. However, we have conducted an analysis on the similarities between our definition and the existing definition. We recognized that the new definition is equivalent to the existing definition. We have proved Theorem 1 saying that our formalization of the \mathcal{DSR} property is equivalent to the existing definition for each $M_{\mathcal{UDS}}$. The theorem guarantees the validity of the model checking approach used in the existing study. Since the existing definition have been model checked with Maude and the new definition is not straightforward to be directly model checked. The theorem also guarantees that we can use the model checking approach used in the existing study to model check the new definition. To prove Theorem 1, we have proven Proposition 1 and Lemma 1. It is required to prove two other lemmas, Lemma 2 and Lemma 3, to prove Lemma 1. The proof of Theorem 1 follows from Proposition 1 and Lemma 1.

Moreover, we have proposed a binary relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$ and then proved that $M_{\mathcal{CLDSA}}$ simulates $M_{\mathcal{UDS}}$ and vice versa, namely we have proved another theorem saying that the binary relation is a bi-simulation relation between $M_{\mathcal{UDS}}$ and $M_{\mathcal{CLDSA}}$, to guarantee that \mathcal{CLDSA} does not alter the behaviors of the \mathcal{UDS} .

The contributions of our research are:

- Completely giving the more faithful formal definition of the \mathcal{DSR} property, which

more faithfully express the informal description of the \mathcal{DSR} property.

- Proving a theorem saying that our formalization of the \mathcal{DSR} property is equivalent to the existing definition for each M_{UDS} to guarantee that the validity of the model checking approach used in the existing study and the existing model checking approach can be used to model check the new definition.
- Proposing a binary relation between M_{UDS} and $M_{\mathcal{CLDSA}}$ and then proving another theorem asserting that $M_{\mathcal{CLDSA}}$ simulates M_{UDS} and vice versa to guarantee that \mathcal{CLDSA} does not alter the behaviors of the UDS .

What I Have Done on the Research

On the research, what I have done are as follows:

1. Learning necessary basic technical knowledge, such as underlying distributed systems, state machine, model checking and Maude specification and programming language.
2. Learning \mathcal{CLDSA} and the \mathcal{DSR} property.
3. Investigating how to specify \mathcal{CLDSA} and how to model check that the algorithm enjoy the \mathcal{DSR} property in the existing study.
4. Investigating how to formalize a UDS and the UDS superimposed by \mathcal{CLDSA} as state machines M_{UDS} and $M_{\mathcal{CLDSA}}$.
5. Studying on completely giving the more faithful formal definition of the \mathcal{DSR} property.
6. Proving a theorem saying that the new definition of the \mathcal{DSR} property is equivalent to the existing definition for each M_{UDS} .
7. Proposing a binary relation between M_{UDS} and $M_{\mathcal{CLDSA}}$ and then proving another theorem asserting that $M_{\mathcal{CLDSA}}$ simulates M_{UDS} and vice versa.

Summary of the Future Work

In the future, we want to come up with how to specify meta-distributed algorithms, how to specify their desired properties, how to model check that meta-distributed algorithms enjoy their desired properties, and design and implement a model checker for meta-distributed algorithms. We want to clarify meta-distributed algorithms and identify a class of meta-distributed algorithms. The main expected achievements of the future work is model checking techniques and a model checker suited for the class of meta-distributed algorithms that cannot be reasonably well tackled by any existing model checkers.

Appendix A

Specification of \mathcal{CLDSA} in Maude

```
***  
*** Specification of the Chandy–Lamport Distributed Snapshot  
Algorithm in Maude  
***
```

```
fmod PID is  
  pr NAT .  
  sort Pid .  
  op p : Nat -> Pid [ctor] .  
endfm
```

```
fmod TOKEN is  
  pr NAT .  
  sort Token .  
  op t : Nat -> Token [ctor] .  
endfm
```

```
fmod MARKER is  
  sort Marker .  
  op marker : -> Marker [ctor] .  
endfm
```

```
fmod MESSAGE is  
  pr TOKEN .  
  pr MARKER .  
  sort Msg .  
  subsorts Token Marker < Msg .  
endfm
```

```
fmod CHANNEL is
```

```

pr MESSAGE .
sorts EmpChan NeChan Chan .
subsorts EmpChan NeChan < Chan .
op empChan : -> EmpChan [ctor] .
op _|_ : Msg Chan -> NeChan [ctor] .
op put : Chan Msg -> NeChan .
op delMC : Chan -> Chan .

vars M1 M2 : Msg .
var C : Chan .
var T : Token .

eq put(empChan,M2) = M2 | empChan .
eq put(M1 | C,M2) = M1 | put(C,M2) .

endfm

fmod PROCESS-STATE is
  pr BOOL .
  pr TOKEN .
  sort PState .
  subsort Token < PState .
  op noToken : -> PState [ctor] .
  op _--_ : PState PState -> PState [ctor assoc comm
  id: noToken] .

  var T : Token .
  eq T T = T .
endfm

fmod PROGRESS is
  sort Prog .
  ops notYet started completed : -> Prog [ctor] .
endfm

fmod OBSERVABLE-COMPONENT is
  pr PID .
  pr CHANNEL .
  pr PROCESS-STATE .
  pr PROGRESS .
  sort OCom .

  *** p-state[p] is the state of process p.

```

```

op (p-state[_] :_) : Pid PState -> OCom [ctor] .
*** c-state[p,q,n] is the n th channel from p to q.
op (c-state[-,-,-] :_) : Pid Pid Nat Chan -> OCom [ctor] .
*** When cnt becomes 0, the snapshot has been taken.
op (cnt :_) : Nat -> OCom [ctor] .
*** the number of markers not yet received by a process.
op (#ms[_] :_) : Pid Nat -> OCom [ctor] .
*** indicating whether a marker has been received from
*** a channel from p to q.
op (done[-,-,-] :_) : Pid Pid Nat Bool -> OCom [ctor] .
*** indicating that a process has not yet started ,
*** has started ,
*** or completed the algorithm.
op (prog[_] :_) : Pid Prog -> OCom [ctor] .
*** indicating whether messages are consumed.
op (consume :_) : Bool -> OCom [ctor] .
endfm

```

```

fmod CONFIGURATIONS is
  pr OBSERVABLE-COMPONENT .
  sort Config .
  subsort OCom < Config .
  op empConfig : -> Config [ctor] .
  op _ _ : Config Config -> Config [ctor assoc comm id:
  empConfig] .

  var OC : OCom .
  eq OC OC = OC .

  var CF : Config .
  vars P' P Q : Pid .
  var PS : PState .
  var C : Chan .
  var M : Msg .
  var N : Nat .

  op bcast : Config Pid Marker -> Config .
  op inchans : Config Pid -> Config .
  op delM : Config -> Config .

