FORMAL SPECIFICATION AND MODEL CHECKING OF DISTRIBUTED SYSTEMS WITH REWRITING LOGIC META-PROGRAMMING FACILITIES

By DOAN, HA THI THU

submitted to
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Written under the direction of
Professor Kazuhiro Ogata

March, 2019

Abstract

In recent decades, key software systems on which human beings heavily rely on are in the form of distributed systems. Such systems are quite complex and thus very hard to design and verify. Model checking is a popular automatic formal technique and highly suitable for verifying distributed systems. Several researchers have attempted to formally analyze and verify distributed systems. However, these are several tough problems or new forms of distributed systems that have not been tackled well by existing approaches (or tools).

Control algorithms are a large important class of distributed algorithms, which deal with significant problems, such as snapshot recording algorithms and checkpointing algorithms. One main characteristic of these algorithms is that they are superimposed on underlying distributed systems. Although some research have been conducted to formally verify control algorithms, all of them directly require humans to specify by hand underlying distributed systems on which control algorithms are superimposed. It is expected to have more general approaches. However, it is challenging to specify control algorithms in almost all existing specification languages for model checkers because it is necessary to treat an underlying distributed system as the data that is handled by these algorithms. Therefore, it is one of the challenging problems to be solved that how to specify control algorithms but not underlying distributed systems on which control algorithms are superimposed is.

Recent advances in distributed computing highlight models and algorithms for autonomous mobile robots that self-organize and cooperate together in order to solve a global objective. Due to the mobility aspect, robot algorithms are often complex, arguably even more complex than classical distributed systems. Designing and analyzing mobile robot algorithms is notoriously difficult.

Rewriting logic is a natural model of computations for concurrency and communication systems. Several specification languages based on rewriting logic, such as Maude, CafeOBJ and Elan have been designed and implemented. Moreover, rewriting logic is a reflective logic that can be faithfully interpreted in itself. Rewriting logic is highly suitable to formalyze distributed algorithms. However, rewriting logic only at the object level may not be powerful enough to precisely specify some distributed algorithms.

This thesis focuses on exploiting Rewriting logic meta-programming facilities to formalize the distributed algorithms that have not been tackled well by existing approaches (or tools), as well as by any methods (or tools) based on rewriting logic at the object level. The aim of the research is to achieve how to tackle two important families of distributed systems, namely control algorithms and mobile robot algorithms, with rewriting logic meta-programming facilities. Theoretically, we have faced the above mentioned problems

by moving from the object level to the meta level, namely that it is necessary to deal with the specifications of underlying distributed systems as data for the specification of a control algorithm and the succinct specification of mobile robot algorithms as data for a translator by which the succinct ones are transformed to those that can be directly treated by Maude.

First of all, we propose a new approach to specifying and model checking control algorithms. Meta-programming technique is applied to the challenges above-mentioned. We have used meta-programs as formal specifications of control algorithms. A control algorithm is specified as a meta-program that takes the specification of an underlying distributed system as an input and generates the specification of the underlying distributed system on which the control algorithm is superimposed (UDS-CA). A control algorithm is only specified at once, and for each underlying distributed system, the specification of the UDS-CA is automatically obtained. Furthermore, we propose a technique that takes the number of each kind of entities used, generate all possible initial states that satisfy some constraints and conduct model checking experiments for all the initial states, which makes it more likely to detect a subtle flaw lurking in a control algorithm or improves the confidence in the correctness of a control algorithm. We have conducted two case studies, which specify and model check snapshot and checkpointing algorithms.

