An Investigation of the Chandy-Lamport Distributed Snapshot Algorithm and its Model Checking

By Wenjie Zhang

A project paper 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

March, 2015

An Investigation of the Chandy-Lamport Distributed Snapshot Algorithm and its Model Checking

By Wenjie Zhang (1310043)

A project paper 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

February, 2015 (Submitted)

Abstract

Concurrent and distributed systems have attained remarkable achievements in the past decades and are being widely used in our real life. Many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. The global state of a distributed system consists of the states of every process and every channel in the system, where the state of a process is characterized by the state of its local memory and depends upon the context, and the state of a channel is characterized by the sequence of messages *"in-transit"*, those that have been sent on that channel, but not yet received by its destination process. Due to the asynchrony of distributed systems, the lack of globally shared memory, global clock and unpredictable message delays in a distributed system make recording such consistent global states non-trivial. Chandy and Lamport have proposed a distributed snapshot algorithm by which a process in a distributed system can determine a consistent global state of the system during its computation.

Since the design of distributed systems is generally complex, with a high possibility that subtle errors will cause erroneous behavior, and also the Chandy-Lamport Distributed Snapshot Algorithm ($\mathcal{CLDSA}$) is a non-trivial distributed algorithm, it deserves to be formally specified and verified with respect to (w.r.t.) some significant properties. Let s_1 be the state when the $\mathcal{CLDSA}$ starts, s_* be the snapshot, and s_2 be the state when the $\mathcal{CLDSA}$ terminates. The $\mathcal{CLDSA}$ should enjoy the property that s_* is always reachable from s_1 and s_2 is always reachable from s_*. The property is called the Distributed Snapshot Reachability ($\mathcal{DSR}$) property.

To formally verify the $\mathcal{DSR}$ property, model checking, which is an automatic verification technique for finite-state concurrent systems, can be used. It refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. The main challenge in model checking is dealing with the *state explosion* problem.

This research aims to investigate the $\mathcal{CLDSA}$, its formal specification in Maude and its model checking with the Maude search command, and to conduct some model checking experiments with several different underlying distributed systems. However, we do not think that the existing study, in which a distributed system superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command, provides the sufficiently good foundation backing up that the $\mathcal{CLDSA}$ is surely model checked w.r.t. the $\mathcal{DSR}$ property, and then the $\mathcal{DSR}$ property encoded in the Maude search command are neither readable nor comprehensible. To make it executable in Maude, moreover, the system superimposed by the $\mathcal{CLDSA}$ has been specified in a very concrete way, in which the state of each process only depends on the tokens owned by the process itself.

To complement the existing study, we describe how to express the $\mathcal{DSR}$ property in a more abstract way in this report. Our way to express the $\mathcal{DSR}$ property has been affected by the Chandy-Misra's.

Contents

Acknowledgements

First and foremost, I would like to express my special appreciation and thanks to my supervisor, Professor Kazuhiro Ogata, for his kindly guidance, encouragement, and unlimited support throughout the duration of my master research. Without his support, this research could not have been completed. His advice on both my research and my career are definitely priceless, which will influence the rest of my life.

I would also like to thank my committee members, Professor Kunihiko Hiraishi, and Associate Professor Toshiaki Aoki for serving as my committee members, and also for giving me lots of precious comments and suggestions.

I would like to especially thank Min Zhang for his advice on my research, and also for his recommend to Ogata lab. The quality of this research was significantly improved according to his advice.

Last but not least, a special thanks to my family for their support and encouragement throughout my study and my life, and thanks to all of my friends who have supported and helped me.

Chapter 1

Introduction

1.1 Overview

Concurrent and distributed systems have attained remarkable achievements in the past decades and are being widely used in our real life. Telecommunication networks such as wireless sensor networks, network applications such as world wide web and banking systems, real-time process control systems such as aircraft control systems and industrial control systems, and cloud computing systems are all distributed systems. Since the design of such distributed systems is generally complex, with a high possibility that subtle errors will cause erroneous behavior, such systems may crash, and we need to recover them if that is the case.

Many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared."

The global state of a distributed system consists of the states of every process and every channel in the system, where the state of a process is characterized by the state of its local memory and depends upon the context, and the state of a channel is characterized by the sequence of messages *"in-transit"*, those that have been sent on that channel, but not yet received by its destination process.

For a global state to be meaningful, the states of all the components of the underlying distributed system ($\mathcal{UDS}$) must be recorded at exactly the same instant. This will be possible if the local clocks at processes were perfectly synchronized or there was a global system clock that could be instantaneously read by the processes. However, given the fact that distributed systems are asynchronous and processes in the system do not share common clocks or memory, each process cannot record its local state at exactly the same time, namely that such global states can never be instantaneously done, and it leaves open the possibility of inconsistent global states. Moreover, the variability in message delays could lead to these separate processes constructing different global states for the same computation. In a word, due to the asynchrony of distributed systems,

the lack of globally shared memory, global clock and unpredictable message delays make recording such consistent global states non-trivial. The global states we obtained may be inconsistent if we record the state of each component (process or channel) in the system whenever we want.

However, it turns out that even if the state of all the components in a $\mathcal{UDS}$ has not been recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent. Basic idea is that an effect should not be present without its cause. A message cannot be received if it was not sent; that is, the state should not violate causality. Such states are called *consistent global states* and are meaningful global states. Inconsistent global states are not meaningful in sense that a $\mathcal{UDS}$ can never be in an inconsistent global state.

Therefore, it is necessary to apply some algorithms when we want to obtain consistent global states of a distributed system. One of those algorithms is known as the Chandy-Lamport Distributed Snapshot Algorithm ($\mathcal{CLDSA}$) [CL85], which was proposed by Chandy and Lamport in 1985.

The $\mathcal{CLDSA}$ can be used to determine consistent global states of a distributed system during its computation. Since it is very important and also non-trivial, it deserves to be formally specified and verified with respect to (w.r.t.) some significant properties. Let s_1 (called the start state) be the state when the $\mathcal{CLDSA}$ starts (a distributed snapshot starts being taken), s_* be the snapshot, and s_2 (called the finish state) be the state when the $\mathcal{CLDSA}$ terminates (the snapshot completes being taken). The $\mathcal{CLDSA}$ should enjoy the property that s_* is always reachable from s_1 and s_2 is always reachable from s_*. The property is called the Distributed Snapshot Reachability ($\mathcal{DSR}$) property, which guarantees the $\mathcal{CLDSA}$ takes consistent global states of a distributed system.

To formally verify the $\mathcal{DSR}$ property, model checking, which is an automatic verification technique for finite-state concurrent systems, can be used. It has been practically used in hardware industry, while many studies have been actively conducted on model checking so that model checking can be effectively and practically used for software.

The process of model checking comprises three main tasks: modeling, specification and verification.

Modeling is to convert a system that is to be reasoned about into a formalism accepted by a model checking tool. State machine can be used to model distributed systems. It consists of a set of states and a set of state transitions (i.e., a binary relation over the states). In such a model, the system is in one of the possible states, and the transition relation describes how the system moves from one state to another.

Specification is to state the properties that the system must satisfy before verification. These are written in a specification language, usually defined in a logic-based formalism. Completeness is one of the important issues in specification. Model checking provides means for checking that a model of the system satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

Verification is to check the validity of the properties that have been stated previously. Ideally it is completely automatic. However, in practice it often involves human assistance

such as the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred.

Model checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. The main challenge in model checking is dealing with the *state explosion* problem caused by the fact that the state machine represents the *state space* of the system under investigation, and thus it is of size *exponential* in the size of the system description. Therefore, even for systems of relatively modest size, it is often impossible to compute their state machines.

There have been several major advances in addressing the *state explosion* problem. One of the first major advances was *symbolic model checking with binary decision diagrams* (BDDs). In this approach, a set of states is represented by a BDD instead of by listing each state individually. The BDD representation is often exponentially smaller in practice. Model checking with BDDs is performed using a *fixed point* algorithm. Another major advance is the *partial order reduction*, which exploits independence of actions in a system with asynchronous composition of processes. A third major advance is *counterexample-guided abstraction refinement*, which adaptively tries to find an appropriate level refinement, precise enough to verify the property of interest yet not burdened with irrelevant detail that slows down verification. Finally, *bounded model checking* exploits fast Boolean satisfiability (SAT) solvers to search for counterexamples of bounded length.

Maude [CDE+07], an algebraic specification language originated from OBJ family, is based on rewriting logic that includes as a sub-logic membership equational logic (an extension of order-sorted equational logic). Maude supports rewriting modulo equational theories such as associativity (`assoc`), commutativity (`comm`), and identity (`id`). Basic units of Maude specifications are modules such as `BOOL` and `NAT` used for boolean values and natural numbers. State machines (or transition systems) are specified in rewriting logic, and their specifications are called system specifications. Data used in state machines are specified in membership equational logic. States of state machines are expressed as tuples and associative-commutative collections (called soups), and state transitions are described in rewrite rules.

The $\mathcal{CLDSA}$ is a non-trivial distributed algorithm that deserves to be formally specified and verified w.r.t. the $\mathcal{DSR}$ property.

As far as we have investigated, to formalize the $\mathcal{DSR}$ property, we have to consider two kinds of states, (1) the states of a $\mathcal{UDS}$, and (2) the states of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. In existing temporal logics such as LTL and CTL, only one kind of states are considered when they are used to formalize system properties. Thus, it is not straightforward to express the $\mathcal{DSR}$ property in LTL and CTL.

Moreover, there is an existing study [OH12] in which a distributed system superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command. We do not, however, think that the existing study provides the sufficiently good foundation backing up that the $\mathcal{CLDSA}$ is surely model checked w.r.t. the $\mathcal{DSR}$ property, because the authors did not discuss whether the property is faithfully expressed or not. And then the $\mathcal{DSR}$ property en-

coded in the Maude search command are neither readable nor comprehensible. To make it executable in Maude, moreover, the system superimposed by the $\mathcal{CLDSA}$ has been specified in a very concrete way, in which the state of each process only depends on the tokens owned by the process itself. We do think that it is necessary to make sure that the property is faithfully expressed to claim that the property is model checked for the $\mathcal{CLDSA}$.

This research aims to investigate the $\mathcal{CLDSA}$, its formal specification in Maude and its model checking with the Maude search command, and to conduct some model checking experiments with several different underlying distributed systems. Moreover, to complement the existing study [OH12], we have considered how to surely model check the $\mathcal{DSR}$ property. To this end, we have already found a way to faithfully express the $\mathcal{DSR}$ property. Our way to express the property relies on two state machines, although the two state machines are closely related. And the property used in the existing study relies on only one state machine. Our way to express the $\mathcal{DSR}$ property has been affected by the Chandy-Misra's [CM88].

1.2 Scope of this Work

Concurrent and distributed systems are no longer rare, but are widely used in applications from television sets to train signaling and workflow systems. The order in which events occur in the execution of such systems is unpredictable and only restricted by synchronization of individual processes. As a result, the design of distributed systems is generally complex, with a high probability that subtle errors will cause erroneous behavior. Without formally verifying the properties the system should enjoy, it is particularly difficult for the developers of such systems to be confident about the correctness of their designs.

Many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. The $\mathcal{CLDSA}$ can be used to determine consistent global states of a distributed system during its computation. Since it is very important and also non-trivial, it deserves to be formally specified and verified with respect to the $\mathcal{DSR}$ property.

Our main goal is to investigate the $\mathcal{CLDSA}$, its formal specification in Maude and its model checking with the Maude search command, and to conduct some model checking experiments with several different $\mathcal{UDS}$s. However, we do not think that the existing study [OH12], in which a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command, provides the sufficiently good foundation backing up that the $\mathcal{CLDSA}$ is surely model checked w.r.t. the $\mathcal{DSR}$ property, and then the property encoded in the Maude search command are neither readable nor comprehensible. To make it executable in Maude, moreover, the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ has been specified in a very concrete way, in which the state of each process only depends on the tokens owned by the process.

To complement the existing study [OH12], we describe how to express the $\mathcal{DSR}$ property in a more abstract way in this report. Our way to express the $\mathcal{DSR}$ property has been affected by the Chandy-Misra's [CM88].

1.3 Contributions

What we have done are the following.

- Existing Studies:
  - Learnt some basic technical knowledge such as distributed systems, state machine, model checking and Maude specification language;
  - Learnt the $\mathcal{CLDSA}$ and the $\mathcal{DSR}$ property;
  - Learnt an existing study [OH12], in which a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command.

- Original Works:
  - Realized the expression of the $\mathcal{DSR}$ property in the existing study [OH12] does not respect the property (written in English) in the original paper [CL85];
  - Given the definition of the function $\mathcal{CL}$;
  - Found a way [ZOZ15] to faithfully express the $\mathcal{DSR}$ property.

1.4 Organization of the Report

The remainder of this report is organized as follows:

- **Chapter 2** introduces some technical background such as $\mathcal{UDS}$, state machine, model checking and Maude specification language.
- **Chapter 3** presents the $\mathcal{CLDSA}$ and the $\mathcal{DSR}$ property.
- **Chapter 4** explains an existing study [OH12], in which a distributed system superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command. And some model checking experiments will be conducted.
- **Chapter 5** shows a consideration on how to model check the $\mathcal{DSR}$ property, which includes how to model a $\mathcal{UDS}$ and the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ as a state machine respectively, the definition of the function $\mathcal{CL}$, and a way to express the $\mathcal{DSR}$ property.
- **Chapter 6** concludes this report and discusses possible issues of future work.
- **Appendix A** gives the specification of the $\mathcal{CLDSA}$ in Maude.
- **Appendix B** shows the verification of the $\mathcal{DSR}$ property by conducting some model checking experiments with several different $\mathcal{UDS}$s.

Chapter 2

Technical Background

2.1 An Underlying Distributed System ($\mathcal{UDS}$)

2.1.1 Definition

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. It can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features [KS08]:

- **No common physical clock** This is an important assumption because it introduces the terminology "distribution" in the system and gives rise to the inherent asynchrony among the processes.
- **No shared memory** This is a key feature that requires the processors in the system to communicate with each other by message-passing. And this feature implies the absence of the common physical clock.
- **Geographical seperation** The geographically wider apart that the processors are, the more representative is the system of a distributed system.
- **Autonomy and heterogeneity** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.