```

— Function bcast to puts markers in all the outgoing channels from a process P;
 eq bcast(empConfig,P,M) = empConfig .

```

eq bcast((c-state[P,Q,N] : C) CF,P,M)
= (c-state[P,Q,N] : put(C,M)) bcast(CF,P,M) .
eq bcast(OC CF,P,M) = OC bcast(CF,P,M) [owise] .

```

—— Function `inchans` to initialize the states of all incoming channels of a process as empty channel;

```

eq inchans(empConfig,P) = empConfig .
eq inchans((c-state[Q,P,N] : C) CF,P)
= (c-state[Q,P,N] : empChan) inchans(CF,P) .
eq inchans(OC CF,P) = inchans(CF,P) [owise] .

```

endfm

```

fmod META-CONFIGURATION-COMPONENT is
pr CONFIGURATIONS .
sort MComp .

```

```

ops base-state start-state finish-state
snapshot control : Config -> MComp .

```

endfm

```

fmod META-CONFIGURATION is
pr META-CONFIGURATION-COMPONENT .
sort MConfig .
subsort MComp < MConfig .
op -- : MConfig MConfig -> MConfig [assoc comm] .

```

```

var MOC : MComp .
eq MOC MOC = MOC .

```

endfm

```

fmod INIT-META-CONFIG is
pr META-CONFIGURATION .

```

*** (

Let us consider the following system:

- There are two processes $p(0)$, $p(1)$.
- There is one token $t(0)$ in the system.
- The state of each process only depends on the tokens

owned by the process. So, the state of each process can be expressed as $\backslash\text{empty}, \{t(0)\}$. Initially, $p(0)$ has the token $t(0)$, and $p(1)$ does not have any tokens.

– There are two channels: $p(0) \longrightarrow p(1)$, $p(1) \longrightarrow p(0)$. Initially each channel is empty.

– Each process repeatedly does the following:

i. If the process has a token, then it puts the token in the outgoing channel. Accordingly its state changes.

ii. If the incoming channel to the process is not empty, then the process gets the token from it. Accordingly its state changes.

Let imc00 be the initial state of the system.

```
)
  op imc00 :  $\rightarrow$  MConfig .
  eq imc00
  = base-state((p-state[p(0)]: t(0)) (p-state[p(1)]: noToken)
    (c-state[p(0),p(1),0]: empChan) (c-state[p(1),p(0),0]: empChan))
  start-state(empConfig)
  finish-state(empConfig)
  snapshot(empConfig)
  control((cnt : 2) (#ms[p(0)]: 1) (#ms[p(1)]: 1)
    (done[p(0),p(1),0]: false) (done[p(1),p(0),0]: false)
    (prog[p(0)]: notYet) (prog[p(1)]: notYet)
    (consume : false)) .
```

***(

Let us consider the following system:

– There are two processes $p(0)$, $p(1)$.

– There is one token $t(0)$ in the system.

– The state of each process only depends on the tokens owned by the process. So, the state of each process can be expressed as $\backslash\text{empty}, \{t(0)\}$. Initially, $p(0)$ has the token $t(0)$, and $p(1)$ does not have any tokens.

- There are two channels: $p(0) \longrightarrow p(1)$, $p(1) \longrightarrow p(0)$. Initially each channel is empty.
- Each process repeatedly does the following:
 - i. The process may consume a token owned by the process.
 - ii. If the process has a token, then it puts the token in the outgoing channel. Accordingly its state changes.
 - iii. If the incoming channel to the process is not empty, then the process gets the token from it. Accordingly its state changes.

Let `imc01` be the initial state of the system.

```
)
  op imc01 : -> MConfig .
  eq imc01
  = base-state((p-state[p(0)]: t(0))
    (p-state[p(1)]: noToken)
    (c-state[p(0),p(1),0]: empChan)
    (c-state[p(1),p(0),0]: empChan))
  start-state(empConfig)
  finish-state(empConfig)
  snapshot(empConfig)
  control((cnt : 2) (#ms[p(0)]: 1) (#ms[p(1)]: 1)
    (done[p(0),p(1),0]: false) (done[p(1),p(0),0]: false)
    (prog[p(0)]: notYet) (prog[p(1)]: notYet)
    (consume : true)) .
```

```
  op imc02 : -> MConfig .
  eq imc02
  = base-state((p-state[p(0)]: t(0))
    (p-state[p(1)]: noToken)
    (p-state[p(2)]: t(2))
    (c-state[p(0),p(1),0]: empChan)
    (c-state[p(0),p(1),1]: (t(1) | empChan)))
  start-state(empConfig)
  finish-state(empConfig)
```

```

snapshot(empConfig)
control((cnt : 3) (#ms[p(0)]: 0)
(#ms[p(1)]: 2) (#ms[p(2)]: 0)
(done[p(0),p(1),0]: false)
(done[p(0),p(1),1]: false)
(prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet)
(consume : true)) .

```

```

op imc03 : -> MConfig .
eq imc03
  = base-state((p-state[p(0)]: (t(0) t(1)))
    (p-state[p(1)]: noToken)
    (c-state[p(0),p(1),0]: empChan)
    (c-state[p(0),p(1),1]: empChan)
    (c-state[p(1),p(0),0]: empChan))
  start-state(empConfig)
  finish-state(empConfig)
  snapshot(empConfig)
  control((cnt : 2) (#ms[p(0)]: 1) (#ms[p(1)]: 2)
    (done[p(0),p(1),0]: false) (done[p(0),p(1),1]: false)
    (done[p(1),p(0),0]: false)
    (prog[p(0)]: notYet) (prog[p(1)]: notYet)
    (consume : true)) .