Several classic distributed algorithms have been formally verified with some techniques based on rewriting logic. We aim to obtain similar achievements for distributed mobile robot systems - a new form of distributed systems. We come up with formal specification and model checking based on rewriting logic for mobile robot algorithms. We have conducted two case studies in which we specify and model check an exploration algorithm and a gathering algorithm on rings in Maude. However, no existing specification language is designed for mobile robot algorithms on rings: rings are not directly supported by such languages and specifications of such algorithms are far from the corresponding mathematical descriptions. This is because of the particular symmetries owned by *rings*. Consequently, we need to specify rings by adapting other defined structures, such as *sets* and *sequences*. It, therefore, makes the specification task tedious as well as time-consuming, while the specifications obtained are complicated and lengthy. It is worth providing a specification environment in which rings are directly supported. An environment and a domain-specific language (DSL) for specifying and model checking mobile robot algorithms on rings (or mobile ring robot algorithms) are proposed. First, we develop Maude Ring Specification Environment (Maude RSE), a ring specification environment that explicitly supports ring-shaped networks. Then, we build our DSL, Mobile Ring Robot Maude (MR^2-Maude), on top of Maude RSE. MR^2-Maude makes it possible to specify mobile ring robot algorithms in such a way that the specifications are as close as possible to their mathematical descriptions. One key underlying these tools is pattern matching between ring patterns and ring instances, called "ring pattern matching." The advantages of Maude RSE and MR^2-Maude are demonstrated by case studies analyzing exploration and gathering mobile robot algorithms.

Keywords: control algorithm, mobile robot algorithm, model checking, meta-programming, domain-specific language.

Acknowledgements

I had an unforgettable time during my doctoral program at JAIST with an interesting topic on formal verification of distributed systems. I would like to express my sincere appreciation to who contributed to my thesis and supported me during this amazing journey of my life and this thesis would not have been completed without their supports.

First and foremost, I would like to express my sincerest gratitude to my supervisor, Professor Kazuhiro Ogata, for his support, motivation and encouragement. I was so lucky to have a supervisor who cared so much about my work. He gave me many valuable advices and great ideas that made my work productive and stimulating. His valuable suggestions, comments and guidance kept my research going on track. His deep insights helped me at many stages of my research. He experienced all of the ups and downs of my research and encouraged me to improve my research. He handed me not only the necessary knowledge but also his valuable research experiences. Moreover, I also appreciate his help and kindly guidance in my daily life at JAIST.

I extremely appreciate useful comments from the committee members: Professor Mizuhito Ogawa, Professor Xavier Defago, Professor Tatsuhiro Tsuchiya and Associate Professor Razvan Beuran, who pointed out the limitations of my research in several cases and provided suggestions for improving my work. Specially, I would like to sincerely thank Professor Xavier Defago, who gave me several important advices, suggestions and comments from the beginning of my research to its end. His valuable comments about the principle of distributed systems made my thesis be more consistent and rich.

My grateful gratitude is reserved for Associate Professor François Bonnet for his valuable insights and suggestions. He suggested me several great research ideas on formal verification of mobile robot algorithms, which is the main contribution of my thesis. Without him, the main part of this thesis could not be established. He discussed and gave me many useful advices and valuable comments for my research. I am really grateful for his time and effort for this research. His wide knowledge and logical way of thinking have been of great value for me. I cannot imagine how difficult it would be to complete this work without his guidance.

I would like to give a great thank to Associate Professor Adrián Riesco, who is an expert of Maude. It is a great opportunity for me to have conducted part of my research under his supervision during my internship at Universidad Complutense de Madrid. He gave a big help on my research, including technical support, with his extensive knowledge of Maude. I cannot thank him enough for all he has done to make this work possible. With his valuable supports, a new environment and a domain specific language for mobile robot algorithms

on rings has been successfully designed and implemented. I am amazed that even during his busy schedule, he was always willing to support me with my research. Furthermore, I would like to take this opportunity to thank Professor Narciso Martí-Oliet, who provided me with a chance to join and work at Department of Software Systems and Computation, Universidad Complutense de Madrid and assisted me greatly on many procedures. I am really grateful for working in a professional environment with all cooperations and supports. I owe the members at the department a big thank for helping me a lot of time with many complex procedures for my stay in Madrid.

I would want to thank JAIST Grants, Monbukagakusho Honors Scholarship, NEC C&C Foundation Grants, Tokyo Institute of Technology for financially supporting me during my study, such as the expenses for attending conferences and the expenses for studying abroad. Their financial supports mean a lot to me and encouraged me to do my best in the studying and researching to be worthy with their supports.

A heartfelt thanks to my supportive wonderful family for their constant support and encouragement. Their unconditional support both in my life and my study as well. The love and support that they gave motivated me to work harder and to strive towards my goal. I would not be me today without your love. You are always with me whenever I'm happy or sad. Thank you so much for being with me in the wonderful, but challenging time of my life.