Consider two processes P and Q in a network of processes. If P computes $f(x) = x^2$ for a given set of values of x, and Q multiplies a set of numbers by π, then we hesitate to call it a distributed system, since there is no interaction between P and Q. However, if P and Q cooperate with one another to compute the areas of a set of circles of radius x, then the system of processes (P and Q) is an example of a meaningful distributed system [Gho06].

2.1.2 Model of a $\mathcal{UDS}$

As described in [CL85], a $\mathcal{UDS}$ consists of a finite set of processes and a finite set of channels, which can be described by a labeled, directed graph in which the vertices represent the processes and the directed edges represent the channels. Figure 2.1 is an example. It shows a distributed system that consists of three processes (p, q, and r) and four channels (c1, c2, c3 and c4), in which there are two channels (c1 and c2) from process p to process q.

In addition, channels are assumed to have infinite buffers, to be error-free and FIFO (messages are delivered in the order sent). The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer. The delay experienced by a message is arbitrary but finite.

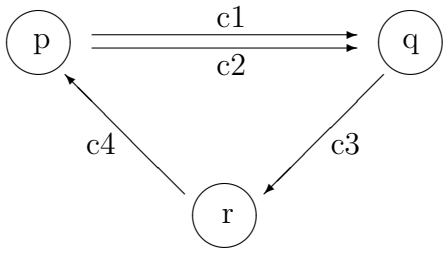

Figure 2.1 A distributed system with processes p, q, r and channels c1, c2, c3 and c4

2.1.3 Global States of a $\mathcal{UDS}$

The global state of a $\mathcal{UDS}$ consists of the states of every process and every channel in the system, where the state of a process is characterized by the state of its local memory and depends upon the context, and the state of a channel is characterized by the sequence of messages *"in-transit"*, those that have been sent on that channel, but not yet received by its destination process.

For a global state to be meaningful, the states of all the components of the $\mathcal{UDS}$ must be recorded at exactly the same instant. This will be possible if the local clocks at processes were perfectly synchronized or there was a global system clock that could be instantaneously read by the processes. However, given the fact that distributed systems are asynchronous and processes in the system do not share common clocks or memory, each process cannot record its local state at exactly the same time, namely that such global states can never be instantaneously done, and it leaves open the possibility of inconsistent global states. Moreover, the variability in message delays could lead to these separate processes constructing different global states for the same computation. In a word, the global states we obtained may be inconsistent if we record the state of each component (process or channel) in the system whenever we want.

However, it turns out that even if the state of all the components in a $\mathcal{UDS}$ has not been recorded at the same instant, such a state will be meaningful provided every message

that is recorded as received is also recorded as sent. Basic idea is that an effect should not be present without its cause. A message cannot be received if it was not sent; that is, the state should not violate causality. Such states are called *consistent global states* and are meaningful global states. Inconsistent global states are not meaningful in sense that a $\mathcal{UDS}$ can never be in an inconsistent global state.

As we all know, many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. Now, the challenge here is how to obtain consistent global states of a $\mathcal{UDS}$. Many snapshot algorithms can be used to determine consistent global states of a $\mathcal{UDS}$ during its computation. One of them is known as the $\mathcal{CLDSA}$ [CL85], which was proposed by Chandy and Lamport in 1985. In this report, we will focus on this algorithm, which will be described in detail in the Chapter 3.

2.2 State Machine

State machine can be used to model distributed systems. It consists of a set of states and a set of state transitions (i.e., a binary relation over the states). In such a model, the system is in one of the possible states, and the transition relation describes how the system moves from one state to another. The definition is the following.

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

1. *a set of states* S*;*
2. *a set of initial states* $I \subseteq S$*;*
3. *a binary relation* $T \subseteq S \times S$*.*

   For each $s, s' \in S$*, if* $(s, s') \in T$*, it denotes that there is a transition from* s *to* s'*.* s' *is a successor state of* s*.*

Definition 2 (*Path*) *A path* π *in* $M \triangleq \langle S, I, T\rangle$ *from a state* s_0 *is an infinite sequence of states* $\pi \triangleq (s_0, s_1, s_2, \dots)$*, where* $\forall i \geq 0, (s_i, s_{i+1}) \in T$*. We say that* $\pi \triangleq (s_0, s_1, s_2, \dots)$ *is rooted at state* s_0*.*

Definition 3 (π's i-th *state*) π_i *denotes* π*'s* i*-th state (i.e.,* s_i*).*

Definition 4 (π's i-th *suffix*) π^i *denotes* π*'s* i*-th suffix (i.e.,* $(s_i, s_{i+1}, s_{i+2}, \dots)$*).*

Definition 5 (*Set of all Paths*) Π *denotes the set of all paths w.r.t.* M*.*

Definition 6 (*Computation*) *A computation is a path starting with an initial state.*

Equational theories and rewrite theories are two main algebraic-based approaches to formalizing state machines [GL05, Mes96]. They are used to verify or falsify computer systems' properties in different techniques, e.g., the former are used for theorem proving [HH82, GL05], and the latter for (bounded) model checking [CDE+07, BM11]. In this report, we focus on the latter one.

2.3 Model Checking

Model Checking is an automatic verification technique for finite state concurrent systems. It has been successfully used to verify system designs and properties in a variety of application domains, ranging from hardware and software systems to biological systems. For extensive overviews of model checking, please refer to [CGP99, CS01].

A model checker requires a model provided in some formal description language and a semantic property that such model is expected to satisfy. The model checker then automatically checks the validity of the specified property in the model semantics. If the property is found to not hold, a counterexample is generated which shows how the property can be falsified.

The automatic generation of counterexamples is one of model checking's powerful features for system fault detection. Counterexamples are meant to help engineers in the tasks of identifying the cause of a property violation and correcting the model. However, these tasks are far from trivial with little automated support. Even in relatively small models such tasks can be very complex since (i) counterexamples are expressed in terms of the model semantics rather than the modeling language, (ii) counterexamples show the symptom and not the cause of the violation and (iii) manual modifications to the model may fail to resolve the problem and even introduce violations to other desirable properties.

Model checking has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols. The most important is that the procedure is completely automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The procedure is also quite fast and often produces an answer in a matter of minutes. Since partial specifications can be checked, it is unnecessary to specify the circuit completely before useful information about its correctness can be obtained. Finally, the logics used for specifications can directly express many of the properties that are needed for reasoning about concurrent systems.

The process of model checking comprises three main tasks: modeling, specification and verification [CGP99].

- **Modeling** is to convert a system that is to be reasoned about into a formalism accepted by a model checking tool.

- **Specification** is to state the properties that the system must satisfy before verification. These are written in a specification language, usually defined in a logic-based formalism.

  Completeness is one of the important issues in specification. Model checking provides means for checking that a model of the system satisfies a given specification,

but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

- **Verification** is to check the validity of the properties that have been stated previously. Ideally it is completely automatic. However, in practice it often involves human assistance such as the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred.

The model checking problem involves the construction of an abstract model M, in the form of variations on finite state automata, and the construction of specification formulas ϕ, in the form of variations on temporal logic [BBF+10]. The model checking verification problem involves establishing that the model semantically entails the specification $M \models \phi$. Then we can define the model checking problem [CKNZ12] as follows.

Definition 7 (*Model Checking Problem*) *Let M be a state-transition graph and let ϕ be a temporal logic formula. The model checking problem is to find all the states $s \in S$ such that $M, s \models \phi$.*

The verification algorithm used in the model checking involves exploring the set of reachable states of the model to ensure that the formula ϕ holds. If ϕ is an invariant assertion, the model checking approach explores the entire state space to ensure that the formula holds in all states. In order to guarantee termination, such approach requires that the set of reachable states to be finite. Furthermore, verification by model checking has gained popularity in industry because the verification procedure can be fully automated and counterexamples are automatically generated if the property being verified does not hold. Since model checkers rely on exhaustive state space enumeration to establish whether a property holds or does not hold, it can put immediate limits on the state space problem that can be explored. This problem, known as the *state explosion* problem [CGJ+01], is an often cited drawback of verification by model checking.

2.3.1 Pros and Cons

Model checking is a very powerful framework for verifying specifications of finite-state systems. One of the main advantages of model checking is that it is fully automated. No expert is required in order to check whether a given finite-state model conforms to a given set of system specifications. Model checking also works with partial specifications, which are often troublesome for techniques based on theorem proving. When a property specification does not hold, a model checker can provide a counterexample (an initial state and a set of transitions) that reflects an actual execution leading to an error state. This is the reason why tools based on model checking are very popular for debugging.

One aspect that can be viewed as negative is that model checkers do not provide correctness proofs. Another negative aspect is that model-checking techniques can be directly applied only to finite-state systems. An infinite-state system can by abstracted

into a finite model; however, this leads to a loss of precision. Perhaps the most important issue in model checking is the *state explosion* problem. It is apparent from the complexity of the CTL model checking algorithm that its practical usefulness critically depends on the size of the state space. Basically, if number of states grows too large, so does the complexity of the verification procedure, possibly making the technique unusable. In the next Section we focus on the *state explosion* problem and on several possible methods to combat it.

2.3.2 State Explosion Problem

The main practical problem in model checking is the so-called **state explosion problem** caused by the fact that the state machine represents the *state space* of the system under investigation, and thus it is of size *exponential* in the size of the system description. Therefore, even for systems of relatively modest size, it is often impossible to compute their state machines.

The number of states of a model can be enormous. For example, consider a system composed by n processes, each having m states. Then, the asynchronous composition of these processes may have m^n states. Similarly, in a n-bit counter, the number of states of the counter is exponential in the number of bits, i.e., 2^n. In model checking we refer to this problem as the *state explosion* problem. All model checkers suffer from it. Using arguments from complexity theory, it can be shown that, in the worst case, this problem is inevitable. However, researchers have developed many techniques that address the *state explosion* problem. These techniques are frequently used in industrial applications of model checking. In this section, we will concentrate on key advances that make model checking a practical technique in both research and industry.

There have been several major advances in addressing the *state explosion* problem. One of the first major advances was *symbolic model checking with binary decision diagrams* (BDDs). In this approach, a set of states is represented by a BDD instead of by listing each state individually. The BDD representation is often exponentially smaller in practice. Model checking with BDDs is performed using a *fixed point* algorithm. Another major advance is the *partial order reduction*, which exploits independence of actions in a system with asynchronous composition of processes. A third major advance is *counterexample-guided abstraction refinement*, which adaptively tries to find an appropriate level refinement, precise enough to verify the property of interest yet not burdened with irrelevant detail that slows down verification. Finally, *bounded model checking* exploits fast Boolean satisfiability (SAT) solvers to search for counterexamples of bounded length.

2.4 Maude Specification Language

Maude [CDE+07], an algebraic specification language originated from OBJ family, is based on rewriting logic that includes as a sub-logic membership equational logic (an extension of order-sorted equational logic). Maude supports rewriting modulo equational theories such as associativity (`assoc`), commutativity (`comm`), and identity (`id`). Basic units of Maude specifications are modules such as `BOOL` and `NAT` used for boolean values and natural numbers. Rewrite theory described in [OH12] is as follows:

State machines (or transition systems) are specified in rewriting logic, and their specifications are called system specifications. Data used in state machines are specified in membership equational logic. States of state machines are expressed as tuples and associative-commutative collections (called soups), and state transitions are described in rewrite rules.

In Maude, *functional modules* are equational theories in membership equational logic satisfying some additional requirements. Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found.

A *functional module* is declared with keywords `fmod` ... `endfm` and contains a set of declarations consisting of:

- importations of previously defined modules (e.g. `protecting`, `including`)
- declarations of sorts (`sort` s `.` or `sorts` s s' `.`)
- declarations of subsort (`subsort` s `<` s' `.`)
- declarations of function symbols (`op` f : $s_1 \ldots s_n$ `->` s `.`)
- declarations of variables (`vars` v v' : s `.`)
- unconditional equations (`eq` $t = t'$ `.`), and
- conditional equations (`ceq` $t = t'$ `if` *cond* `.`)

For example, we declare a functional module of natural number `NAT` as follows:

```
fmod NAT is
    protecting BOOL .
    sorts Zero NzNat Nat .
    subsort Zero NzNat < Nat .
    op 0 : -> Zero [ctor] .
    op s_ : Nat -> NzNat [ctor] .
endfm
```

where `0` is a constant for zero in natural number, and `s_` means the successor of the input value `Nat`. For instance, `s 0` means the successor of `0`, namely "1" in natural number. And `s s s 0` returns natural number "3" analogically.

Some built-in modules are also 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`. Precisely, there are three sorts for natural numbers `Zero`, `NzNat`, and `Nat` that are for zero, non-zero natural numbers, and natural numbers that may be zero or non-zero. Sort `Nat` is the super-sort of `Zero` and `NzNat`, namely that sort `Zero` and `NzNat` are sub-sort of sort `Nat`.

Besides *functional modules*, Maude also has *functional theories*, which can be declared with keywords `fth` ... `endfth`. It can also do the same thing as what *functional modules* do such as declaring sorts, operators, and variables, and can import other theories or modules. Theories have a *loose* semantics, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model.

However, there is a full Maude which is the extension of Maude. The syntax of full Maude is similar to the one of Maude but some are different. For example, the parenthesis are needed to cover *functional modules* or *functional theories*, i.e., `(fmod` ... `endfm)` or `(fth` ... `endfth)`.

2.4.1 Specifying in Maude

Now let us consider a simple system as an example. The system consists of two processes p and q and one channel c which is an unbounded queue (namely FIFO channel with infinite buffer) from p to q. Each process has a set of natural numbers, which is regarded as the state of the process.