```

```

op imc04 : -> MConfig .
eq imc04
  = base-state((p-state[p(0)]: (t(0) t(1)))
    (p-state[p(1)]: noToken)
    (p-state[p(2)]: noToken)
    (c-state[p(0),p(1),0]: empChan)
    (c-state[p(0),p(2),0]: empChan)
    (c-state[p(1),p(0),0]: empChan)
    (c-state[p(1),p(2),0]: empChan))
  start-state(empConfig)
  finish-state(empConfig)
  snapshot(empConfig)
  control((cnt : 3) (#ms[p(0)]: 1)
    (#ms[p(1)]: 1) (#ms[p(2)]: 2)
    (done[p(0),p(1),0]: false)
    (done[p(0),p(2),0]: false)

```



```

      (done[p(1),p(0),0]: false)
      (done[p(1),p(2),0]: false)
      (prog[p(0)]: notYet) (prog[p(1)]: notYet)
      (prog[p(2)]: notYet)
      (consume : false)) .

```

```

op imc05 : -> MConfig .
eq imc05
= base-state((p-state[p(0)]: (t(0) t(1)))
  (p-state[p(1)]: noToken)
  (p-state[p(2)]: noToken)
  (c-state[p(0),p(1),0]: empChan)
  (c-state[p(0),p(2),0]: empChan)
  (c-state[p(1),p(0),0]: empChan)
  (c-state[p(1),p(2),0]: empChan))
start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 3) (#ms[p(0)]: 1)
  (#ms[p(1)]: 1) (#ms[p(2)]: 2)
  (done[p(0),p(1),0]: false)
  (done[p(0),p(2),0]: false)
  (done[p(1),p(0),0]: false)
  (done[p(1),p(2),0]: false)
  (prog[p(0)]: notYet) (prog[p(1)]: notYet)
  (prog[p(2)]: notYet)
  (consume : true)) .

```

```

op imc06 : -> MConfig .
eq imc06
= base-state((p-state[p(0)]: (t(0) t(1)))
  (p-state[p(1)]: noToken)
  (p-state[p(2)]: noToken)
  (c-state[p(0),p(1),0]: empChan)
  (c-state[p(0),p(1),1]: empChan)
  (c-state[p(1),p(2),0]: empChan)
  (c-state[p(2),p(0),0]: empChan))
start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 3) (#ms[p(0)]: 1)

```

```

(#ms[p(1)]: 2) (#ms[p(2)]: 1)
(done[p(0),p(1),0]: false)
(done[p(0),p(1),1]: false)
(done[p(1),p(2),0]: false)
(done[p(2),p(0),0]: false)
(prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet)
(consume : true)) .

```

```

op imc07 : -> MConfig .
eq imc07
= base-state((p-state[p(0)]: (t(0) t(1) t(2)))
  (p-state[p(1)]: noToken)
  (p-state[p(2)]: noToken)
  (c-state[p(0),p(1),0]: empChan)
  (c-state[p(0),p(1),1]: empChan)
  (c-state[p(1),p(2),0]: empChan)
  (c-state[p(2),p(0),0]: empChan))
start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 3) (#ms[p(0)]: 1)
  (#ms[p(1)]: 2) (#ms[p(2)]: 1)
  (done[p(0),p(1),0]: false)
  (done[p(0),p(1),1]: false)
  (done[p(1),p(2),0]: false)
  (done[p(2),p(0),0]: false)
  (prog[p(0)]: notYet) (prog[p(1)]: notYet)
  (prog[p(2)]: notYet)
  (consume : true)) .

```

```

op imc08 : -> MConfig .
eq imc08
= base-state((p-state[p(0)]: (t(0) t(1)))
  (p-state[p(1)]: noToken)
  (p-state[p(2)]: noToken)
  (c-state[p(0),p(1),0]: empChan)
  (c-state[p(0),p(1),1]: empChan)
  (c-state[p(0),p(2),0]: empChan)
  (c-state[p(1),p(2),0]: empChan)
  (c-state[p(2),p(0),0]: empChan))

```

```

start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 3) (#ms[p(0)]: 1)
  (#ms[p(1)]: 2) (#ms[p(2)]: 2)
  (done[p(0),p(1),0]: false)
  (done[p(0),p(1),1]: false)
  (done[p(0),p(2),0]: false)
  (done[p(1),p(2),0]: false)
  (done[p(2),p(0),0]: false)
  (prog[p(0)]: notYet) (prog[p(1)]: notYet)
  (prog[p(2)]: notYet)
  (consume : true)) .

```

```

op imc09 : -> MConfig .
eq imc09
= base-state((p-state[p(0)]: noToken)
  (p-state[p(1)]: noToken) (p-state[p(2)]: noToken)
  (p-state[p(3)]: t(0)) (p-state[p(4)]: noToken)
  (c-state[p(0),p(1),0]: empChan)
  (c-state[p(1),p(2),0]: empChan)
  (c-state[p(2),p(3),0]: empChan)
  (c-state[p(2),p(4),0]: empChan)
  (c-state[p(3),p(0),0]: empChan)
  (c-state[p(3),p(0),1]: empChan)
  (c-state[p(3),p(2),0]: empChan)
  (c-state[p(4),p(3),0]: empChan))
start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 5) (#ms[p(0)]: 2) (#ms[p(1)]: 1)
  (#ms[p(2)]: 2) (#ms[p(3)]: 2) (#ms[p(4)]: 1)
  (done[p(0),p(1),0]: false)
  (done[p(1),p(2),0]: false)
  (done[p(2),p(3),0]: false)
  (done[p(2),p(4),0]: false)
  (done[p(3),p(0),0]: false)
  (done[p(3),p(0),1]: false)
  (done[p(3),p(2),0]: false)
  (done[p(4),p(3),0]: false)
  (prog[p(0)]: notYet) (prog[p(1)]: notYet)
  (prog[p(2)]: notYet)

```

```

      (prog[p(3)]: notYet) (prog[p(4)]: notYet)
      (consume : true)) .