Last but not least, my acknowledgement would be incomplete without thanking my friends both at JAIST and in Madrid, who make my social life a pleasure experience. Their friendship and kindness help me to adapt my self to a new environment and culture. They made me more happy to avoid stress in doing my research. I am indebted to them for their help. Thank to great time with them, I could balance my life and recover my energy for pursuing doctoral degree during these challenging years.

Doan Thi Thu Ha

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Systems

In recent decades, key software systems on which humans heavily rely on are in the form of distributed systems, which consist of multiple nodes (or processes) connected with networks (or channels). Distributed system is an essential form of today software with many significant applications, such as telephone networks and aircraft control systems. A distributed system consists of independent entities that cooperate to solve a problem that cannot be individually solved. The entities are connected and communicate by passing messages though communication networks. Modern applications of distributed systems, such as cloud computing and web search engine, usually provide their services to a large number of customers.

1.1.1 Control Algorithms

Because distributed systems are collections of many individual components, it has raised many global problems, which cannot be detected and handled by a single component and thus require global solutions. Among such problems are detecting termination, detecting distributed deadlock, detecting global stable predicates, recording global states and checkpointing. It is, therefore, necessary to use many non-trivial distributed algorithms, such as termination detection algorithms, deadlock detection algorithms, global stable predicate detection algorithms, snapshot recording algorithms and checkpointing algorithms. These algorithms need to be executed in order to monitor underlying application executions or to perform various auxiliary functions. They are essential to distributed systems. One main characteristic of these algorithms is that a control algorithm execution is superimposed on an underlying application execution and in many cases does not interfere with the application execution (Fig 1.1). Said differently, they run concurrently with underlying distributed systems (UDSs). Such distributed algorithms are called Control algorithms [43]. An underlying distributed system is regarded and treated as data by control algorithms.

Based on our surveys of [43] and [60], there are around 30% problems tackled by control

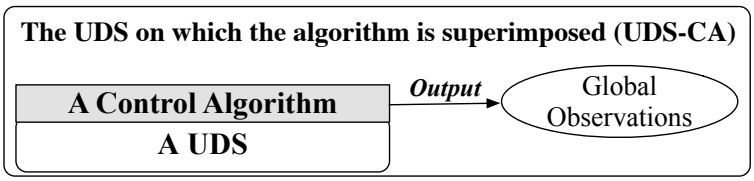


Figure 1.1: The control algorithm execution is superimposed on an underlying application execution.

algorithms. Control algorithms, therefore, is a large and important class of distributed algorithms.

1.1.2 Mobile Robot Algorithms

Traditionally, distributed computing has focused on systems that consist of immobile entities connected with fixed communication networks in which communication is explicit. However, distributed computing is extending to solve problems relevant to a distributed mobile environment where entities are mobile and communication might not be explicit. Mobile distributed computing has become the heart of various different systems, such as software mobile agent in communication networks, mobile sensor networks, swarms and robotic networks. Motivated by the various tasks that can be performed by autonomous mobile robots, many researchers have been investigating the field of autonomous mobile robot. Recent developments in theoretical computer science (especially in distributed computing) focus on models and algorithms for autonomous mobile robots that self-organize and cooperate in order to achieve global goals. Autonomous mobile robots have been proposed for several promising applications, such as working in damaged environments. The seminal model introduced by Suzuki and Yamashita [40] proposes a distributed system of k robots that have low capacities: identical (they are indistinguishable and all execute the same algorithm), oblivious (they have no memory of their past actions), and disoriented (they share no common orientation). Moreover, the robots do not communicate by sending or receiving messages, but have the ability to sense their environment and see the relative positions of the other robots. Researchers have proposed formal models for these systems and have designed algorithms for solving some predefined tasks. Most papers focus on the computability of mobile robots; one of the main goals is usually to find the weakest assumptions (on robots and/or model), such as synchrony, multiplicity detection, and chirality that might make a problem solvable or unsolvable. In this context, various models and algorithms have been proposed to solve particular problems. There exist several different models, but they can be classified in two main classes: (i) discrete models in which movement is restricted to a graph [10, 11, 22] and (ii) continuous models in which entities move on a continuous space [34, 40, 64]. For both cases, a large variety of tasks have been considered, especially gathering and exploration. In discrete models, robots perform their activities in specific shape networks, such as rings and grids. They can be observed only at specific discrete locations of these networks.