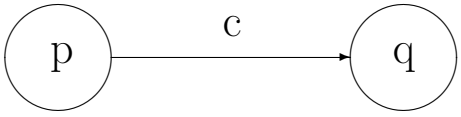

Figure 2.2 A simple system with processes p, q and channel c

Initially, the set of p is $\{0, 1, 2\}$, the set of q is empty, and the channel is also empty. p arbitrarily chooses and deletes one natural number x from its set, and puts x into the channel, which is referred to as p's action. If the channel is not empty and q's set does not contain 0, q gets the top y from the channel and adds y to its set, which is referred to as q's action. Let us specify this system, precisely a state machine modeling this system, in Maude.

A set of natural numbers is expressed as a soup of natural numbers. The corresponding sort is `NSoup` that is declared as a super-sort of `Nat`, which means that a natural number itself is also the singleton. The empty set is denoted as `noNat`, and the soup of n natural numbers x_1, x_2, ..., x_n as $x_1\ x_2\ \ldots x_n$ whose constructor is called the juxtaposition operator or the empty syntax. `noNat` is declared as an identity of the constructor.

Let N, N1 and N2 be Maude variables of sort Nat, NS be a Maude variable of sort NSoup, C be a Maude variable of sort Chan in the rest of this section. The corresponding module is described as follows:

```
fmod NSOUP is
  pr NAT .
  pr BOOL .
  sort NSoup .
  subsort Nat < NSoup .

  op noNat : -> NSoup [ctor] .
  op __ : Nat Nat -> NSoup [ctor assoc comm id: noNat] .
  op _in_ : Nat NSoup -> Bool .

  var N : Nat .
  vars NS : NSoup .

  eq N in N NS = true .
  eq N in NS = false [owise] .
endfm
```

where in takes a natural number N and a soup of natural numbers NS, and returns the boolean value true if N is in the soup NS, and false otherwise.

The empty channel is denoted as empChan, and the non-empty channel that consists of n natural numbers x_1, x_2, ..., x_n as x_1 | x_2 | $\cdots$ | x_n | empChan. The corresponding sort is Chan. And the module can be described as follows:

```
fmod CHAN is
  pr NSOUP .
  sorts EmpChan NeChan Chan .
  subsorts EmpChan NeChan < Chan .

  op empChan : -> EmpChan [ctor] .
  op _|_ : Nat Chan -> NeChan [ctor] .
  op put : Chan Nat -> NeChan .

  vars N1 N2 : Nat .
  var C : Chan .

  eq put(empChan,N2) = N2 | empChan .
  eq put(N1 | C,N2) = N1 | put(C,N2) .
endfm
```

where put takes a channel C and a natural number N and returns the channel obtained by putting N into C at the end.

We can use *name-value* pairs (called observable components) to express the states of processes and channels, where *name* may have parameters.

For process state, we use parameter `Pid` to identify which process we are referring to. States of processes p and q are expressed as `p-state[p] : ns`, `p-state[q] : ms`, respectively, where `ns` and `ms` are soups of natural numbers. For channel state, we use parameter `Cid` to identify which channel we are referring to. State of channel c is expressed as `c-state[c] : qn`, where `qn` is a channel (queue) of natural numbers. `p-state[p] : ns`, `p-state[q] : ms`, and `c-state[c] : qn` are called observable components, and the corresponding sort is `OCom`.

```
fmod ID is
  sorts Pid Cid Id .
  subsorts Pid Cid < Id .

  ops p q : -> Pid [ctor] .
  op c : -> Cid [ctor] .
endfm

fmod OCOM is
  pr ID .
  pr CHAN .
  sort OCom .

  op p-state[_]:_ : Pid NSoup -> OCom [ctor] .
  op c-state[_]:_ : Cid Chan -> OCom [ctor] .
endfm
```

A state of the system is expressed as a soup (called a configuration) of those observable components, which is expressed as `(p-state[p] : ns)(p-state[q] : ms)(c-state[c] : qn)`. The corresponding sort is `Config` that is a super-sort of `OCom`.

```
fmod CONFIG is
  pr OCOM .
  sort Config .
  subsort OCom < Config .

  op empConfig : -> Config [ctor] .
  op __ : Config Config -> Config [ctor assoc comm id: empConfig] .
endfm
```

Let `ic` be the initial configuration, as we have mentioned at the beginning, initially, the set of p is {0, 1, 2}, the set of q is empty, and the channel is also empty. So we can describe the initial state (configuration) of the system as the following:
`(p-state[p] : (0 1 2)) (p-state[q] : noNat) ((c-state[c] : empChan))`.

In this system, there are two actions `snd` and `rec`. p arbitrarily chooses and deletes one natural number x from its set, and puts x into the channel, which is referred to as p's action `snd`. If the channel is not empty and q's set does not contain 0, q gets the top y from the channel and adds y to its set, which is referred to as q's action `rec`.

p's action is described in the following rewrite rule:

```
rl [snd] :
  (p-state[p] : (N NS)) (c-state[c] : C)
  =>
  (p-state[p] : NS) (c-state[c] : put(C, N)) .
```

where `snd` is the label of the rewrite rule, and `put` takes a channel `C` and a natural number `N` and returns the channel obtained by putting `N` into `C` at the end. If a given term contains an instance of `(p-state[p] : (N NS)) (c-state[c] : C)`, the instance is replaced with the corresponding instance of `(p-state[p] : NS) (c-state[c] : put(C, N))`.

q's action is described in the following rewrite rule:

```
crl [rec] :
  (c-state[c] : (N | C)) (p-state[q] : NS)
  =>
  (c-state[c] : C) (p-state[q] : (N NS))
if not(0 in NS) .
```

This rewrite rule is conditional. The condition `not(0 in NS)` means that $0 \notin$ NS. The rule can be applied if the condition holds.

2.4.2 The Search Command

The Maude system is equipped with model checking facilities: the search command and the LTL model checker. In my research, the model checking invariants through search is used.

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:

$$\textbf{search in } \boldsymbol{M} \textbf{ : } \boldsymbol{s} \Rightarrow^* \boldsymbol{p} \textbf{ such that } \boldsymbol{c} \textbf{ .}$$

where M is a module in which the specification of the state machine concerned is described or available. A rewrite expression $t \Rightarrow t'$ can be used in the optional condition c. This checks if t' is reachable from t by zero or more rewrite steps with rewrite rules. This is the essence of model checking the $\mathcal{DSR}$ property, which will be described in the Chapter 3.

The following search finds all states (configurations) such that they are reachable from `ic` and the q's set contains only 2:

```
search in EXPERIMENT : ic =>* (p-state[q] : 2) CF .
```

where `EXPERIMENT` is the module in which the specification of the system we have been discussing is available. The search finds 5 solutions as follows:

```
Solution 1 (state 12)
states: 13  rewrites: 37 in 0ms cpu (2ms real) (89805 rewrites/second)
CF --> (p-state[p]: 0 1) c-state[c]: empChan

Solution 2 (state 22)
states: 23  rewrites: 85 in 0ms cpu (2ms real) (141196 rewrites/second)
CF --> (p-state[p]: 1) c-state[c]: 0 | empChan

Solution 3 (state 24)
states: 25  rewrites: 93 in 0ms cpu (2ms real) (142857 rewrites/second)
CF --> (p-state[p]: 0) c-state[c]: 1 | empChan

Solution 4 (state 31)
states: 32  rewrites: 143 in 0ms cpu (3ms real) (175030 rewrites/second)
CF --> (p-state[p]: noNat) c-state[c]: 0 | 1 | empChan

Solution 5 (state 33)
states: 34  rewrites: 154 in 0ms cpu (3ms real) (176000 rewrites/second)
CF --> (p-state[p]: noNat) c-state[c]: 1 | 0 | empChan

No more solutions.
states: 38  rewrites: 199 in 1ms cpu (3ms real) (193016 rewrites/second)
```

The following search finds all states (configurations) such that they are reachable from `ic`, the q's set contains only 2, and `(p-state[p] : noNat) (c-state[c] : empChan) (p-state[q] : (0 1 2))` is reachable from them:

```
search in EXPERIMENT : ic =>* (p-state[q] : 2) CF
  such that (p-state[q] : 2) CF
            =>
            (p-state[p] : noNat) (c-state[c] : empChan) (p-state[q] : (0 1 2)) .
```

The search finds 3 solutions as follows:

```
Solution 1 (state 12)
states: 13  rewrites: 74 in 0ms cpu (0ms real) (232704 rewrites/second)
CF --> (p-state[p]: 0 1) c-state[c]: empChan

Solution 2 (state 24)
states: 25  rewrites: 163 in 0ms cpu (0ms real) (238653 rewrites/second)
CF --> (p-state[p]: 0) c-state[c]: 1 | empChan
```

```
Solution 3 (state 33)
states: 34  rewrites: 239 in 0ms cpu (0ms real) (244126 rewrites/second)
CF --> (p-state[p]: noNat) c-state[c]: 1 | 0 | empChan

No more solutions.
states: 38  rewrites: 284 in 1ms cpu (1ms real) (243150 rewrites/second)
```

Note that although the reachable state space from `ic` 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. If the maximum number n of solutions is given, the search terminates when it finds n solutions. Therefore, even if the reachable state space from a given state is unbounded, the search command can be used and may terminate. If the maximum depth d of search is given, only the bounded reachable state space from a given state up to depth d is searched. Hence, the search command can be used as a bounded model checker. These options are not used in this report.

Chapter 3

The Chandy-Lamport Distributed Snapshot Algorithm ($\mathcal{CLDSA}$)

3.1 Motivation of the $\mathcal{CLDSA}$

As described before, many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. The global state of a distributed system consists of the states of every process and every channel in the system, where the state of a process is characterized by the state of its local memory and depends upon the context, and the state of a channel is characterized by the sequence of messages *"in-transit"*, those that have been sent on that channel, but not yet received by its destination process.

Given the fact that distributed systems are asynchronous and processes in the system do not share common clocks or memory, each process cannot record its local state at exactly the same time, namely that such global states can never be instantaneously done, and it leaves open the possibility of inconsistent global states. Moreover, the variability in message delays could lead to these separate processes constructing different global states for the same computation. It is not straightforward to obtain consistent global states of a $\mathcal{UDS}$, namely that what we obtained may be inconsistent. Therefore, it is necessary to apply some algorithms when we want to obtain consistent global states of a $\mathcal{UDS}$. One of those algorithms is known as the $\mathcal{CLDSA}$ [CL85], which was proposed by Chandy and Lamport in 1985. It can be used to determine consistent global states of a distributed system during its computation.

The $\mathcal{CLDSA}$ plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds — a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to

define "meaningful" and then to determine how the photographs should be taken.

Example 1 *We now consider the following system described in Figure 3.1 as an example to motivate the steps of the algorithm.*

The system contains one token (represented by the pentastar) that is passed from one process to another, and hence we call this system the "single-token conservation" system. Each process has two states, s_0 and s_1, where s_0 is the state in which the process does not possess the token and s_1 is the state in which it does. The initial state of P is s_1 and of Q is s_0. Each process has two events: (1) a transition from s_1 to s_0 with the sending of the token, and (2) a transition from s_0 to s_1 with the receipt of the token.

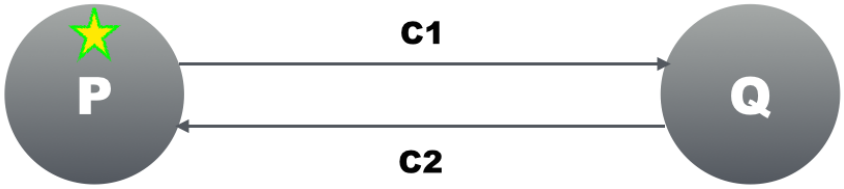

Figure 3.1 The single-token conservation system

In the example we shall assume that we can record the state of a channel instantaneously. Let C1 and C2 be the channel from P to Q and the channel from Q to P, respectively. The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the states of channels C1 and C2 are to be recorded and the instants at which the states of processes P and Q are to be recorded.

3.1.1 Scenarios of Inconsistent Global States

The global state of a distributed system consists of the sate of every process and the state of every channel in the system. Due to the asynchrony of distributed systems, the lack of globally shared memory, global clock and unpredictable message delays make recording such consistent global states non-trivial. What we obtained may be inconsistent.

We assume that the states of processes and channels can be recorded whenever we want. Now let us see two scenarios of inconsistent global states.

Scenario 1

- Step 1: We record the states of P and C2. It shows that one token in P and empty in C2.

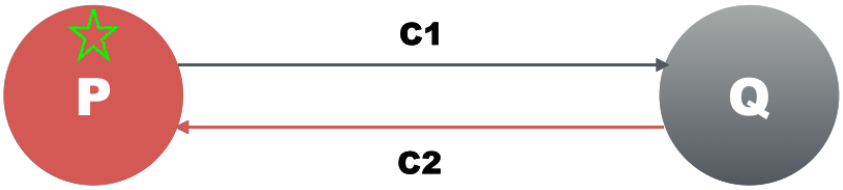

Figure 3.2 Recording the states of P and C2

- Step 2: P sends the token to Q by putting the token into C1.

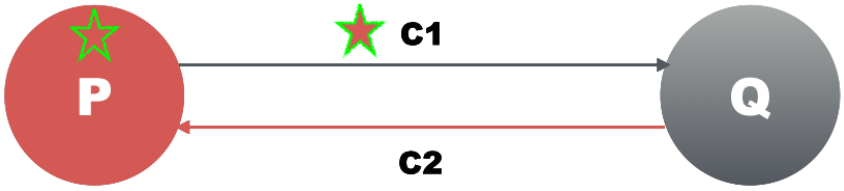

Figure 3.3 Sending the token (by putting it into C1)

- Step 3: We record the states of Q and C1 after sending the token. It shows that empty in Q and one token in C1.

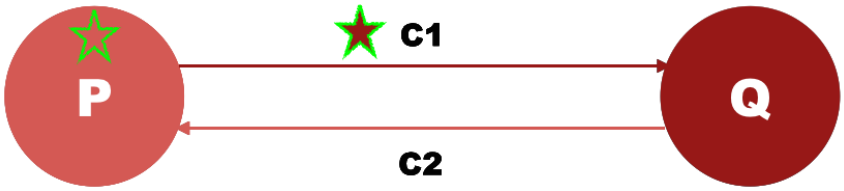

Figure 3.4 Recording the states of Q and C1

- Step 4: We can obtain the global state of the system by combining the states we recorded. And it show one token in P, one token in C1, and empty in others.