```

```

***

```

```

op imc10 : -> MConfig .
eq imc10
  = base-state((p-state[p(0)]: (t(0) t(1)))
    (p-state[p(1)]: noToken) (p-state[p(2)]: noToken)
    (p-state[p(3)]: noToken) (p-state[p(4)]: noToken)
    (c-state[p(0),p(1),0]: empChan)
    (c-state[p(1),p(2),0]: empChan)
    (c-state[p(2),p(3),0]: empChan)
    (c-state[p(2),p(4),0]: empChan)
    (c-state[p(3),p(0),0]: empChan)
    (c-state[p(3),p(0),1]: empChan)
    (c-state[p(3),p(2),0]: empChan)
    (c-state[p(4),p(3),0]: empChan))
  start-state(empConfig)
  finish-state(empConfig)
  snapshot(empConfig)
  control((cnt : 5) (#ms[p(0)]: 2) (#ms[p(1)]: 1)
    (#ms[p(2)]: 2) (#ms[p(3)]: 2) (#ms[p(4)]: 1)
    (done[p(0),p(1),0]: false)
    (done[p(1),p(2),0]: false)
    (done[p(2),p(3),0]: false)
    (done[p(2),p(4),0]: false)
    (done[p(3),p(0),0]: false)
    (done[p(3),p(0),1]: false)
    (done[p(3),p(2),0]: false)
    (done[p(4),p(3),0]: false)
    (prog[p(0)]: notYet) (prog[p(1)]: notYet)
    (prog[p(2)]: notYet)
    (prog[p(3)]: notYet) (prog[p(4)]: notYet)
    (consume : true)) .

```

```

endfm

```

```

mod CHANDY-LAMPORT is
  pr META-CONFIGURATION .
  vars BC CC SC SSC : Config .
  vars P' P Q : Pid .
  var T : Token .
  var PS : PState .
  var N : Nat .

```

```

vars C C' : Chan .
vars NzN NzN' : NzNat .

***** Consumption of Tokens *****
***
*** When a distributed snapshot has been taken ,
*** we intentionally
*** stop the base computation because we want
*** not to make the
*** size of the reachable state space too large.
***
*** Process P only changes its state.
rl [chgStt] :
  base-state((p-state[P] : (T PS)) BC)
  finish-state(empConfig)
  control((consume : true) CC)
=>
  base-state((p-state[P] : PS) BC)
  finish-state(empConfig)
  control((consume : true) CC) .

***** Sending of Tokens *****
*** Process P sends a token to process Q.
rl [sndTkn] :
  base-state((p-state[P] : (T PS))
  (c-state[P,Q,N] : C) BC)
  finish-state(empConfig)
=>
  base-state((p-state[P] : PS)
  (c-state[P,Q,N] : put(C,T)) BC)
  finish-state(empConfig) .

***** Receipt of Tokens *****
*** Process P receives a token along an incoming channel.
*** case-1: The process has not yet started the algorithm.
*** Note: No need (done[Q,P,N] : false) on both sides.
rl [recTkn&notYet&~done] :
  base-state((p-state[P] : PS)
  (c-state[Q,P,N] : T | C) BC)
  finish-state(empConfig)
  control((prog[P] : notYet) CC)

```

```

=>
base-state((p-state[P] : (T PS))
(c-state[Q,P,N] : C) BC)
finish-state(empConfig)
control((prog[P] : notYet) CC) .

*** case-2: The process has completed the algorithm.
*** Note: No need (done[Q,P,N] : true) on both sides.
rl [recTkn&completed&done] :
  base-state((p-state[P] : PS)
(c-state[Q,P,N] : T | C) BC)
  finish-state(empConfig)
  control((prog[P] : completed) CC)
=>
base-state((p-state[P] : (T PS))
(c-state[Q,P,N] : C) BC)
finish-state(empConfig)
control((prog[P] : completed) CC) .

*** case-3: The process has started the algorithm,
*** namely that it has already recorded its state,
*** not yet completed it, and has not yet received a marker
*** from the incoming channel.
rl [recTkn&started&~done] :
  base-state((p-state[P] : PS) (c-state[Q,P,N] : T | C) BC)
  snapshot((c-state[Q,P,N] : C') SSC)
  finish-state(empConfig)
  control((prog[P] : started) (done[Q,P,N] : false) CC)
=>
base-state((p-state[P] : (T PS)) (c-state[Q,P,N] : C) BC)
snapshot((c-state[Q,P,N] : put(C',T)) SSC)
finish-state(empConfig)
control((prog[P] : started) (done[Q,P,N] : false) CC) .

*** case-4: The process has started the algorithm,
*** not yet completed it, and has already received a marker
*** from the incoming channel.
rl [recTkn&started&done] :
  base-state((p-state[P] : PS) (c-state[Q,P,N] : T | C) BC)
  finish-state(empConfig)
  control((prog[P] : started) (done[Q,P,N] : true) CC)
=>
base-state((p-state[P] : (T PS)) (c-state[Q,P,N] : C) BC)

```

```

finish-state(empConfig)
control((prog[P] : started) (done[Q,P,N] : true) CC) .

```

```

***** Record of Process States *****
*** Process P starts taking the distributed snapshot.
*** case-1: The process globally initiates the algorithm,
*** namely the first process that records its state in the
*** system.
*** case-2: The process does not, namely that there exists
*** another process
*** that has globally initiated the algorithm.
*** case-1: is further split into three sub-cases:
*** case-1-1: The underlying system only consists of
*** the process.
*** Note: finish-state should be added.
rl [start&cnt=1&#ms=0] :
  base-state((p-state[P] : PS))
  start-state(empConfig)
  snapshot(empConfig)
  finish-state(empConfig)
  control((cnt : 1) (prog[P] : notYet)
  (#ms[P] : 0) CC)
=>
  base-state((p-state[P] : PS))
  start-state((p-state[P] : PS))
  snapshot((p-state[P] : PS))
  finish-state((p-state[P] : PS))
  control((cnt : 0) (prog[P] : completed)
  (#ms[P] : 0) CC) .

*** case-1-2: The system consists of more than one process,
*** and the process does not have any incoming channels.
crl [start&cnt>1&#ms=0] :
  base-state((p-state[P] : PS) BC)
  start-state(empConfig)
  snapshot(empConfig)
  control((cnt : NzN') (prog[P] : notYet)
  (#ms[P] : 0) CC)
=>
  base-state((p-state[P] : PS) bcast(BC,P,marker))
  start-state((p-state[P] : PS) BC)
  snapshot((p-state[P] : PS))

```

```

    control((cnt : sd(NzN',1)) (prog[P] : completed)
    (#ms[P] : 0) CC)
if NzN' > 1 .