What and how problems can be solved by a group of autonomous mobile robots on

ring-shaped networks is an important topic in the area, as shown by the large number of algorithms that have been proposed: e.g. the papers [10, 24, 26, 27, 33, 44] propose algorithms for exploration on ring, robot gathering on rings is solved in [11, 21, 23, 39, 41, 55], and some other problems are solved in [22, 59]. In this thesis, we focus on discrete models where robots move on a ring-shaped network.

1.2 Formal Verification of Distributed Systems

Distributed systems are quite complex and thus very hard to design and verify. Formal verification is one of the main approaches to distributed system verification and model checking [4, 32] is a popular automatic formal technique and highly suitable for verifying such systems. Several researchers [3, 42, 65, 66] have attempted to formally analyze and verify distributed systems (or algorithms) and many tools are developed to that purpose. However, these are several tough problems that have not been tackled well by existing approaches and more new forms of distributed systems that require new methods (or tools) to deal with.

Control algorithms have raised a challenge to verification by model checking because such algorithms cannot be specified independently of the underlying systems. Although some control algorithms have been specified and model checked in [7, 12, 57], all of them directly require humans to specify by hand the underlying systems on which control algorithms are superimposed (Fig. 1.2(a)). When the same control algorithm is to be verified for different underlying systems (or different variants of the same underlying system), we need to write new specifications of the underlying systems on which the algorithm is superimposed. This consumes time and exhausts specifiers due to repetitions. We may, therefore, want to propose a new approach such that we only need to specify a control algorithm at once and the specification of the system on which a control algorithms is superimposed (UDS-CA) is automatically generated from the specification of an underlying system. However, it is challenging to specify control algorithms in almost all existing specification languages for model checkers, such as PROMELA [37], because it is necessary to treat an underlying system as data handled by control algorithms. Moreover, two different systems (an underlying distributed system and the UDS-CA) may be taken into account to model check some properties for control algorithms, while almost all existing model checkers (to our knowledge), such as SPIN [37], NuSMV [14], and TLC [45], do not support it.

Designing and analyzing mobile robot algorithms is notoriously difficult. In the literature, the correctness of such algorithms relies on handmade mathematical proofs, which are error-prone. Most of these proofs are handmade and may consist of a large number of cases, especially when the algorithm is given explicitly as a set of transition rules (*e.g.* [10]) in opposition to a more abstract algorithm where movements are implicitly given by some mathematical rules (*e.g.* [22]). It is not easy and not trustful to check the correctness of these algorithms only by hand. We believe that the research field of distributed mobile robots is now mature; some main models have emerged and have been adopted by the community. It is time to study how to automatically verify such algorithms and this

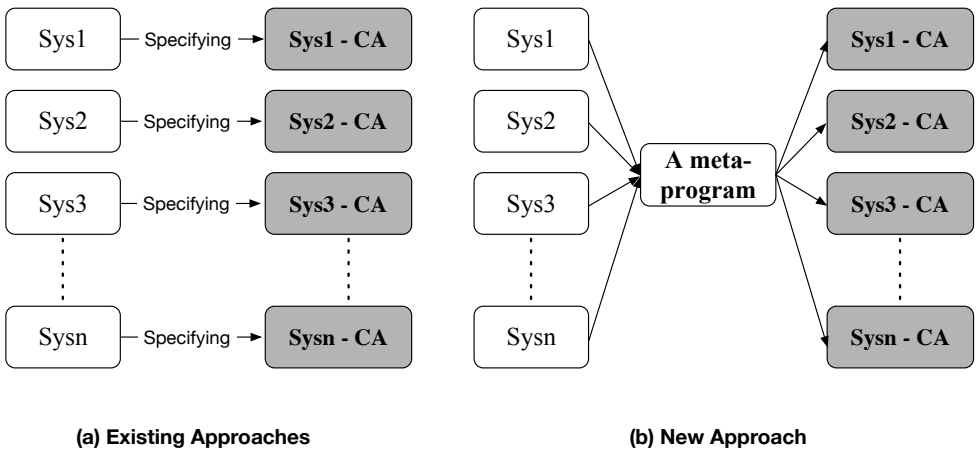


Figure 1.2: The existing approaches and our new approach to specifying *control algorithms*. Sys1, ..., Sys$n$ are underlying distributed systems and Sys1-CA, ..., Sys$n$-CA are the underlying distributed systems on which a control algorithm is superimposed.