But we know, only one token should be in the system. Namely, we got an inconsistent global state.

Scenario 2

- Step 1: We record the states of Q and C1. It shows that empty in Q and empty in C1.
- Step 2: P sends the token to Q by putting the token into C1.
- Step 3: We record the states of P and C2 after sending the token. It shows that empty in P and empty in C2.
- Step 4: We can obtain the global state of the system by combining the states we recorded. And it show no token in the system.

But we know, one token should be in the system. Namely, we got another inconsistent global state.

3.2 Outline of the $\mathcal{CLDSA}$

When a snapshot is taken, each process in the system records its own local state, and the states of all its incoming channels. Since there is no globally shared memory or clock in a $\mathcal{UDS}$, processes can only communicate with each other through messages passed on channels that connect them. That leads to getting an inconsistent global state, namely that the snapshot may not be reachable from the state when the snapshot has started being taken.

The $\mathcal{CLDSA}$ uses a control message, called a marker whose role in a FIFO channel is to separate messages in the channels. The $\mathcal{CLDSA}$ can be initiated by any process, identified as the initiator. The initiator spontaneously records its state and starts executing the $\mathcal{CLDSA}$. Moreover, the $\mathcal{CLDSA}$ superimposes the underlying computation, i.e., it runs concurrently with, but does not alter, the underlying computation. The $\mathcal{CLDSA}$ requires processes to record their states, send markers and record some messages received, but does not interfere with the underlying computation.

The $\mathcal{CLDSA}$ cannot ensure that the states of all processes and all channels are recorded at exactly the same time. However, the $\mathcal{CLDSA}$ does ensure that the recorded process and channel states form a meaningful (consistent) global system state. The outline of the $\mathcal{CLDSA}$ in the form of rules is presented as the following.

Marker Sending Rule for a Process p

- p records its state;
- For each outgoing channel C on which a marker has not been sent, p sends one marker along C before p sends further messages along C.

Marker Receiving Rule for a Process q
On receiving a marker along a channel C:

if q has not recorded its state **then**

- q records its state;
- q records the state of C as the empty sequence.

else

- q records the state of C as the sequence of messages received along C after q's state was recorded and before q received the marker along C.

The $\mathcal{CLDSA}$ 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 as the sequence of messages received along the channel after its state was recorded and before it received the marker along the channel.

The $\mathcal{CLDSA}$ terminates in finite time after each process in the system has received a marker along all of its incoming channels. All the local snapshots get disseminated to all other processes, allowing all processes to determine the recorded consistent global state.

3.3 Termination of the $\mathcal{CLDSA}$

The *"Marker Sending Rule"* and the *"Marker Receiving Rule"* guarantee that if a marker is received along every channel, then each process will record its state and the states of all incoming channels. To ensure that the $\mathcal{CLDSA}$ terminates in finite time, each process must ensure that (L1) no marker remains forever in an incident input channel and (L2) it records its state within finite time of initiation of the algorithm.

The $\mathcal{CLDSA}$ can be initiated by one or more processes, each of which records its state spontaneously, without receiving markers from other processes. If process p records its state and there is a channel from p to a process q, then q will record its state in finite time because p will send a marker along the channel and q will receive the marker in finite time (L1). Hence if p records its state and there is a path (in the graph representing the system) from p to a process q, then q will record its state in finite time because, by induction, every process along the path will record its state in finite time. Termination in finite time is ensured if for every process q:

- Process q spontaneously records its state, or
- There is a path from a process p, which spontaneously records its state, to q.

In particular, if the graph is strongly connected and at least one process spontaneously records its state, then all processes will record their states in finite time (provided L1 is ensured).

The $\mathcal{CLDSA}$ described so far allows each process to record its state and the states of incoming channels. The recorded process and channel states must be collected and assembled to form the recorded global state (snapshot state). We shall not describe algorithms for collecting the recorded information because such algorithms have been described elsewhere [DS80, MC82]. A simple algorithm for collecting information in a system whose topology is strongly connected is for each process to send the information it records along all outgoing channels, and for each process receiving information for the first time to copy it and propagate it along all of its outgoing channels. All the recorded information will then get to all the processes in finite time, allowing all processes to determine the recorded global state.

3.4 The Distributed Snapshot Reachability ($\mathcal{DSR}$) Property

Let s_1, s_* and s_2 be the global state when the $\mathcal{CLDSA}$ starts, the snapshot, and the global state when the $\mathcal{CLDSA}$ terminates, respectively. Although the snapshot s_* may not be identical to any of the global states that occurs in the computation between s_1 and s_2, the desired properties the $\mathcal{CLDSA}$ should satisfy are the following $\mathcal{DSR}$ property:

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

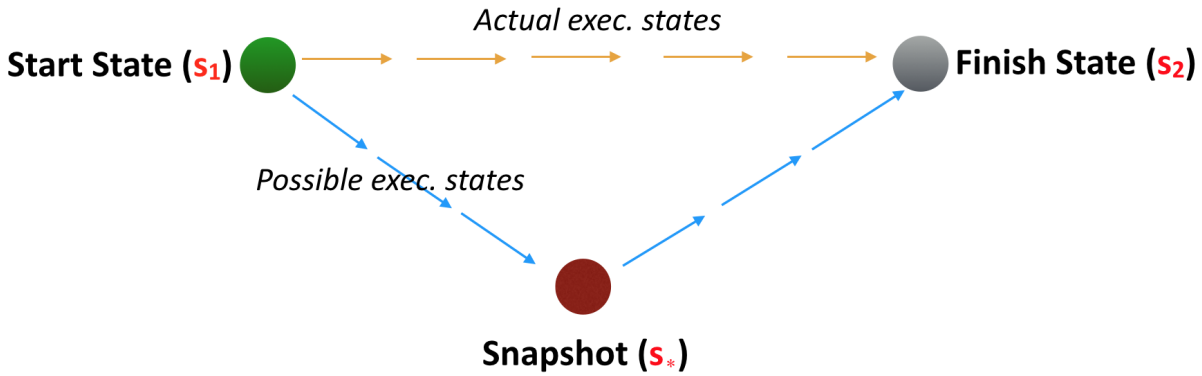

Figure 3.5 The Distributed Snapshot Reachability ($\mathcal{DSR}$) Property

Chapter 4

A Study on How to Specify and Model Check the $\mathcal{CLDSA}$ in Maude

Many model checkers such as symbolic model checkers, explicit-state model checkers and SAT/SMT-based bounded model checkers have been proposed [CCG+02, Hol04, dMOR+04]. Accordingly, many case studies have been conducted by applying them to mechanical analysis of systems including distributed systems, protocols and algorithms [TS11, AP10, OF07a].

To the best of our knowledge, however, there are few case studies except for [OH12] in which the Chandy-Lamport Distributed Snapshot Algorithm ($\mathcal{CLDSA}$) [CL85] is mechanically analyzed with model checkers. As we all know, many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states. The $\mathcal{CLDSA}$, which was proposed by Chandy and Lamport in 1985, can be used to determine consistent global states of a distributed system during its computation. Since it is very important and also non-trivial, it deserves to be formally specified and verified with respect to (w.r.t.) some significant properties.

Let s_1 (called the start state) be the state when the $\mathcal{CLDSA}$ starts (a distributed snapshot starts being taken), s_* be the snapshot, and s_2 (called the finish state) be the state when the $\mathcal{CLDSA}$ terminates (the snapshot completes being taken). The $\mathcal{CLDSA}$ should enjoy the property that s_* is always reachable from s_1 and s_2 is always reachable from s_*. The property is called the Distributed Snapshot Reachability ($\mathcal{DSR}$) property, which guarantees the $\mathcal{CLDSA}$ takes consistent global states of a distributed system.

As far as we have investigated, to formalize the $\mathcal{DSR}$ property, we have to consider two kinds of states, (1) the states of a $\mathcal{UDS}$, and (2) the states of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. In existing temporal logics such as LTL and CTL, only one kind of states are considered when they are used to formalize system properties. Thus, it is not straightforward to express the $\mathcal{DSR}$ property in LTL and CTL.

There is an existing study [OH12] in which a distributed system superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command, which demonstrates the power of the command, namely that more general invariant properties can be checked by the command than standard LTL and CTL model checkers. The case study also demonstrates the importance of case anal-

ysis in specification, which often needs to be conducted for interactive theorem proving. It is also worth noting that the formal specification of the $\mathcal{CLDSA}$ in Maude depends on neither the number of processes nor the number of channels in the distributed system, although we need to fix them to conduct model checking.

Learning the existing formal specification and verification of a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ in Maude is a good way to understand the $\mathcal{CLDSA}$ more completely and deeply.

In this chapter, I will give an explanation on the existing study [OH12], including how to specify distributed systems superimposed by the algorithm in Maude and how to model check the $\mathcal{DSR}$ property with the Maude search command. Moreover, I will also conduct some more experiments to model check the $\mathcal{DSR}$ property for the $\mathcal{CLDSA}$.

4.1 System Specification of the $\mathcal{CLDSA}$

In the existing study, the authors described their way of modeling (formalizing) the $\mathcal{CLDSA}$. What is modeled (formalized) is actually $\mathcal{UDS}$s on which the $\mathcal{CLDSA}$ is superimposed. In other words, two parts have been considered in the system specification, one is the $\mathcal{UDS}$ part, and the other one is the $\mathcal{CLDSA}$ part. In the $\mathcal{UDS}$ part, the processes in the system typically have three kinds of events (*local*, *send* and *receive*). And in the $\mathcal{CLDSA}$ part, the behaviors of the algorithm are reflected.

For the *local* event, the processes change their states without sending or receiving tokens; for the *send* event, the process send tokens owned by themselves to another processes by putting the tokens into their outgoing channels, and change their states accordingly; for the *receive* event, the process receive tokens from another processes by getting the tokens from their incoming channels, and change their states accordingly. Note that the processes in the system may not have any incoming channels or outgoing channels.

As we have described, a $\mathcal{UDS}$ consists of one or more processes that are connected with directed channels that are unbounded queues. To cover all possible situations, a system may consists of one process only, and some processes have no outgoing channels, no incoming channels, or neither of them, although such a system may not be regarded as a distributed system, or may be regarded as multiple distributed systems. Processes exchange non-marker messages that are called tokens and may consume them. They suppose that the state of each process only depends on the set of tokens owned by the process, and also suppose that at most one distributed snapshot is taken in each computation of a distributed system, and there is no self-channel, namely a channel from a process to the same process.

4.1.1 State Expression for a $\mathcal{UDS}$ Superimposed by the $\mathcal{CLDSA}$

Basic Data Used

Processes (or process identifiers) are denoted as $p(0)$, $p(1)$, ..., and the sort is `Pid`. The corresponding module is described as follows:

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

where `Nat` is for natural numbers such as 0, 1, 2, ... to identify which process we are referring to.

Tokens are denoted as $t(0)$, $t(1)$, ..., and the sort is `Token`. A marker is denoted as `marker`, and the sort is `Marker`. Sort `Msg` is declared as a super-sort of `Token` and `Marker`. The corresponding modules are described as follows:

```
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
```

Sorts `EmpChan`, `NeChan`, and `Chan` are for the empty channel, non-empty channels, and channels that may be empty or non-empty, respectively. The empty channel is denoted as `empChan`. A non-empty channel that consists of n messages in the order of m_0, m_1, ..., m_{n-1} is denoted as $m_0 \mid m_1 \mid \ldots \mid m_{n-1} \mid$ `empChan`. A function `put` takes a channel c and a message m (namely a token or a marker), and returns the channel obtained by putting m into c at the end (channels are assumed to be FIFO). The corresponding module is described as follows:

```
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 .

  vars M1 M2 : Msg .
  var C : Chan .

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

Since the state of a process only depends on the set of tokens owned by the process itself, the state can be expressed as the soup of tokens. The sort for soups of tokens is `PState` that is also declared as a super-sort of sort `Token`. The empty soup is denoted as `noToken`. The soup of n tokens $t(0)$, $t(1)$, ..., $t(n-1)$ is denoted as $t(0)\ t(1)\ \ldots\ t(n-1)$. Note that `noToken` is declared as an identity of the constructor (the juxtaposition operator) of soups of tokens. The corresponding module is described as follows:

```
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
```

Each process has three kinds of progresses as follows:

1. has not yet started the $\mathcal{CLDSA}$,
2. has already started but not yet completed the $\mathcal{CLDSA}$, or
3. has already completed the $\mathcal{CLDSA}$.

Those situations are denoted as `notYet`, `started` and `completed`, respectively. The corresponding sort is `Prog`, and the module is described as follows:

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

Observable Components and (Meta) Configurations

The state of $\mathcal{UDS}$ consists of the state of each process and the state of each channel in the system. The state ps of a process p is denoted as `p-state`$[p] : ps$, and the state cs of a channel from a process p to a process q is denoted as `c-state`$[p, q, n] : cs$, where n is a natural number. Since there may be more than one channel from process p to process q, n is used to identify one of them. "`p-state`$[p] : ps$" and "`c-state`$[p, q, n] : cs$" are called observable components. The corresponding sort is `OCom`, and the module is described as follows:

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

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

  op (cnt :_) : Nat -> OCom [ctor] .
  op (#ms[_] :_) : Pid Nat -> OCom [ctor] .
  op (done[_,_,_] :_) : Pid Pid Nat Bool -> OCom [ctor] .
  op (prog[_] :_) : Pid Prog -> OCom [ctor] .
  op (consume :_) : Bool -> OCom [ctor] .
endfm
```

In addition to `p-state` and `c-state` observable components, we also need to add the following observable components in the above module `OBSERVABLE-COMPONENT` for control information to specify the behaviors of the $\mathcal{CLDSA}$:

- (**cnt** : n): n is the number of processes that have not yet completed the $\mathcal{CLDSA}$. When n becomes 0, a distributed snapshot has been taken.
- (**prog**$[p] : pg$): pg is the progress of a process p, indicating that the process has not yet started, has started, or completed the $\mathcal{CLDSA}$. $notYet$, $started$, or $completed$ can be used to represent them respectively.
- (#**ms**$[p] : n$): n is the number of incoming channels to a process p from which markers have not yet been received. When n becomes 0, p has received markers from all of its incoming channels, implying that p completes the $\mathcal{CLDSA}$ if p has one or more incoming channels. Note that p may not have any incoming channels and then n may be 0 even in initial states.
- (**done**$[p, q, n] : b$): b is either $true$ or $false$. If b is $true$, a process q has received a marker from the incoming channel identified by n from a process p to q. Otherwise, q has not.