```

*** case-1-3: The system consists of more than one process ,
 *** and the process has one or more incoming channels.

```

rl [start&cnt>1&#ms>0] :
  base-state((p-state[P] : PS) BC)
  start-state(empConfig)
  snapshot(empConfig)
  control((prog[P] : notYet) (#ms[P] : NzN') CC)
=>
  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) .

```

*** case-2: The process does not, namely that there exists
 *** another process
 *** that has globally initiated the algorithm.
 *** case-2: is further split into three sub-cases:
 *** case-2-1: The process does not have any incoming
 *** channels ,
 *** and there are no processes except for the process
 *** that have not completed the algorithm.
 *** Note: finish-state should be added.

```

crl [record&cnt=1&#ms=0] :
  base-state((p-state[P] : PS))
  start-state(SC)
  snapshot(SSC)
  finish-state(empConfig)
  control((cnt : 1) (prog[P] : notYet) (#ms[P] : 0) CC)
=>
  base-state((p-state[P] : PS))
  start-state(SC)
  snapshot((p-state[P] : PS) SSC)
  finish-state((p-state[P] : PS))
  control((cnt : 0) (prog[P] : completed) (#ms[P] : 0) CC)
if (SC /= empConfig) .

```

*** case-2-2: The process does not have any incoming channels ,
 *** and there are some other processes that have not completed
 the algorithm.


```

crl [record&cnt>1&#ms=0] :
  base-state((p-state[P] : PS) BC)
  start-state(SC)
  snapshot(SSC)
  control((cnt : NzN') (prog[P] : notYet)
    (#ms[P] : 0) CC)
=>
  base-state((p-state[P] : PS) bcast(BC, P, marker))
  start-state(SC)
  snapshot((p-state[P] : PS) SSC)
  control((cnt : sd(NzN',1))
    (prog[P] : completed)
    (#ms[P] : 0) CC)
if (NzN' > 1) /\ (SC != empConfig) .

*** case-2-3: The process has some incoming channels.
crl [record&cnt>1&#ms>0] :
  base-state((p-state[P] : PS) BC)
  start-state(SC)
  snapshot(SSC)
  control((prog[P] : notYet) (#ms[P] : NzN') CC)
=>
  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 != empConfig) .

```

```

***** Receipt of Markers *****
*** Process P receives a marker along an incoming channel.
*** case-1: The process has not yet started the algorithm.
*** case-2: The process has already started the algorithm.
*** case-1 is further split into three sub-cases:
*** case-1-1: The process has only one incoming channel,
*** and there are no processes that have not yet completed
*** the algorithm
*** except for the process, which implies that the process
*** does not have any outgoing channels.
*** Note: finish-state should be added.
rl [recMkr&notYet&#ms=1&cnt=1] :
  base-state((p-state[P] : PS)
    (c-state[Q,P,N] : marker | C) BC)

```

```

    snapshot(SSC)
    finish-state(empConfig)
    control((prog[P] : notYet)
    (#ms[P] : 1) (cnt : 1)
    (done[Q,P,N] : false) CC)
=>
    base-state((p-state[P] : PS)
    (c-state[Q,P,N] : C) BC)
    snapshot((p-state[P] : PS)
    (c-state[Q,P,N] : empChan) SSC)
    finish-state((p-state[P] : PS)
    (c-state[Q,P,N] : C) BC)
    control((prog[P] : completed)
    (#ms[P] : 0) (cnt : 0)
    (done[Q,P,N] : true) CC) .

*** case-1-2: The process has only one incoming channel ,
*** and there are some other processes that have
*** not yet completed the algorithm.
crl [recMkr&notYet&#ms=1&cnt>1] :
    base-state((p-state[P] : PS)
    (c-state[Q,P,N] : marker | C) BC)
    snapshot(SSC)
    control((prog[P] : notYet) (#ms[P] : 1)
    (cnt : NzN)
    (done[Q,P,N] : false) CC)
=>
    base-state((p-state[P] : PS)
    (c-state[Q,P,N] : C)
    bcast(BC,P,marker))
    snapshot((p-state[P] : PS )
    (c-state[Q,P,N] : empChan) SSC)
    control((prog[P] : completed)
    (#ms[P] : 0) (cnt : sd(NzN,1))
    (done[Q,P,N] : true) CC)
if NzN > 1 .

*** case-1-3: The process has more than one incoming channel.
crl [recMkr&notYet&#ms>1&cnt>1] :
    base-state((p-state[P] : PS)
    (c-state[Q,P,N] : marker | C) BC)
    snapshot(SSC)
    control((prog[P] : notYet)

```

```

(#ms[P] : NzN') (cnt : NzN)
(done[Q,P,N] : false) CC)
=>
base-state((p-state[P] : PS)
(c-state[Q,P,N] : C)
bcast(BC,P,marker))
snapshot((p-state[P] : PS )
(c-state[Q,P,N] : empChan)
inchans(BC,P) SSC)
control((prog[P] : started)
(#ms[P] : sd(NzN', 1)) (cnt : NzN)
(done[Q,P,N] : true) CC)
if NzN' > 1 .

*** case-2: The process has already started the algorithm.
*** case-2 is further split into three sub-cases:
*** case-2-1: There are no incoming channels
*** from which markers have not been
*** received except for the incoming channel,
*** and there are no processes
*** that have not yet completed the algorithm
*** except for the process.
rl [recMkr&started&#ms=1&cnt=1] :
  base-state((p-state[P] : PS)
(c-state[Q,P,N] : marker | C) BC)
  finish-state(empConfig)
  control((prog[P] : started)
(#ms[P] : 1) (cnt : 1)
(done[Q,P,N] : false) CC)
=>
  base-state((p-state[P] : PS)
(c-state[Q,P,N] : C) BC)
  finish-state((p-state[P] : PS)
(c-state[Q,P,N] : C) BC)
  control((prog[P] : completed)
(#ms[P] : 0) (cnt : 0)
(done[Q,P,N] : true) CC) .

*** case-2-2: 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 the algorithm.