thesis presents an example of automatic verification. However, due to the mobility aspect, mobile robot algorithms are often complex, arguably even more complex than classic distributed systems. This inherent difficulty explains probably the limited number of attempts in obtaining formal verifications. The untruthfulness of handmade mathematical proofs has been pointed out in [5, 9, 28, 29]. Formal, automatic techniques could help us increase the confidence of the existing algorithms/proofs, as shown in [5, 9, 19, 28, 29]. For discrete models, model-checking has been proven useful to find errors in the proposed algorithms [9, 28, 29]. However, ring discrete models are not well supported by any existing specification languages, such as SPIN [37], DVE [6], and Maude [16]. This is because of the particular symmetries owned by *rings*. Consequently, the specifiers, such as Berard *et al.* [9] and Doan *et al.* [28, 29], need to specify rings by adapting other defined structures, such as *sets* and *sequences*. For instance, Doan *et al.* [28] need to use commutative binary operators to specify rings in Maude. It, therefore, makes the specification task tedious as well as time-consuming, while the specifications obtained are complicated and lengthy.

Recent publications, such as [5], have attempted to include formal proofs. Such proofs require the hard efforts of researchers to have the knowledge for specifying an algorithm in a particular specification language. However, no existing specification language is designed for mobile robot algorithms on rings: rings are not directly supported by such languages and the specifications of such algorithms are far from the corresponding mathematical descriptions. Therefore, it is worth providing a specification environment in which rings are directly supported. The environment would allow users to concentrate on mobile ring robot algorithms. It is also worth designing a domain-specific language dedicated to mobile ring robot algorithms on top of the environment. The domain-specific language allows users to specify mobile ring robot algorithms such that the specifications are as close as possible to their mathematical descriptions. It also provides predefined LTL

formulas as well as atomic propositions to model check that such algorithms enjoy the desired properties.

Rewriting logic is a natural model of computations for concurrency, parallel and communication systems. Several specification languages based on rewriting logic, such as Maude, CafeOBJ and ELAN have been designed and implemented. Moreover, rewriting logic is a reflective logic that can be faithfully interpreted in itself. This makes some of these languages, such as Maude able to provide meta-programming facilities. Rewriting logic is highly suitable to formalyze distributed algorithms. This is demonstrated by many researches on formalization of distributed systems based on the rewriting logic framework. However, rewriting logic only at the object level may not be powerful enough to precisely specify some tough distributed algorithms.

1.3 Contributions

This thesis focuses on exploiting rewriting logic meta-programming facilities to formalize the distributed algorithms that have not been tackled well by existing approaches (or tools), as well as by any methods (or tools) based on rewriting logic at the object level. The aim of the research is to achieve the formal specification and model checking of distributed systems with rewriting logic meta-programming facilities. Theoretically, we have faced the above mentioned problems by moving from the object level to the meta level. We have manipulated rewriting logic as a reflective logic and used meta-programming techniques to specify distributed algorithms.

1.3.1 A New Approach to Specifying and Model Checking Control Algorithms

We carefully take into account the important aspect of control algorithms — they run concurrently with an underlying distributed system. We have investigated one possible solution that is to apply meta-programming techniques to the challenges above-mentioned. A meta-program is a program that takes a program (or specification) as an input and performs some useful computations, such as analyzing the program and transforming the program into another. Metaprogramming is very powerful with many important applications. Translators and debuggers, for instance, are such applications.