– (**consume** : b): b is either $true$ or $false$. If b is $true$, tokens may be consumed. Otherwise, tokens are not.

The control information is expressed as **control**(...) that is also a meta configuration component, where ... is a soup of `cnt`, `prog`, `#ms`, `done` and `consume` observable components. When the content in each `prog` component is *completed*, a snapshot of the whole system has been taken. To make the system specification less complicated, the component `cnt` is used, although it seems to be redundant.

Example 2 (*A configuration of the 3-process & 5-channel system*) *Let us consider the 3-process & 5-channel system.*

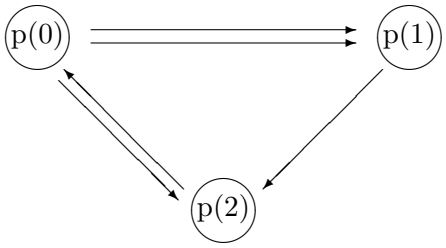

Figure 4.1 The 3-process & 5-channel system

We suppose that p(0) has one token t(1), the other processes have no token, one channel from p(0) to p(1) consists of one token t(0), and the other channels are empty. The state of the system is expressed as the following configuration:

(p-state[p(0)] : $t(1)$) (p-state[p(1)] : $noToken$)
(p-state[p(2)] : $noToken$) (c-state[p(0), p(1), 0] : ($t(0)$ | $empChan$))
(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] : $empChan$)

The state of a system is expressed as a soup of observable components that is called a configuration. The corresponding sort is `Config` that is a super-sort of `OCom`. The empty configuration is denoted as `empConfig` that is an identity of the constructor of soups of observable components. The corresponding module is described as follows:

```
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 .

  --- bcast
  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] .

  --- inchans
  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] . --- no OC!!!
endfm
```

where the function `bcast` takes a configuration cf, a process p and a marker $marker$, and returns another configuration by putting one marker into each of its outgoing channels. And the function `inchans` takes a configuration cf and a process p, and returns another configuration by initializing the states of all incoming channels of process p.

Moreover, to take into account a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$, we should add some more information to record the start state when the $\mathcal{CLDSA}$ starts, the snapshot and the finish state when the $\mathcal{CLDSA}$ terminates.

The states of a $\mathcal{UDS}$, the start state, the snapshot, and the finish state are expressed as **base-state**(...), **start-state**(...), **snapshot**(...), and **finish-state**(...), respectively, where ... is a soup (associative-commutative collection) of `p-state` and `c-state` observable components. Those are called meta configuration components and the corresponding sort is `MCComp`. The corresponding modules are described as follows:

```
fmod META-CONFIGURATION-COMPONENT is
  pr CONFIGURATIONS .
  sort MCComp .

  ops base-state start-state finish-state
      snapshot control : Config -> MCComp .
endfm
```

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

  var MOC : MCComp .
  eq MOC MOC = MOC .
endfm
```

A global state of a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ is expressed as a soup of meta configuration components:

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

which is called a meta configuration. And the corresponding sort is `MConfig`. Initially, all of sc, ssc and fc are $empConfig$. If sc is not $empConfig$, it means that a distributed snapshot has started being taken. If fc is not $empConfig$, it means that a distributed snapshot has been taken and then ssc is the snapshot.

Example 3 (*A meta configuration of the 3-process & 5-channel system*) *Let us consider the 3-process & 5-channel system again.*

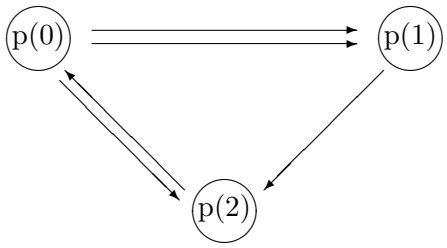

Figure 4.2 The 3-process & 5-channel system

Initially, we suppose that p(0) has one token t(1), the other processes have no tokens, and one channel from p(0) to p(1) consists of one token t(0) , and the other channels are empty. Tokens may be consumed. Then the global sate of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ can be expressed as a meta configuration as follows:

base-state((p-state[p(0)] : $t(1)$) (p-state[p(1)] : $noToken$) (p-state[p(2)] : $noToken$)
(c-state[p(0), p(1), 0] : ($t(0)$ | $empChan$))
(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($empConfig$)
snapshot($empConfig$)
finish-state($empConfig$)
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 : $true$))

4.1.2 State Transitions for a $\mathcal{UDS}$ Superimposed by the $\mathcal{CLDSA}$

What each process in a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ does is as follows:

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 $\mathcal{UDS}$ part.

In the rest of the report, `BC`, `CC`, `SC` and `SSC` are Maude variables of sort `Config`, `P` and `Q` are Maude variables of sort `Pid`, `T` is a Maude variable of sort `Token`, `PS` is a Maude variable of sort `PState`, `N` is a Maude variable of sort `Nat`, `C` and `C'` are Maude variables of sort `Chan`, and `NzN` and `NzN'` are Maude variables of sort `NzNat`. Note that `NzNat` for natural numbers except 0, such as $1, 2, 3, \ldots$.

For a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$, we can have the following rewrite rules.

- **Consumption of Tokens.** For the first case, it can be described by the following rewrite rule:

```
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) .
```

where `chgStt` is the label of the rewrite rule. When a distributed snapshot has been taken, namely that the content of the meta configuration component `finish-state` is not `empConfig`, then we intentionally stop the underlying computation because we want to reduce the size of the reachable state space. This is why we have `finish-state(empConfig)` on both sides of the rule.

- **Sending of Tokens.** For the second case, it can be described by the following rewrite rule:

```
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) .
```

where `put` takes a channel `C` and a token `T` and returns the channel obtained by putting `T` into `C` at the end. If a given term contains an instance of `(p-state[P] : (T PS)) (c-state[P,Q,N] : C)`, the instance is replaced with the corresponding instance of `(p-state[P] : PS) (c-state[P,Q,N] : put(C,T))`.

- **Receipt of Tokens.** For the third case, we need to take into account four subcases:

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

```
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) .
```

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

```
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) .
```

3. The process has started the $\mathcal{CLDSA}$, 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) .
```

4. The process has started the $\mathcal{CLDSA}$, 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) .
```

When a process `P` starts the $\mathcal{CLDSA}$, it initializes the record of each incoming channel (identified by a natural number `N`) from each process `Q` unless a marker has been received from the channel. The initialization is described by adding "`c-state[Q,P,N] : empChan`" into the `snapshot` meta configuration component. For the third case, such a record is updated by putting the received token. For the other three cases, it is not necessary to update such a record.

- **Record of Process States.** If a process has already received a marker, namely that it has already recorded its state as well. Hence, we only need to take into account the case in which a process has not yet received any markers. When a process records its state in the case, the case is split into two subcases:

  1. The process globally initiates the $\mathcal{CLDSA}$, namely that it is the first process that records its state in the system. This case is further split into three subcases:

     (a) The $\mathcal{UDS}$ only consists of the process. For this case, the $\mathcal{CLDSA}$ will be completed.

```
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) .
```

where `start-state(empConfig)` in the left hand side of the rewrite rule indicates that the process `P` is the first one that starts taking a distributed snapshot. And then the start state is recorded as "`(p-state[P] : PS)`" in the `start-state` meta configuration component, and the state of process `P` is recorded and the state of each incoming channel to `P` is initialized as"`(p-state[P] : PS)`" in the `snapshot` meta configuration component. And the observable component `(cnt : 0)` in `control` meta configuration component means that the $\mathcal{CLDSA}$ will be completed, representing by the observable component `(prog[P] : completed)`.

(b) The system consists of more than one process, and the process does not have any incoming channels. For this case, the process will locally complete the $\mathcal{CLDSA}$.

```
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 .
```

where `bcast(BC,P,marker)` puts one marker into each of its outgoing channels, and `sd` stands for symmetric difference, takes two natural numbers x, y, and returns $(x - y)$ if $x > y$ and $(y - x)$ otherwise. And the observable component `(#ms[P] : 0)` in `control` meta configuration component means that the process does not have any incoming channels. After recording its state and putting a marker into each of its outgoing channels, the process will locally complete the $\mathcal{CLDSA}$, representing by the observable component `(prog[P] : completed)`.

(c) 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) .
```

where `inchans(BC,P)` initializes the states of its all incoming channels. `(#ms[P] : NzN')` means that the process `P` has one or more incoming channels because `NzN'` is a non-zero natural number, and then the system consists of more than one process. `P` has not started the $\mathcal{CLDSA}$, which is indicated by `(prog[P] : notYet)`.

The start state is recorded as "`(p-state[P] : PS) BC`" in the `start-state` meta configuration component, and the state of process `P` is recorded and the state of each incoming channel to `P` is initialized as"`(p-state[P] : PS) inchans(BC,P)`" in the `snapshot` meta configuration component.

In other words, after recording its state, the process `P` will also need to start to record the states of all its incoming channels, representing by `inchans(BC,P)`.

2. The process does not, namely that there exists another process that has globally initiated the $\mathcal{CLDSA}$. 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 have not completed the $\mathcal{CLDSA}$. For this case, the $\mathcal{CLDSA}$ will be completed.

```
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) .
```

The condition `SC =/= empConfig` indicates that the process P is not the

first one that starts the $\mathcal{CLDSA}$, namely that there exists another process in the system that has already started the $\mathcal{CLDSA}$.

(b) The process does not have any incoming channels, and there are some other processes that have not completed the $\mathcal{CLDSA}$. For this case, the process will locally complete the $\mathcal{CLDSA}$.

```
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) .
```

(c) 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) .
```

The three subcases in the second case are almost the same as what described in the first case. The main difference between the two cases is the condition part `SC =/= empConfig`, which indicates that the process P is not the first one that starts the $\mathcal{CLDSA}$, namely that there exists another process in the system that has already started the $\mathcal{CLDSA}$.

- **Receipt of Markers.** When a process receives a marker from an incoming channel, we first need to take into account the following two cases:

1. The process has not yet started the $\mathcal{CLDSA}$. 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 the $\mathcal{CLDSA}$ except for the process, which implies that the process does not have any outgoing channels.

```
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) .
```

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

```
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 .
```

(c) 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 .
```

Note that when a process receives a marker from an incoming channel, the natural number in the `#ms` observable component must be greater than zero and the natural number in the `cnt` observable component must be greater than zero. This is why we have `(cnt : NzN)` on both sides.

2. The process has already started the $\mathcal{CLDSA}$. This case is further split into three subcases:
   (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 the $\mathcal{CLDSA}$ 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) .
```

   When the process `P` receives the marker, a global snapshot has been taken because of the assumption. The receipt of the marker by `P` does not affect (the contents of) the global snapshot. This is why the rewrite rule does not have any `snapshot` meta configuration components. The finish state is recorded as `(p-state[P] : PS) (c-state[Q,P,N] : C) BC` in the `finish-state` meta configuration component.
   (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 the $\mathcal{CLDSA}$.

```
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 .
```

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

```
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 .
```

Case analysis, which often needs to be conducted for interactive theorem proving, has played a very important role in the specification. The system specification depends on neither the number of processes nor the number of channels, which demonstrates the power of Maude. Note that the same specification can also be described in CafeOBJ, a sibling language of Maude. The current implementation of Maude is superior to that of CafeOBJ, however, in terms of execution performance. This is why the authors have used Maude in their case study.

To model check the $\mathcal{DSR}$ property ($\mathcal{RP}1$ and $\mathcal{RP}2$), however, we need to fix those numbers, which are described in initial meta configurations. The corresponding module is described as follows:

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

  op imc : -> MConfig .
  eq imc
     = 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)) .
```

A meta configuration is a global view (state) of the system (a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$). We need to have such a global view so that we can check (or verify) the $\mathcal{DSR}$ property of the system. Global views of the system are also used to describe rewrite rules. For each of most basic actions of processes such as receipt of a marker, there are multiple rewrite rules that have been obtained by case analyzes (or case distinctions) based on predicates that are not locally observable by any process. The purpose of the case analyzes is to cover all possible situations. The case analyzes based on global predicates do not affect the action of each process designated by the system. This is because each rewrite rule whose main player is a process `P` can modify only the `P`'s state, incoming channels and/or outgoing channels w.r.t. the `base-state` meta configuration component.

We informally reason about the algorithm to write rewrite rules. For **Record of Process States**, we argue the following. "If a process has already received a marker, it has already recorded its state as well. Hence, we only need to take into account the case in which a process has not yet received any markers." For the case 1-(a) in **Receipt of Markers**, we argue the following. "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." This informal reasoning can reduce the number of rewrite rules but may overlook some possible situations.

Since any process of making formal models and writing formal specifications is not formal, however, any way of dosing so may overlook some possible situations. As far as we know, all we can do is to carefully make formal models and write formal specifications, and carefully validate them by some means such as animation. Since formal specifications in Maude (and any other OBJ family languages such as CafeOBJ) are executable, we can animate/execute formal specifications to validate them.

4.2 Model Checking of the $\mathcal{DSR}$ Property

As we described before, the desired properties the $\mathcal{CLDSA}$ should satisfy are the following $\mathcal{DSR}$ property:

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

where s_1 (called the start state) is the state when the $\mathcal{CLDSA}$ starts (a distributed snapshot starts being taken), s_* is the snapshot, and s_2 (called the finish state) is the state when the $\mathcal{CLDSA}$ terminates (the snapshot completes being taken).

As far as we have investigated, to formalize the $\mathcal{DSR}$ property, we have to consider two kinds of states, (1) the states of a $\mathcal{UDS}$, and (2) the states of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. In existing temporal logics such as LTL and CTL, only one kind of states are considered when they are used to formalize system properties. Thus, it is not straightforward to express the $\mathcal{DSR}$ property in LTL and CTL.