```

```

*** Note: finish-state should not be added.
crl [recMkr&started&#ms=1&cnt>1] :
  base-state((p-state[P] : PS)
    (c-state[Q,P,N] : marker | C) BC)
  control((prog[P] : started)
    (#ms[P] : 1) (cnt : NzN)
    (done[Q,P,N] : false) CC)
=>
  base-state((p-state[P] : PS)
    (c-state[Q,P,N] : C) BC)
  control((prog[P] : completed)
    (#ms[P] : 0) (cnt : sd(NzN,1))
    (done[Q,P,N] : true) CC)
if NzN > 1 .

*** case-2-3: There are some other incoming channels
*** from which markers have not
*** been received.
*** Note: finish-state should not be added.
crl [recMkr&started&#ms>1&cnt>1] :
  base-state((p-state[P] : PS)
    (c-state[Q,P,N] : marker | C) BC)
  control((prog[P] : started)
    (#ms[P] : NzN') (cnt : NzN)
    (done[Q,P,N] : false) CC)
=>
  base-state((p-state[P] : PS)
    (c-state[Q,P,N] : C) BC)
  control((prog[P] : started)
    (#ms[P] : sd(NzN',1)) (cnt : NzN)
    (done[Q,P,N] : true) CC)
if NzN' > 1 .
endm

mod EXPERIMENT is
  pr CHANDY-LAMPORT .
  pr INIT-META-CONFIG .
  vars SC FC SSC : Config .
  vars MC : MConfig .
endm

```

Appendix B

Model Checking of \mathcal{CLDSA} in Maude

```
***
*** Model Checking of the Distributed Snapshot Reachability
*** Property
***
```

```
*** Experiment for imc00 ***
```

```
*** states: 164
search in EXPERIMENT : imc00 =>* MC such that false .
```

```
*** Solution 40 (state 163)
*** states: 164 rewrites: 1341 in 18ms cpu
*** (6159ms real) (70817 rewrites/second)
search in EXPERIMENT :
  imc00 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig .
```

```
*** Solution 40 (state 163)
*** states: 164 rewrites: 1578 in 25ms cpu
*** (5896ms real) (61566 rewrites/second)
search in EXPERIMENT :
  imc00 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
      (consume : false))
  =>
  base-state(SSC) finish-state(empConfig)
```

```

        control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
                (consume : false)) .

*** Solution 40 (state 163)
*** states: 164  rewrites: 1575 in 26ms cpu
*** (5871ms real) (60057 rewrites/second)
search in EXPERIMENT :
    imc00 =>* start-state(SC) finish-state(FC)
    snapshot(SSC) MC
    such that FC /= empConfig
    /\ base-state(SSC) finish-state(empConfig)
       control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
               (consume : false))

    =>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (consume : false)) .

*** End of Experiment for imc00 ***

```

```

*** Experiment for imc01 ***

```

```

*** states: 239
search in EXPERIMENT : imc01 =>* MC such that false .

```

```

*** Solution 55 (state 238)
*** states: 239  rewrites: 1968 in 27ms cpu
*** (2169ms real) (70386 rewrites/second)
search in EXPERIMENT :
    imc01 =>* start-state(SC) finish-state(FC)
    snapshot(SSC) MC
    such that FC /= empConfig .

```

```

*** Solution 55 (state 238)
*** states: 239  rewrites: 2396 in 38ms cpu
*** (2153ms real) (62003 rewrites/second)
search in EXPERIMENT :
    imc01 =>* start-state(SC) finish-state(FC)
    snapshot(SSC) MC
    such that FC /= empConfig
    /\ base-state(SC) finish-state(empConfig)
       control((prog[p(0)]: notYet) (prog[p(1)]: notYet)

```

```

        (consume : true))
=>
    base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (consume : true)) .

*** Solution 55 (state 238)
*** states: 239  rewrites: 2360 in 38ms cpu
*** (2324ms real) (61615 rewrites/second)
search in EXPERIMENT :
    imc01 =>* start-state(SC) finish-state(FC)
    snapshot(SSC) MC
    such that FC /= empConfig
    /\ base-state(SSC) finish-state(empConfig)
       control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
               (consume : true))

=>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (consume : true)) .

*** End of Experiment for imc01 ***

*** Experiment for imc02 ***

*** states: 8451  rewrites: 101397 in 706ms cpu
*** (707ms real) (143426 rewrites/second)
search in EXPERIMENT : imc02 =>* MC such that false .

*** Solution 874 (state 8450)
*** states: 8451  rewrites: 109842 in 974ms cpu
*** (59634ms real) (112729 rewrites/second)
search in EXPERIMENT :
    imc02 =>* start-state(SC) finish-state(FC)
    snapshot(SSC) MC
    such that FC /= empConfig .

*** Solution 874 (state 8450)
*** states: 8451  rewrites: 139329 in 1552ms cpu
*** (44910ms real) (89716 rewrites/second)
search in EXPERIMENT :
    imc02 =>* start-state(SC) finish-state(FC)

```

```

snapshot(SSC) MC
such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
              (prog[p(2)]: notYet) (consume : true))
    =>
    base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
              (prog[p(2)]: notYet) (consume : true)) .

*** Solution 874 (state 8450)
*** states: 8451 rewrites: 121493 in 1255ms cpu
*** (40742ms real) (96790 rewrites/second)
search in EXPERIMENT :
  imc02 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
    /\ base-state(SSC) finish-state(empConfig)
      control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
                (prog[p(2)]: notYet) (consume : true))
      =>
      base-state(FC) finish-state(empConfig)
      control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
                (prog[p(2)]: notYet) (consume : true)) .

*** End of Experiment for imc02 ***

*** Experiment for imc03 ***

*** states: 60695
search in EXPERIMENT : imc03 =>* MC such that false .

*** Solution 9315 (state 60694)
*** states: 60695 rewrites: 553158 in 8221ms cpu
*** (462784ms real) (67285 rewrites/second)
search in EXPERIMENT :
  imc03 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig .

*** Solution 9315 (state 60694)
*** states: 60695 rewrites: 1814286 in 24295ms cpu

```



```

*** (409544ms real) (74674 rewrites/second)
search in EXPERIMENT :
  imc03 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (consume : true))
    =>
    base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (consume : true)) .