As we have mentioned, when the same control algorithm is to be verified for different underlying distributed systems (or different variants of the same underlying distributed system), we need to write new specifications of the underlying distributed systems on which the control algorithm is superimposed (UDS-CA). We present an approach that aims at avoiding this repetition by specifying a control algorithm as a transformation that converts the specification of an underlying distributed system into one that combines both the underlying distributed system and the algorithm. A control algorithm is specified as a meta-program that takes the specification of an underlying distributed system as an input and generates the specification of the UDS-CA. A control algorithm is specified at once, and for each underlying distributed system, the specification of the UDS-CA is

automatically obtained. This is depicted in Fig. 1.2(b). The model checking is conducted at the meta level as well. The meta level makes it possible to model check the properties that involve both an underlying distributed system and the UDS-CA. The distributed snapshot reachability property is one of these properties. We explain how to model check it in a case study presented in Section 4.3.1.

Model checking is a verification technique that explores all possible system executions and checks whether a desired property that should be satisfied by an algorithm is satisfied. It is necessary to provide the initial states of an underlying distributed system to model check that the UDS-CA enjoys a desired property. One main challenge is that counterexamples may be found for some initial states, while they may not be for others. Even fixing the number of every kind of entities, used in an underlying distributed system, there may be more than one underlying distributed system due to many options, such as network topologies, such as mobile support stations and mobile hosts in a mobile checkpointing algorithm. Facing the challenge, then, one piece of our work is to come up with a verification technique that generate all possible initial states of an underlying distributed system for a fixed number of each kind of entities. Because too many initial states could be generated, however, we use some constraints to make the number of initial states moderate. One possible constraint is that mobile support stations are strongly connected with a reliable wired network. We demonstrate the usefulness by reporting on a case study in which a counterexample is found for some specific initial states but not for the other initial states, detecting a subtle flaw lurking in a mobile checkpointing algorithm.

We have conducted two case studies in which the approach is utilized to specify and model check a snapshot algorithm and a checkpointing algorithm. Two main contributions of this work are:

(1) *Specification*: It is an approach to specifying control algorithms as meta-programs, which allows us:

- to specify a control algorithm once and automatically generate the specification of the UDS-CA for each underlying distributed system;
- to faithfully specify and model check properties that involve both an underlying distributed system and the UDS-CA.

(2) *Model checking*: We propose a technique that takes the number of each kind of entities used, generate all possible initial states and conduct model checking experiments for all the initial states, which makes it more likely to detect a subtle flaw lurking in a control algorithm. Our model checking has found two subtle errors in a checkpointing algorithm.

1.3.2 An Environment and a Domain-Specific Language for Specifying and Model Checking Mobile Ring Robot Algorithms

Several classic distributed algorithms have been formally verified with methods based on rewriting logic, while few studies have been conducted to formally verify distributed mobile

robot systems — a new form of distributed systems — with those based on rewriting logic. Our work implies that the rewriting logic framework is powerful and flexible to successfully express a new form of distributed system.

Formal Analyzing Mobile Robot Algorithms in Maude

We come up with formal specification and model checking based on writing logic to mobile robot algorithms. We present how to formalize a mobile robot algorithm as a state machine and then specify the state machine in Maude, a language and a system supporting executable specification and declarative programming in rewriting logic. We have demonstrated in [28, 30, 31] that Maude allows us to specify distributed algorithms/systems more succinctly than others. For instance, it supports *associative* and *commutative* operator attributes that are very necessary to concisely specify mobile robot algorithms as shown in [28]. A case study of how to specify and model check a given robot exploration algorithm is conducted. A formal model of a system consisting of three robots on the ring shaped network is given. The main work consists in using the model checking approach to automatic verification of a perpetual ring exploration algorithm. Rewriting logic makes it possible to naturally specify dynamic systems, and the Maude system has an LTL model checker. The two significant properties which the algorithm should satisfy have been specified as Linear Temporal Logic (LTL) formulas [38]. We then model check the two properties for the algorithm with the LTL model checker. As the result of our model checking for the algorithm, a counterexample has been found. This states that the algorithm is not correct since it does not satisfy the two properties.