In the existing study [OH12], the authors model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command after a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude. This demonstrates the power of the command, namely that more general invariant properties can be checked by the command than standard LTL and CTL model checkers. It is also worth noting that the formal specification of the $\mathcal{CLDSA}$ in Maude depends on neither the number of processes nor the number of channels in the distributed system, although we need to fix them to conduct some model checking experiments.

The $\mathcal{DSR}$ property can be divided into $\mathcal{RP}1$ and $\mathcal{RP}2$, which can be checked with the Maude search command, respectively. Let `imc` be an initial meta configuration of a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. We suppose that the system consists of n processes $\mathrm{p}(0), \ldots, \mathrm{p}(n-1)$.

The following search (called the search for snapshots) finds all states in which a snapshot has been taken:

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

where EXPERIMENT is a module where the system specification module is imported and some variables such as SC, FC, SSC and MC are declared. Let m_0 be the number of the solutions to the search for snapshots.

The following search (called the search for $\mathcal{RP}1$) finds all states in which a snapshot has been taken such that the snapshot SSC is reachable from the start state SC under the $\mathcal{UDS}$:

```
search in EXPERIMENT :
  imc =>* 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(n-1)]: notYet) (consume : b))
      =>
      base-state(SSC) finish-state(empConfig)
      control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b)) .
```

where b is true if tokens may be consumed, and false otherwise. The rewrite expression in the condition checks if SSC is reachable from SC with the three rewrite rules "chgStt", "sndTkn" and "recTkn&notYet&~done", namely under the underlying distributed system. We need to have finish-state(empConfig) and control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b)) in the rewrite expression to enforce using the three rewrite rules.

Let m_1 be the number of the solutions to the search for $\mathcal{RP}1$. If m_1 equals m_0, the $\mathcal{CLDSA}$ enjoys $\mathcal{RP}1$ w.r.t. the underlying distributed system.

$\mathcal{RP}2$ can be checked likewise. All needed to do is to replace SC and SSC with SSC and FC, respectively, in the rewrite expression. This search is called the search for $\mathcal{RP}2$ and it can be described as follows:

```
search in EXPERIMENT :
  imc =>* 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(n-1)]: notYet) (consume : b))
      =>
      base-state(FC) finish-state(empConfig)
      control((prog[p(0)]: notYet) ... (prog[p(n-1)]: notYet) (consume : b)) .
```

Let m_2 be the number of the solutions to the search for $\mathcal{RP}2$. If m_2 equals m_0, the $\mathcal{CLDSA}$ enjoys $\mathcal{RP}2$ w.r.t. the underlying distributed system.

The $\mathcal{DSR}$ property ($\mathcal{RP}1$ and $\mathcal{RP}2$) 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.

4.2.1 Conducting Some Experiments for the $\mathcal{DSR}$ Property

To model check the $\mathcal{DSR}$ property the $\mathcal{CLDSA}$ should satisfy, I have already conducted some model checking experiments with several different $\mathcal{UDS}$s with the Maude search command.

For each experiment of a concrete $\mathcal{UDS}$, we should do three sub-experiments to find m_0, m_1 and m_2, respectively, and then check whether they are equal or not. If and only if $(m_0 == m_1 == m_2)$, then the $\mathcal{DSR}$ property holds for this concrete $\mathcal{UDS}$.

Let us consider the following concrete system:

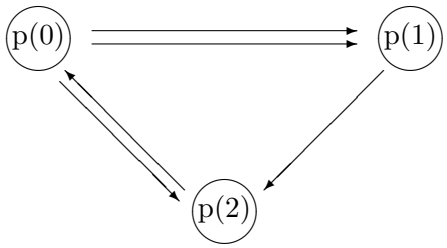

Figure 4.3 The 3-process & 5-channel system

- There are three processes p(0), p(1), p(2) in the system.
- There are at most two tokens t(0), t(1) in the system.
- The state of each process only depends on the tokens owned by itself. So, the state of each process can be expressed as $\varnothing$, {t(0)}, {t(1)} or {t(0),t(1)}. Initially, p(0) has the two tokens t(0), t(1), and the other processes do not have any tokens.
- There are five channels: $p(0) \rightarrow p(1)$, $p(0) \rightarrow p(1)$, $p(0) \rightarrow p(2)$, $p(1) \rightarrow p(2)$, $p(2) \rightarrow p(0)$. Initially, each channel is empty sequence.

What each process in the system does repeatedly is as follows:

1. The process may consume a token owned by it. Accordingly, it changes its state.
2. The process may put a token owned by it in one of its outgoing channels if it has some outgoing channels. Accordingly, it changes its state.
3. The process may get a token from one of its non-empty incoming channels if it has some non-empty incoming channels. Accordingly, it changes its state.

The results returned by Maude shows that all the searches for snapshots, $\mathcal{RP}1$ and $\mathcal{RP}2$ find 190,434 solutions. Therefore, the algorithm enjoys $\mathcal{RP}1$ and $\mathcal{RP}2$ with respect to the 3-process & 5-channel system. Note that the number of reachable states (meta configurations) from *imc* (initial meta configuration) with respect to the 3-process & 5-channel system is 3,587,681.

The following table shows the information of some other experiments I have conducted.

Table 4.1 Model checking experiments for some concrete $\mathcal{UDS}$s

	#Processes	#Channels	#Tokens	Consume	#Solutions	Time (mins)
imc00	2	2	1	F	40	0.03
imc01	2	2	1	T	55	0.05
imc02	3	2	3	T	874	0.75
imc03	2	3	2	T	9,315	7.90
imc04	3	4	2	F	20,851	34.6
imc05	3	4	2	T	33,344	63.1
imc06	3	4	2	T	81,740	169
imc07	3	4	3	T	—	910
imc08	3	5	2	T	190,434	828
imc09	5	8	1	T	2,380	9.47
imc10	5	8	2	T	—	243

where

- imc** indicates initial meta configurations of concrete $\mathcal{UDS}$s superimposed by the $\mathcal{CLDSA}$ we are conducting.
- #Processes indicates the number of processes in the $\mathcal{UDS}$.
- #Channels indicates the number of channels in the $\mathcal{UDS}$.
- #Tokens indicates the number of tokens in the $\mathcal{UDS}$, namely that the total number of tokens in processes and channels.
- Consume indicates whether tokens can be consumed or not. The value of consume b is either *true* (T) or $false$ (F). If b is *true*, tokens may be consumed. Otherwise, tokens are not.
- #Solutions indicates the number of solutions we obtained from Maude system by using the search command, namely that the values of m_0, m_1 and m_2. Note that all values of them are the same, meaning that the $\mathcal{DSR}$ property holds in the concrete $\mathcal{UDS}$.
- Time (mins) indicates how much time we need to find the solution of each sub-experiment. Since for each experiment of a concrete $\mathcal{UDS}$, we should do three sub-experiments to find m_0, m_1 and m_2, respectively, and then check whether they are equal or not, so we need 3 times the value of Time (mins) showed in the table to check whether the $\mathcal{DSR}$ property holds or not for the concrete $\mathcal{UDS}$.

Note that for some concrete $\mathcal{UDS}$s such as imc07 and imc10, we cannot obtain the solutions because of the *state explosion* problem in case that the number of components (process, channel and token) in the system are large enough to the Maude system.

As described before, a $\mathcal{UDS}$ consists of a finite set of processes and a finite set of channels, which can be described by a labeled, directed graph in which the vertices represent the processes and the directed edges represent the channels.

The system may be different even though the number of processes and channels are exactly the same, because it also depends on the direction of the channels and how to connect the processes by channels. This is the reason why the number of solutions are different for some $\mathcal{UDS}$s with the same number of processes, channels and tokens. And the following are the corresponding labeled, directed graphs representing the $\mathcal{UDS}$s described in **Table 4.1**.

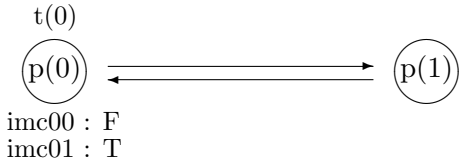

Figure 4.4 The system for `imc00` & `imc01`

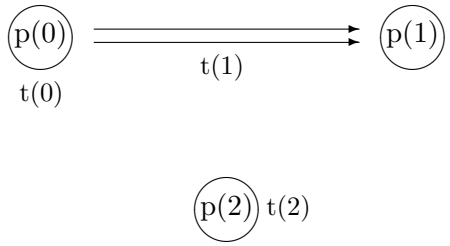

Figure 4.5 The system for `imc02`

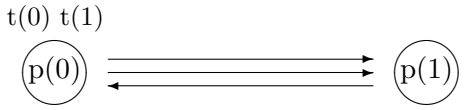

Figure 4.6 The system for `imc03`

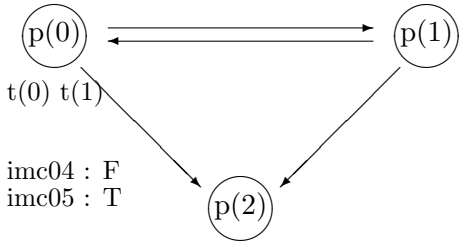

Figure 4.7 The system for `imc04` & `imc05`

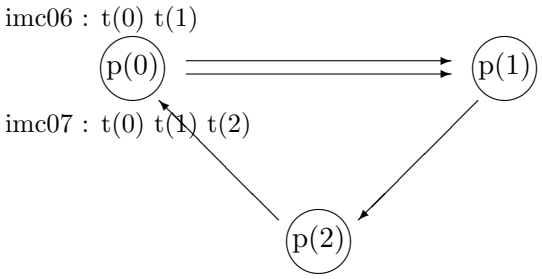

Figure 4.8 The system for `imc06` & `imc07`

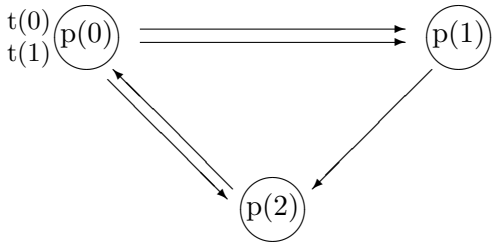

Figure 4.9 The system for `imc08`

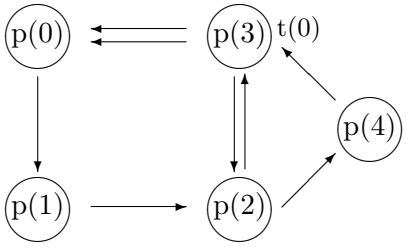

Figure 4.10 The system for `imc09`

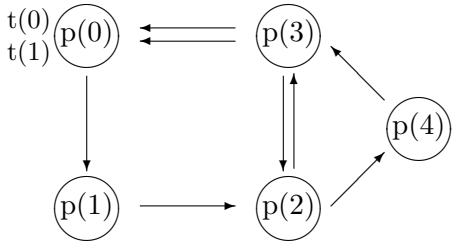

Figure 4.11 The system for `imc10`

Only model checking does not let us conclude that the $\mathcal{CLDSA}$ enjoys the $\mathcal{DSR}$ property ($\mathcal{RP}1$ and $\mathcal{RP}2$) for all distributed systems. One possible way to achieve this goal is to prove that if the $\mathcal{CLDSA}$ does not enjoy the $\mathcal{DSR}$ property for a distributed system that consists of an arbitrary number of processes and an arbitrary number of channels, then neither does it for a smaller system that consists of a few processes and a few channels such as the 3-process & 5-channel system.

Chapter 5

A Consideration on How to Model Check the $\mathcal{DSR}$ Property

5.1 Motivation of the Consideration

Many problems in distributed systems such as stable property detection and checkpointing can be cast in terms of the problem of detecting global states.

The $\mathcal{CLDSA}$ [CL85], which was proposed by Chandy and Lamport in 1985, can be used to determine consistent global states of a distributed system during its computation. Since it is very important and also non-trivial, it deserves to be formally specified and verified with respect to (w.r.t.) the $\mathcal{DSR}$ property.

As far as we have investigated, to formalize the $\mathcal{DSR}$ property, we have to consider two kinds of states, (1) the states of a $\mathcal{UDS}$, and (2) the states of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. In existing temporal logics such as LTL and CTL, only one kind of states are considered when they are used to formalize system properties. Thus, it is not straightforward to express the $\mathcal{DSR}$ property in LTL and CTL.

Moreover, there is an existing study [OH12] in which a distributed system superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command. We do not, however, think that the existing study [OH12] provides the sufficiently good foundation backing up that the $\mathcal{CLDSA}$ is surely model checked w.r.t. the $\mathcal{DSR}$ property, because the authors did not discuss whether the property is faithfully expressed or not. And then the $\mathcal{DSR}$ property encoded in the Maude search command are neither readable nor comprehensible. To make it executable in Maude, moreover, the system superimposed by the $\mathcal{CLDSA}$ has been specified in a very concrete way, in which the state of each process only depends on the tokens owned by the process itself. We do think that it is necessary to make sure that the property is faithfully expressed to claim that the property is model checked for the $\mathcal{CLDSA}$.

To complement the existing study [OH12], we consider how to surely model check the $\mathcal{DSR}$ property [ZOZ15]. To this end, we should find a way to faithfully express the $\mathcal{DSR}$ property. Our way to express the property relies on two state machines, although

the two state machines are closely related. And the property used in the existing study relies on only one state machine. Our way to express the $\mathcal{DSR}$ property has been affected by the Chandy-Misra's [CM88].

From here on, we specify the system in a more abstract way, namely that we can just use one kind of symbol such as *ps* instead of a set of tokens to express its state, use messages instead of tokens, and consider marker as a special kind of message.

5.2 Modeling a $\mathcal{UDS}$ as a State Machine

Each process in the distributed system has its own local state, and so does each channel. The state of a $\mathcal{UDS}$ should consist of the state of each process and the state of each channel in the system, where the state of a process is characterized by the state of its local memory and depends upon the context, and the state of a channel is characterized by the sequence of messages *"in-transit"*, those that have been sent on that channel, but not yet received by its destination process.