```

```

*** Solution 9315 (state 60694)
*** states: 60695 rewrites: 1725059 in 23364ms cpu
*** (612072ms real) (73833 rewrites/second)
search in EXPERIMENT :
  imc03 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (consume : true))
    =>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (consume : true)) .

```

*** End of Experiment for imc03 ***

*** Experiment for imc04 ***

```

*** states: 269508
search in EXPERIMENT : imc04 =>* MC such that false .

```

```

*** Solution 20851 (state 269506)
*** states: 269507 rewrites: 3825380 in 43031ms cpu
*** (2074578ms real) (88898 rewrites/second)
search in EXPERIMENT :
  imc04 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig .

```

```

*** Solution 20851 (state 269506)
*** states: 269507 rewrites: 7333415 in 102291ms cpu
*** (2741392ms real) (71691 rewrites/second)
search in EXPERIMENT :
  imc04 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (consume : false))

=>
  base-state(SSC) finish-state(empConfig)
  control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
          (prog[p(2)]: notYet) (consume : false)) .

```

```

*** Solution 20851 (state 269506)
*** states: 269507 rewrites: 6935913 in 99523ms cpu
*** (2715449ms real) (69690 rewrites/second)
search in EXPERIMENT :
  imc04 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (consume : false))

=>
  base-state(FC) finish-state(empConfig)
  control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
          (prog[p(2)]: notYet) (consume : false)) .

```

*** End of Experiment for imc04 ***

*** Experiment for imc05 ***

```

*** states: 471295
search in EXPERIMENT : imc05 =>* MC such that false .

```

```

*** Solution 33344 (state 471293)
*** states: 471294 rewrites: 6874881 in 86366ms cpu
*** (5815273ms real) (79601 rewrites/second)
search in EXPERIMENT :

```

```

imc05 =>* start-state(SC) finish-state(FC)
snapshot(SSC) MC
such that FC /= empConfig .

*** Solution 33344 (state 471293)
*** states: 471294 rewrites: 14393284 in 225842ms cpu
*** (3378231ms real) (63731 rewrites/second)
search in EXPERIMENT :
  imc05 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true))
  =>
    base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true)) .

*** Solution 33344 (state 471293)
*** states: 471294 rewrites: 12802410 in 214543ms cpu
*** (3572482ms real) (59672 rewrites/second)
search in EXPERIMENT :
  imc05 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true))
  =>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true)) .

*** End of Experiment for imc05 ***

*** Experiment for imc06 ***

*** states: 810938 rewrites: 11021149 in
*** 5357391096ms cpu
*** (434192ms real) (2 rewrites/second)
*** search in EXPERIMENT : imc06 =>* MC such that false .

```

```

*** Solution 81740 (state 810937)
*** states: 810938 rewrites: 11832087 in
*** 5357391621ms cpu
*** (1726642ms real) (2 rewrites/second)
search in EXPERIMENT :
  imc06 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig .

*** Solution 81740 (state 810937)
*** states: 810938 rewrites: 31965889 in
*** 5357390769ms cpu
*** (2014779ms real) (5 rewrites/second)
search in EXPERIMENT :
  imc06 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true))
  =>
    base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true)) .

*** Solution 81740 (state 810937)
*** states: 810938 rewrites: 31087639 in
*** 5357394898ms cpu
*** (1914408ms real) (5 rewrites/second)
search in EXPERIMENT :
  imc06 =>* start-state(SC) finish-state(FC)
  snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true))
  =>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
             (prog[p(2)]: notYet) (consume : true)) .

*** End of Experiment for imc06 ***

```

*** Experiment for imc07 ***

```
*** maude.intelDarwin(708,0xac6062c0) malloc:
*** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
*** terminate called throwing an exceptionAbort trap: 6
```

search in EXPERIMENT : imc07 =>* MC such that false .

```
*** Solution 137140 (state 6854875)
*** states: 6854876 rewrites: 105922148 in 5213061ms cpu
*** (27530117ms real) (20318 rewrites/second)
```

```
*** maude.intelDarwin(737,0xac6062c0) malloc:
*** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
*** terminate called throwing an exceptionAbort trap: 6
```

search in EXPERIMENT :
imc07 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC /= empConfig .

```
*** Solution 122682 (state 6400916)
*** states: 6400917 rewrites: 440565484 in 38431146ms cpu
*** (54612651ms real) (11463 rewrites/second)
```

```
*** maude.intelDarwin(12040,0xac1522c0) malloc:
*** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
*** terminate called throwing an exceptionAbort trap: 6
```

search in EXPERIMENT :
imc07 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC /= empConfig
 /\ base-state(SC) finish-state(empConfig)
 control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
 (prog[p(2)]: notYet) (consume : true))
=>

```

        base-state(SSC) finish-state(empConfig)
        control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
                (prog[p(2)]: notYet) (consume : true)) .

*** Solution 125725 (state 6485240)
*** states: 6485241 rewrites: 373514223 in 23539706ms cpu
*** (39141309ms real) (15867 rewrites/second)

-----
*** maude.intelDarwin(205,0xac6062c0) malloc:
*** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
*** terminate called throwing an exceptionAbort trap: 6

-----
search in EXPERIMENT :
  imc07 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (consume : true))
    =>
    base-state(FC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (consume : true)) .

***(imc07 SSC FC
*** Solution 123827 (state 6448377)
*** states: 6448378 rewrites: 368379995 in 32136576ms
*** cpu (46709597ms real) (11462 rewrites/second)
***maude.intelDarwin(310,0xac1522c0) malloc:
*** mmap(size=262144) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
terminate called throwing an exceptionAbort trap: 6
)

*** End of Experiment for imc07 ***

*** Experiment for imc08 ***

*** states: 3587681 rewrites: 51560637 in 1914834ms cpu
*** (1915236ms real) (26926 rewrites/second)

```

search in EXPERIMENT : imc08 =>* MC such that false .