Furthermore, we have proposed a formal model for mobile robot algorithms on anonymous ring shaped network under multiplicity and asynchrony assumptions. About timing assumption, we consider the more general asynchronous model ASYNC. We take into account multiplicity assumption, which makes it much harder to formalize mobile robot algorithms. We focus on the gathering problem and analyze the algorithm proposed by D'Angelo *et al.* [21] as a case study. As the result of the model checking, counterexamples have been found. We refute by model checking that the algorithm enjoys desired properties. We detect the sources of some unforeseen design errors. We, furthermore, give our explanations on these errors. Two main results are: (1) a formal model for mobile robot algorithms on anonymous ring shaped network under multiplicity and asynchrony assumptions – our model is general enough and could be applied to other problems (in the ring); (2) a refutation by model checking that the algorithm enjoys desired properties — in detail, the algorithm contains design errors that prevent robots from gathering into one location. The additional contributions are a preliminary set of Maude modules that could be re-used for future verifications and the interpretations of the errors found.

The contribution of our work is the proof by example that formal methods must be used to verify distributed robot algorithms. Indeed, even algorithms described and proven using mathematical abstractions (may) still contain errors. While some of them are minor and could have been detected by a careful reader (simple typos), some errors would have been almost *impossible* to detect without model-checking. Said differently, we claim that informal and semi-formal mathematical proofs are not enough for this kind of algorithms.

The complexity of analyzing all situations may be too high for human brains. Another valuable contribution of our work is that we discover the challenges to specify and model check mobile ring robot algorithms. Because rings are not supported in Maude, we need to specify rings by adapting other defined structures, such as *sets* and *sequences*. It, thus, makes the specification task tedious as well as time-consuming, while the specifications obtained are complicated and lengthy.

An Environment and a Domain-Specific Language

An environment and a domain-specific language for specifying and model checking mobile robot algorithms on rings (or mobile ring robot algorithms) are proposed. First, we develop Maude Ring Specification Environment (Maude RSE), a ring specification environment that explicitly supports ring-shaped networks. Maude RSE is implemented in Maude. Then, we build our domain-specific language, Mobile Ring Robot Maude (MR^2-Maude), on top of Maude RSE. MR^2-Maude makes it possible to specify mobile ring robot algorithms in such a way that the specifications are as close as possible to their mathematical descriptions. One key underlying these tools is pattern matching between ring patterns and ring instances, called "ring pattern matching." Because rings are not commonly available data structures in any existing specification language, we encode ring patterns as sets of sequence patterns and simulate ring pattern matching by pattern matching between sets of sequence patterns and sequence instances, which is proven correct and transparent to both Maude RSE and MR^2-Maude users. MR^2-Maude predefines some LTL formulas as well as atomic propositions to model check that such algorithms enjoy desired properties. The advantages of Maude RSE and MR^2-Maude are demonstrated by case studies analyzing exploration and gathering mobile robot algorithms.

Maude RSE and MR^2-Maude themselves are the main achievements. Our research illustrates the power of rewriting logic in that Maude RSE can be implemented by extending Maude, more precisely Full Maude, and MR^2-Maude can be implemented by further extending Maude RSE. That is, we do not need to implement such formal tools from scratch but we can do so by extending Maude and/or new formal tools on top of Maude. Case studies are conducted with Maude RSE as well as MR^2-Maude, demonstrating that because Maude RSE supports ring structures, mobile ring robot algorithm specifications in Maude RSE are more concise and compact than those in Maude; the overhead incurred by handling rings is almost nothing and mobile ring robot algorithm specifications in MR^2-Maude are close to their mathematical descriptions. From a theoretical point of view, we prove that ring pattern matching can be correctly simulated by pattern matching between sets of sequence patterns and sequence instances. Therefore, Maude RSE as well as MR^2-Maude will benefit researchers in both the formal methods community and the distributed computing community.

1.3.3 Summary

Our research discovers the power of rewriting logic as well as its meta-programming facilities to formal specification and model checking of distributed systems. We have demon-

strated that meta-programs can be used as formal specifications of distributed algorithms. The main contributions of the thesis are: (1) a formal specification and model checking approach based on meta-programming to control algorithms, (2) a specification environment and a domain-specific language for specifying and model checking mobile ring robot algorithms.

1.4 Thesis Structure

We organize the structure of this thesis into seven chapters. We summarize each chapter as follow:

Chapter 1 is an introductory chapter. It first gives a brief introduction to control algorithms and mobile robot algorithms. It then summaries the problems of formal verification of distributed algorithms followed by our contributions, and thesis structure.