State machine can be used to model distributed systems. It consists of a set of states and a set of state transitions (i.e., a binary relation over the states).

5.2.1 State Expression for a $\mathcal{UDS}$

We can use *name-value* pairs (called observable components) to express the states of processes and channels, where name may have parameters, and use a set of process and channel states to represent the state of a $\mathcal{UDS}$ as follows:

```
op p-state[_]:_ : Pid PState → OCom [ctor] .
op c-state[_, _, _]:_ : Pid Pid Nat CState → OCom [ctor] .
op empConfig : → Config [ctor] .
op _ _ : Config Config → Config [ctor assoc comm id: empConfig] .
```

where

- **Pid** for process identifiers such as $p, q, r, \ldots$;
- **Nat** for natural numbers such as $0, 1, 2, \ldots$, which is used to identify one of the channels in case that there are more than one channel from one process to another;
- **OCom** for observable components;
- **Config** for $\mathcal{UDS}$ states, a super-sort of OCom.

Given a term s of sort Config, let p-state(s) and c-state(s) be the set of p-state observable components in s and the set of c-state observable components in s, respectively. Given two p-state observable components (p-state[p1] : ps_1) and (p-state[p2] : ps_2), they are equal up to process states iff (p1 = p2), and given two c-state observable components (c-state[p1, q1, n1] : cs_1) and (c-state[p2, q2, n2] : cs_2), they are equal up to channel states iff ((p1 = p2) $\wedge$ (q1 = q2) $\wedge$ (n1 = n2)).

Given two terms $s1$ and $s2$ of sort Config, they are equal up to process and channel states iff (if $s1$ contains (p-state[p1] : ps_1), then $s2$ contains (p-state[p1] : ps_2), and vice versa; and if $s1$ contains (c-state[p1, q1, n1] : cs_1), then $s2$ contains (c-state[p1, q1, n1] : cs_2), and vice versa).

Example 4 (*A configuration of the 3-process & 4-channel system*) *Let us consider the 3-process & 4-channel system described as the following.*

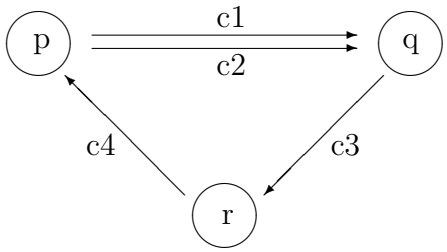

Figure 5.1 The 3-process & 4-channel system

Then the global sate $s \in S_{\mathcal{UDS}}$ of of the $\mathcal{UDS}$ described in Figure 5.1 can be expressed as a configuration as follows:

$s \triangleq$ ((p-state[p] : ps_1) (p-state[q] : ps_2) (p-state[r] : ps_3)
(c-state[p, q, 0] : cs_1) (c-state[p, q, 1] : cs_2)
(c-state[q, r, 0] : cs_3) (c-state[r, p, 0] : cs_4))

5.2.2 State Transitions for a $\mathcal{UDS}$

Taking an arbitrary process, an incoming channel and an outgoing channel of the process from a $\mathcal{UDS}$, we have the following three state transitions. Note that the process may not have any incoming channel or outgoing channel, and $(s, s') \in T$ is represented in the form of $s \Rightarrow s'$.

- **Change of Process States.** The process may change its state without sending or receiving any message, which can be described by the following transition rule:

  $\mathcal{TR}$[chgStt] :

  (p-state[P] : PS)

  $\Rightarrow$

  (p-state[P] : PS')

- **Sending of Messages.** The process may put a message into one of its outgoing channels if it has some outgoing channels and may change its state accordingly, which can be described by the following transition rule:

  $\mathcal{TR}$[sndMsg] :

  (p-state[P] : PS) (c-state[P, Q, N] : Ms)

$\Rightarrow$

(p-state[P] : PS') (c-state[P, Q, N] : $enqueue(Ms, M)$)

where $enqueue(Ms, M)$ means that putting the message M into the end of the FIFO queue Ms.

- **Receipt of Messages.** The process may get a message from one of its nonempty incoming channels if it has some nonempty incoming channels and may change its state accordingly, which can be described by the following transition rule:

  $\mathcal{TR}$[recMsg] :

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

  $\Rightarrow$

  (p-state[P] : PS') (c-state[Q, P, N] : Ms)

Definition 8 ($M_{\mathcal{UDS}}$) *For a state machine $M \triangleq \langle S, I, T \rangle$ formalizing a $\mathcal{UDS}$, $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$, where*

1. *$S_{\mathcal{UDS}}$ is a set of terms whose sorts are Config such that all terms in the set are equal up to process and channel states;*
2. *$I_{\mathcal{UDS}}$ is a subset of $S_{\mathcal{UDS}}$;*
3. *$T_{\mathcal{UDS}}$ is defined from those three transition rules $\mathcal{TR}$s described above for each process, each incoming channel and each outgoing channel of the process.*

5.3 Modeling a $\mathcal{UDS}$ Superimposed by the $\mathcal{CLDSA}$ as a State Machine

5.3.1 State Expression for a $\mathcal{UDS}$ Superimposed by the $\mathcal{CLDSA}$

As we described in Chapter 4, a global state of a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ is expressed as a soup of meta configuration components:

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

where **base-state**(bc) for the state of a $\mathcal{UDS}$, **start-state**(sc) for the start state, **snapshot**(ssc) for the snapshot, **finish-state**(fc) for the finish state, and **control**(ctl) for specifying the behaviors of the $\mathcal{CLDSA}$. Those are called meta configuration components and the corresponding sort is MCComp. bc, sc, ssc and fc, whose sort are Config, are a soup (associative-commutative collection) of p-state and c-state observable components. And ctl, whose sort is CtlConfig, is a soup of the following observable components:

- (**cnt** : n): n is the number of processes that have not yet completed the $\mathcal{CLDSA}$. When n becomes 0, a distributed snapshot has been taken.

– (**prog**[p] : pg): pg is the progress of a process p, indicating that the process has not yet started, has started, or completed the $\mathcal{CLDSA}$. $notYet$, $started$, or $completed$ can be used to represent them respectively.

– (#**ms**[p] : n): n is the number of incoming channels to a process p from which markers have not yet been received. When n becomes 0, p has received markers from all of its incoming channels, implying that p completes the $\mathcal{CLDSA}$ if p has one or more incoming channels. Note that p may not have any incoming channels and then n may be 0 even in initial states.

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

When the content in each prog component is *completed*, a snapshot of the whole system has been taken. To make the system specification less complicated, the component cnt is used, although it seems to be redundant.

base-state(bc) **start-state**(sc) **snapshot**(ssc) **finish-state**(fc) **control**(ctl) is called a meta configuration. And the corresponding sort is MConfig. Initially, all of sc, ssc and fc are $empConfig$. If sc is not $empConfig$, it means that a distributed snapshot has started being taken. If fc is not $empConfig$, it means that a distributed snapshot has been taken and then ssc is the snapshot.

Example 5 (*A meta configuration of the 3-process & 5-channel system*) *Let us consider the 3-process & 5-channel system described as the following.*

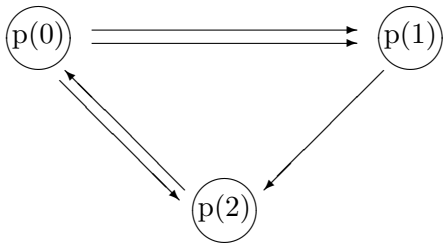

Figure 5.2 The 3-process & 5-channel system

Initially, we suppose that one channel from p(0) to p(1) consists of one message M , and the other channels are empty. Then the global sate $s \in S_{\mathcal{CLDSA}}$ of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ described in Figure 5.2 can be expressed as a meta configuration as follows:

$s \triangleq$ **base-state**((p-state[p(0)] : ps_0) (p-state[p(1)] : ps_1) (p-state[p(2)] : ps_2)
(c-state[p(0), p(1), 0] : (M | $empChan$))
(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($empConfig$)
snapshot($empConfig$)

finish-state($empConfig$)
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)] : $notYet$))

5.3.2 State Transitions for a $\mathcal{UDS}$ Superimposed by the $\mathcal{CLDSA}$

What each process in a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ does is as follows:

1. The process may change its state without sending or receiving any message.
2. The process may put a message into one of its outgoing channels if it has some outgoing channels and changes its state accordingly.
3. The process may get a message 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 $\mathcal{UDS}$ part.

In the rest of the report, BC, SC and SSC are variables of sort Config, CC is a variable of sort CtlConfig, P and Q are variables of sort Pid, PS and PS' are variables of sort PState, N is a variable of sort Nat, Ms and Ms' are variables to represent sequences of messages, and NzN and NzN' are variables of sort NzNat. Note that NzNat for natural numbers except 0, such as $1, 2, 3, \ldots$.

For a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$, we can have the following state transitions.

- **Change of Process States.** For the first case, it can be described by the following transition rule:

  $\mathcal{TR}'$[chgStt] :

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

  $\Rightarrow$

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

- **Sending of Messages.** For the second case, it can be described by the following transition rule:

  $\mathcal{TR}'$[sndMsg] :

  base-state((p-state[P] : PS) (c-state[P, Q, N] : Ms) BC)

  $\Rightarrow$

  base-state((p-state[P] : PS') (c-state[P, Q, N] : $enqueue(Ms, M)$) BC)

- **Receipt of Messages.** For the third case, we need to take into account four subcases:

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

     $\mathcal{TR}'$[recMsg&notYet&$\sim$done] :

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

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

     $\mathcal{TR}'$[recMsg&completed&done] :

     base-state((p-state[P] : PS) (c-state[Q, P, N] : $M \mid Ms$) BC)
     control((prog[P] : *completed*) CC)
     $\Rightarrow$
     base-state((p-state[P] : PS') (c-state[Q, P, N] : Ms) BC)
     control((prog[P] : *completed*) CC)

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

     $\mathcal{TR}'$[recMsg&started&$\sim$done] :

     base-state((p-state[P] : PS) (c-state[Q, P, N] : $M \mid Ms$) BC)
     snapshot((c-state[Q, P, N] : Ms') SSC)
     control((prog[P] : *started*) (done[Q, P, N] : $false$) CC)
     $\Rightarrow$
     base-state((p-state[P] : PS') (c-state[Q, P, N] : Ms) BC)
     snapshot((c-state[Q, P, N] : $enqueue(Ms', M)$) SSC)
     control((prog[P] : *started*) (done[Q, P, N] : $false$) CC)

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

     $\mathcal{TR}'$[recMsg&started&done] :

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

control((prog[P] : *started*) (done[Q, P, N] : *true*) *CC*)
⇒
base-state((p-state[P] : PS') (c-state[Q, P, N] : Ms) *BC*)
control((prog[P] : *started*) (done[Q, P, N] : *true*) *CC*)

- **Record of Process States.** If a process has already received a marker, namely that it has already recorded its state as well. Hence, we only need to take into account the case in which a process has not yet received any markers. When a process records its state in the case, the case is split into two subcases:

  1. The process globally initiates the $\mathcal{CLDSA}$, namely that it is the first process that records its state in the system. This case is further split into three subcases:

     (a) The $\mathcal{UDS}$ only consists of the process.
     $\mathcal{TR}'$[start&cnt=1&#ms=0] :
     base-state((p-state[P] : PS))
     start-state($empConfig$)
     snapshot($empConfig$)
     finish-state($empConfig$)
     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.
     $\mathcal{TR}'$[start&cnt>1&#ms=0] :
     base-state((p-state[P] : PS) *BC*)
     start-state($empConfig$)
     snapshot($empConfig$)
     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 sd stands for symmetric difference, takes two natural numbers x, y, and returns $(x - y)$ 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.

$\mathcal{TR}'$[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)
where bcast(BC, P, marker) puts one marker into each of its outgoing channels, and inchans(BC, P) initializes the states of its all incoming channels.

2. The process does not, namely that there exists another process that has globally initiated the $\mathcal{CLDSA}$. 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 have not completed the $\mathcal{CLDSA}$.
   $\mathcal{TR}'$[record&cnt=1&#ms=0] :
   base-state((p-state[P] : PS))
   start-state(SC)
   snapshot(SSC)
   finish-state($empConfig$)
   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))
   control((prog[P] : *completed*) (cnt : 0) (#ms[P] : 0) CC)
   if $SC \neq empConfig$

   (b) The process does not have any incoming channels, and there are some other processes that have not completed the $\mathcal{CLDSA}$.
   $\mathcal{TR}'$[record&cnt>1&#ms=0] :
   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((p-state[P] : PS) SSC)
   control((prog[P] : *completed*) (cnt : $sd(NzN', 1)$) (#ms[P] : 0) CC)
   if $(NzN' > 1) \wedge (SC \neq empConfig)$

(c) The process has some incoming channels.

$\mathcal{TR}'$[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 $SC \neq empConfig$

- **Receipt of Markers.** When a process receives a marker from an incoming channel, we first need to take into account the following two cases:

1. The process has not yet started the $\mathcal{CLDSA}$. 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 the $\mathcal{CLDSA}$ except for the process, which implies that the process does not have any outgoing channels.