*** Solution 190434 (state 3587680)
*** states: 3587681 rewrites: 55148210 in 2047317ms cpu
*** (46673794ms real) (26936 rewrites/second)

search in EXPERIMENT :
imc08 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC != empConfig .

*** Solution 190434 (state 3587680)
*** states: 3587681 rewrites: 118469042 in 5181498ms cpu
*** (49712756ms real) (22863 rewrites/second)

search in EXPERIMENT :
imc08 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC != empConfig
/\ base-state(SC) finish-state(empConfig)
control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet) (consume : true))

=>
base-state(SSC) finish-state(empConfig)
control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet) (consume : true)) .

*** Solution 190434 (state 3587680)
*** states: 3587681 rewrites: 116431923 in 5081011ms cpu
*** (48872905ms real) (22915 rewrites/second)

search in EXPERIMENT :
imc08 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
such that FC != empConfig
/\ base-state(SSC) finish-state(empConfig)
control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet) (consume : true))

=>
base-state(FC) finish-state(empConfig)
control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
(prog[p(2)]: notYet) (consume : true)) .

*** End of Experiment for imc08 ***

```

*** Experiment for imc09 ***

*** states: 579896  rewrites: 24480376 in 192794ms cpu
*** (192858ms real) (126976 rewrites/second)
search in EXPERIMENT : imc09 =>* MC such that false .

*** Solution 2380 (state 579886)
*** states: 579887  rewrites: 25059911 in 194375ms cpu
*** (500287ms real) (128925 rewrites/second)
search in EXPERIMENT :
  imc09 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig .

*** Solution 2380 (state 579886)
*** states: 579887  rewrites: 25128901 in 236470ms cpu
*** (562877ms real) (106266 rewrites/second)
search in EXPERIMENT :
  imc09 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
              (prog[p(2)]: notYet) (prog[p(3)]: notYet)
              (prog[p(4)]: notYet) (consume : true))
  =>
  base-state(SSC) finish-state(empConfig)
  control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (prog[p(3)]: notYet)
            (prog[p(4)]: notYet) (consume : true)) .

*** Solution 2380 (state 579886)
*** states: 579887  rewrites: 25128496 in 267381ms cpu
*** (647594ms real) (93980 rewrites/second)
search in EXPERIMENT :
  imc09 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
              (prog[p(2)]: notYet) (prog[p(3)]: notYet)
              (prog[p(4)]: notYet) (consume : true))
  =>
  base-state(FC) finish-state(empConfig)
  control((prog[p(0)]: notYet) (prog[p(1)]: notYet)

```



```

        (prog[p(2)]: notYet) (prog[p(3)]: notYet)
        (prog[p(4)]: notYet) (consume : true)) .

*** End of Experiment for imc09 ***

*** Experiment for imc10 ***

*** maude.intelDarwin(295,0xac6062c0) malloc:
*** mmap(size=262144) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
*** terminate called throwing an exceptionAbort trap: 6
—
search in EXPERIMENT : imc10 =>* MC such that false .

***
search in EXPERIMENT :
  imc10 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig .

*** Solution 779 (state 5662572)
*** states: 5662573 rewrites: 254245940 in 14463958ms cpu
*** (14540092ms real) (17577 rewrites/second)
search in EXPERIMENT :
  imc10 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
              (prog[p(2)]: notYet) (prog[p(3)]: notYet)
              (prog[p(4)]: notYet) (consume : true))
  =>
  base-state(SSC) finish-state(empConfig)
  control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
            (prog[p(2)]: notYet) (prog[p(3)]: notYet)
            (prog[p(4)]: notYet) (consume : true)) .

***
search in EXPERIMENT :
  imc10 =>* start-state(SC) finish-state(FC) snapshot(SSC) MC
  such that FC /= empConfig
  /\ base-state(SSC) finish-state(empConfig)
    control((prog[p(0)]: notYet) (prog[p(1)]: notYet)

```

```

                (prog[p(2)]: notYet) (prog[p(3)]: notYet)
                (prog[p(4)]: notYet) (consume : true))
=>
base-state(FC) finish-state(empConfig)
control((prog[p(0)]: notYet) (prog[p(1)]: notYet)
        (prog[p(2)]: notYet) (prog[p(3)]: notYet)
        (prog[p(4)]: notYet) (consume : true)) .

*** End of Experiment for imc10 ***

```

Bibliography

- [1] Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed system. ACM TOCS 3, 63-75 (1985).
- [2] Ogata, K., Huyen, T.T.P.: Specification and Model Checking of the Chandy and Lamport Distributed Snapshot Algorithm in Rewriting Logic. 14th ICFEM, LNCS 7635, 87-102 (2012).
- [3] Holzmann, G.J.: The SPIN Model Checker . Primer and Reference Manual. Addison-Wesley (2004).
- [4] 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. 14th CAV, LNCS 2404, Springer, 359-364 (2002).
- [5] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS 4350, Springer.
- [6] Zhang, W., Ogata, K., Zhang, M.: A Consideration on How to Model Check Distributed Snapshot Reachability Property, IEICE Technical Report, Vol. 114, No. 416, ISSN 0913-5685, 49-54 (2015).
- [7] Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988).
- [8] Ajay D. Kshemkalyani, Mukesh Singhal: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2013.
- [9] Andriamiarina, M. B., Méry, D., Singh, N. K.: Revisiting Snapshot Algorithms by Refinement-based Techniques. Computer Science and Information Systems, Vol. 11, No. 1, 251 - 270 (2014)
- [10] Mordechai Ben-Ari. Principles of the Spin Model Checker. Springer, London, 2008.
- [11] E. M. Clarke, O. Grumberg, and D. A. Peled. Model checking. MIT Press, Cambridge, 1999.
- [12] Ogata, K., Futatsugi, K.: Simulation-based Verification for Invariant Properties in the OTS/CafeOBJ Method. Electr. Notes Theor. Comput. Sci. 201: 127-154, 2008.