$\mathcal{TR}'$[recMkr&notYet&#ms=1&cnt=1] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : $marker \mid Ms$) 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] : Ms) BC)
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)

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

$\mathcal{TR}'$[recMkr&notYet&#ms=1&cnt>1] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : $marker \mid Ms$) 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] : Ms)
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 $NzN > 1$

(c) The process has more than one incoming channel.
$\mathcal{TR}'$[recMkr&notYet&#ms>1&cnt>1] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : $marker \mid Ms$) 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] : Ms)
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 $NzN' > 1$

2. The process has already started the $\mathcal{CLDSA}$. This case is further split into three subcases:

(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 the $\mathcal{CLDSA}$ except for the process.
$\mathcal{TR}'$[recMkr&started&#ms=1&cnt=1] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : $marker \mid Ms$) 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] : Ms) BC)
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)

(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 the $\mathcal{CLDSA}$.
$\mathcal{TR}'$[recMkr&started&#ms=1&cnt>1] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : $marker \mid Ms$) 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] : Ms) BC)
control((prog[P] : *completed*) (#ms[P] : 0) (cnt : $sd(NzN, 1)$)
(done[Q, P, N] : *true*) CC)

if $NzN > 1$

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

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

5.3.3 The Function $\mathcal{CL}$

In this section, we will give a definition of the function $\mathcal{CL}$, which takes a state machine $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$, and returns another state machine $M_{\mathcal{CLDSA}} \triangleq \langle S_{\mathcal{CLDSA}}, I_{\mathcal{CLDSA}}, T_{\mathcal{CLDSA}} \rangle = \mathcal{CL}(M_{\mathcal{UDS}})$. Note that $M_{\mathcal{UDS}}$ represents the $\mathcal{UDS}$ and $M_{\mathcal{CLDSA}}$ represents the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$. Now let us consider how to model a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ as a state machine. It should include the $\mathcal{UDS}$ part discussed in §5.2, and another part to reflect the behaviors of the $\mathcal{CLDSA}$.

Now, we give the definition of the function $\mathcal{CL}$ as follows:

Definition 9 ($\mathcal{CL}(M_{\mathcal{UDS}})$) *For a state machine $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$ formalizing a $\mathcal{UDS}$, we have $M_{\mathcal{CLDSA}} = \mathcal{CL}(M_{\mathcal{UDS}}) \triangleq \langle \mathcal{CL}_{State}(S_{\mathcal{UDS}}), \mathcal{CL}_{Init}(I_{\mathcal{UDS}}), \mathcal{CL}_{Trans}(T_{\mathcal{UDS}}) \rangle$, where*

1. *$\mathcal{CL}_{State}(S_{\mathcal{UDS}}) \triangleq \{$base-state$(bc)$ start-state(sc) snapshot(ssc) finish-state(fc) control(ctl) $\mid bc \in S_{\mathcal{UDS}}$, $sc \in$ Config, $ssc \in$ Config, $fc \in$ Config, $ctl \in$ CtlConfig$\}$, where Config and CtlConfig are used as the sets of terms whose sorts are Config and CtlConfig, respectively;*

2. *$\mathcal{CL}_{Init}(I_{\mathcal{UDS}}) \triangleq \{$base-state$(bc)$ start-state$(empConfig)$ snapshot$(empConfig)$ finish-state$(empConfig)$ control(ctl) $\mid bc \in I_{\mathcal{UDS}}$, $ctl \in$ CtlConfig$\}$;*

3. *$\mathcal{CL}_{Trans}(T_{\mathcal{UDS}}) \subseteq \mathcal{CL}_{State}(S_{\mathcal{UDS}}) \times \mathcal{CL}_{State}(S_{\mathcal{UDS}})$, which can be induced by the transition rules $\mathcal{TR}'$s described in* §5.3.2.

Initially, all of sc, ssc and fc are $empConfig$. This is the reason why empConfig is used in **start-state**, **snapshot** and **finish-state** in the definition of $\mathcal{CL}_{Init}(I_{\mathcal{UDS}})$. If sc is not $empConfig$, it means that a distributed snapshot has started being taken. If fc is not $empConfig$, it means that a distributed snapshot has been taken and then ssc is the snapshot.

However, the **control**(ctl) part in the definition $\mathcal{CL}_{Init}(I_{\mathcal{UDS}})$ is not precise, just saying that $ctl \in$ CtlConfig. We know that for an initial state of a $\mathcal{UDS}$, there are some

constraints to obtain the *ctl* part, which is not an arbitrary soup of cnt, prog, #ms, and done observable components.

Example 6 (*A meta configuration of the 3-process & 5-channel system*) *Let us consider the 3-process & 5-channel system described as the following.*

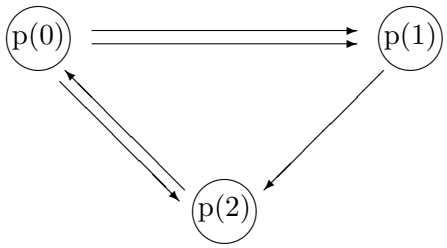

Figure 5.3 The 3-process & 5-channel system

Initially, we suppose that one channel from p(0) to p(1) consists of one message M , and the other channels are empty. Then the global sate $s \in S_{\mathcal{CLDSA}}$ of the $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ described in Figure 5.3 can be expressed as a meta configuration as follows:

$s \triangleq$ **base-state**((p-state[p(0)] : ps_0) (p-state[p(1)] : ps_1) (p-state[p(2)] : ps_2)
(c-state[p(0), p(1), 0] : (M | $empChan$))
(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($empConfig$)
snapshot($empConfig$)
finish-state($empConfig$)
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)] : $notYet$))

From the contents (a soup of observable components) of **base-state** and **control**, we can have some relations between them as follows:

1. The value of cnt observable component should be the number of the processes;
2. The value of #ms observable component should be the number of the incoming channels of each process;
3. The value of done observable component should be $false$ for each channel;
4. The value of prog observable component should be $notYet$ for each process.

Moreover, to obtain the *ctl* part, we should contain one cnt, #ms for each process, done for each channel and prog for each process.

For a state machine $M_{\mathcal{CLDSA}} \triangleq \langle S_{\mathcal{CLDSA}}, I_{\mathcal{CLDSA}}, T_{\mathcal{CLDSA}} \rangle$ formalizing an underlying distributed system superimposed by the $\mathcal{CLDSA}$, we have the following definitions for *start-state*, *snapshot*, $finish\text{-}state$ and *terminates*.

Definition 10 (*start-state*, *snapshot*, *finish-state* and *terminates*) *For a state machine $M_{\mathcal{CLDSA}} \triangleq \langle S_{\mathcal{CLDSA}}, I_{\mathcal{CLDSA}}, T_{\mathcal{CLDSA}} \rangle$, $\forall s \in S_{\mathcal{CLDSA}}$,*

- $start\text{-}state(s) \triangleq sc.$
- $snapshot(s) \triangleq ssc.$
- $finish\text{-}state(s) \triangleq fc.$
- $terminates(s) \triangleq (fc \neq empConfig).$

5.4 A Way to Express the $\mathcal{DSR}$ Property

The following is the definition that a state s'' is reachable from a state s' in a path π w.r.t. a state machine M:

Definition 11 ($M, \pi \models isReachable(s', s'')$) *For a state machine $M \triangleq \langle S, I, T \rangle$, a path π w.r.t. M and states $s', s'' \in S$, $M, \pi \models isReachable(s', s'')$ iff $\exists i, j \in Nat\ ((i \leq j) \wedge (s' = \pi_i) \wedge (s'' = \pi_j))$.*

The following is the definition that a state s'' is reachable from a state s' w.r.t. a state machine M:

Definition 12 ($M \models isReachable(s', s'')$) *For a state machine $M \triangleq \langle S, I, T \rangle$ and states $s', s'' \in S$, $M \models isReachable(s', s'')$ iff $\exists \pi \in \Pi\ (M, \pi \models isReachable(s', s''))$.*

In other words, a state s'' is said to be reachable from a state s' iff there is a sequence of states in which s' occurs earlier than or at the same time as s''. For the latter case, s'' is the same state as s'.

Our way to express the $\mathcal{DSR}$ property has been affected by Chandy-Misra's, in which the $\mathcal{CLDSA}$ was analyzed based on the formal method UNITY [CM88]. Now we give our definition of the $\mathcal{DSR}$ property as follows:

Definition 13 ($\mathcal{DSR}$ *Property*) *For a state machine $M_{\mathcal{UDS}} \triangleq \langle S_{\mathcal{UDS}}, I_{\mathcal{UDS}}, T_{\mathcal{UDS}} \rangle$, $\forall s \in \mathcal{CL}_{State}(S_{\mathcal{UDS}})$,* **if** *$((\mathcal{CL}(M_{\mathcal{UDS}}) \models terminates(s))$,* **then** *$(M_{\mathcal{UDS}} \models isReachable(s_1, s_*) \wedge M_{\mathcal{UDS}} \models isReachable(s_*, s_2))$, where*

1. $s_1 \triangleq start\text{-}state(s);$
2. $s_* \triangleq snapshot(s);$
3. $s_2 \triangleq finish\text{-}state(s).$

Chapter 6

Conclusions

6.1 Contributions

What we have done are the following.

- Existing Studies:
  - Learnt some basic technical knowledge such as distributed systems, state machine, model checking and Maude specification language;
  - Learnt the $\mathcal{CLDSA}$ and the $\mathcal{DSR}$ property;
  - Learnt an existing study [OH12], in which a $\mathcal{UDS}$ superimposed by the $\mathcal{CLDSA}$ has been formally specified in Maude and model checked w.r.t. the $\mathcal{DSR}$ property with the Maude search command.

- Original Works:
  - Realized the expression of the $\mathcal{DSR}$ property in the existing study [OH12] does not respect the property (written in English) in the original paper [CL85];
  - Given the definition of the function $\mathcal{CL}$;
  - Found a way [ZOZ15] to faithfully express the $\mathcal{DSR}$ property.

6.2 Future Work

Until now, we still have not model checked the $\mathcal{DSR}$ property defined in this report in a proper way. There are several things to do as future issues.

1. To specify the system according to the expression of the $\mathcal{DSR}$ property (Definition 13).

2. To prove that the function $\mathcal{CL}$ preserves the behaviors of a state machine $M_{\mathcal{UDS}}$ formalizing a $\mathcal{UDS}$.

For the second issue, we need to prove that reachability is preserved from $M_{\mathcal{UDS}}$ to $\mathcal{CL}(M_{\mathcal{UDS}})$. What to prove is $\forall s', s'' \in S_{\mathcal{UDS}}$ ($M_{\mathcal{UDS}} \models isReachable(s', s'')$ iff $\mathcal{CL}(M_{\mathcal{UDS}})$ $\models isReachable(base\text{-}state(s'), base\text{-}state(s''))$). In other words, we need to prove that the reachability in $M_{\mathcal{UDS}}$ is exactly the same as the reachability in $\mathcal{CL}(M_{\mathcal{UDS}})$. The simulation-based invariant verification method [OF07b] is one candidate way to prove it.

Appendix A

Specification of the $\mathcal{CLDSA}$ 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 .

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

  ---delMC
  eq delMC(empChan) = empChan .
  eq delMC(marker | C) = delMC(C) .
  eq delMC(T | C) = T | delMC(C) [owise] .
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 nth 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 .

  --- bcast
  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] .

  --- inchans
  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] . --- no OC!!!

  --- delM
  eq delM(empConfig) = empConfig .
  eq delM((c-state[P,Q,N] : C) CF)
     = (c-state[P,Q,N] : delMC(C)) delM(CF) .
  eq delM(OC CF) = OC delM(CF) [owise] .
endfm
```

```
***(
red (p-state[p(0)] : t(1)) (p-state[p(1)] : noToken) (p-state[p(2)
   ] : noToken)
    (c-state[p(0), p(1), 0] : (t(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) .
)


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

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

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

  var MOC : MCComp .
  eq MOC MOC = MOC .
endfm


***(
red 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)) .
)


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
   \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) --> p(1), p(1) --> 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 : -> 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
   \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) --> p(1), p(1) --> 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

***
eof
***
```

Appendix B

Verification of the $\mathcal{DSR}$ Property

```
***
*** Verification 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

[AP10] X. An and J. Pang. Model checking round-based distributed algorithms. pages 127–135. 15th IEEE ICECCS, 2010.

[BBF+10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2010.

[BM11] K. Bae and J. Meseguer. State/event-based ltl model checking under parametric generalized fairness. volume 6860 of *LNCS*, pages 132–148. 23rd CAV, Springer, 2011.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Rovere, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. volume 2404 of *LNCS*, pages 359–364. Springer, Heidelberg, 2002.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[CGJ+01] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. informatics - 10 years back. 10 years ahead. volume 2000 of *LNCS*, pages 176–194. Springer-Verlag, 2001.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, 1999.

[CKNZ12] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. volume 7682 of *LNCS*, pages 1–30, 2012.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed system. pages 63–75. ACM TOCS 3, 1985.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CS01] E. M. Clarke and H. Schlingloff. *Model checking*. Handbook of Automated Reasoning. Elsevier, 2001.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. volume 3114 of *LNCS*, pages 496–500. Springer, Heidelberg, 2004.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. pages 1–4. Inf. Proc. Lett. 11, 1, 1980.

[Gho06] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall/CRC, 2006.

[GL05] J. Goguen and K. Lin. Specifying, programming and verifying with equational logic. *We Will Show Them!*, 2:1–38, 2005.

[HH82] G. Huet and J. Hullot. Proofs by induction in equational theories with constructors. *Computer and System Sciences*, 25(2):239–266, 1982.

[Hol04] G. J. Holzmann. The spin model checker - primer and reference manual. Addison-Wesley, 2004.

[KS08] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[MC82] J. Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. pages 37–43. ACM Trans. Program. Lang. Syst. 4, 1, 1982.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. volume 1119 of *LNCS*, pages 331–372. 7th CONCUR, Springer, 1996.

[OF07a] K. Ogata and K. Futatsugi. Comparison of maude and sal by conducting case studies model checking a distributed algorithm. Fundamentals E90-A, pages 1690–1703. IEICE Trans, 2007.

[OF07b] K. Ogata and K. Futatsugi. Simulation-based verification for invariant properties in the ots/cafeobj method. Electronic Notes in Theoretical Computer Science 201 (2008), pages 127–154. BCS-FACS Refinement Workshop, 2007.

[OH12] K. Ogata and T. T. P. Huyen. Specification and model checking of the chandy and lamport distributed snapshot algorithm in rewriting logic. volume 7635 of *LNCS*, pages 87–102. 14th ICFEM, 2012.

[TS11] T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. In *Distributed Computing 23*, pages 341–358, 2011.

[ZOZ15] W. J. Zhang, K. Ogata, and M. Zhang. A consideration on how to model check distributed snapshot reachability property. In *IEICE Technical Report*, volume 114, No. 416, ISSN 0913-5685, pages 49–54. IEICE, 2015.