Chapter 2 provides preliminaries to this thesis. Many researchers have attempted to formally analyze and verify distributed systems (or algorithms). This chapter will present a careful investigation on the advancements of some related researches as well. This chapter then introduces rewriting logic and its applications, and explains about its important aspects as reflective logic.

Chapter 3 introduces Maude and its LTL model checker. It then describes how to specify and model check a distributed system in Maude. It also introduces Maude meta-progamming facilities and show how to implement a meta-program in Maude.

Chapter 4 proposes a new approach to specifying and model checking control algorithms. The approach is explained by applying it to specifying and model checking a termination detection algorithm. We give our idea on generation of all possible initial states that fulfill some constraints so as to make it more likely to detect subtle errors lurking in control algorithms. Two case studies are given at the end of the chapter.

Chapter 5 focuses on how to specify and model check mobile robot algorithms in Maude. Namely, we analyze a perpetual exploration algorithm and a gathering algorithm.

Chapter 6 introduces Maude RSE and describes the theory of ring-patten matching. It presents how to specify mobile ring robot algorithms in Maude RSE. Maude RSE allows us to concisely and naturally specify mobile robot algorithms on ring shape networks. It then introduces MR^2-Maude and evaluates Maude RSE and MR^2-Maude.

Chapter 7 concludes the thesis, summarizing the main contributions made. It also reveals future directions.

References and publications follow Chapter 7.

Chapter 2

Preliminaries

This chapter provides preliminaries to this thesis. It first introduces control algorithms and mobile robot algorithms. Many researchers have attempted to formally analyze and verify distributed systems (or algorithms). This chapter will present a careful investigation on the advancements of some related research as well. This chapter then introduces rewriting logic and its applications, and explains about its important aspects as reflective logic.

2.1 Distributed Algorithms

A distributed system consists of a collection of computation units abstractly named *process* that cooperate to achieve a common goal. A set of n processes could be denoted as $\Pi = \{p_1, \ldots, p_n\}$, where each p_i, $1 \leq i \leq n$, represents a distinct process. The processes communicate by sending and receiving *messages* through *channels*, which are *wired* or *wireless* network. In some case, channels are assumed to be reliable (no lost, modify, or duplicate messages) and *first in first out (FIFO)*, which means that the messages are received in the order sent.

2.1.1 Control Algorithms

This section introduces a class of distributed algorithms - Control algorithms to which our methods are going to apply.

In distributed systems, an underlying application executes its own program, which represents the logic of the application. An application execution involves processes and their related communication channels. In many cases, however, some extra additional tasks should be carried out in order to monitor the application execution or to perform various global functions, such as: detecting termination, detecting distributed deadlock, detecting global stable predicates, recording global states, and checkpointing. It is, therefore, necessary to use many non-trivial distributed algorithms, such as termination detection algorithms, deadlock detection algorithms, global stable predicate detection algorithms, snapshot recording algorithms, and checkpointing algorithms. The essential characteris-

tics of such algorithms is that an algorithm execution is superimposed on the underlying application execution. An underlying distributed system may be regarded and treated as data by control algorithms (Fig 2.1).

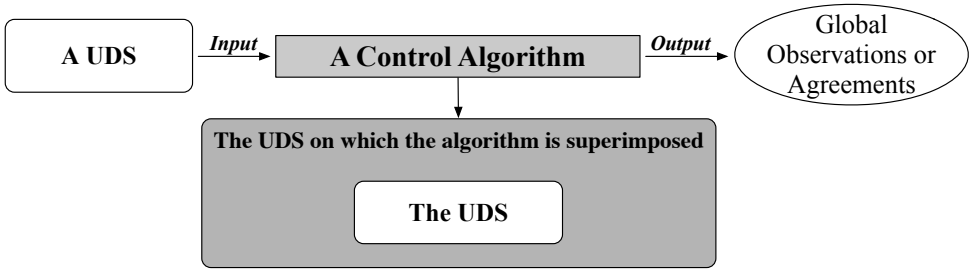


Figure 2.1: A control algorithm is superimposed on an underlying distributed system (UDS), which may be regarded and treated as data by the control algorithm.

The following part presents three different control algorithms. They are a termination detection algorithm, a snapshot recording algorithm and a checkpointing algorithm.

Four-Counter Algorithm

Let us consider the following simple, but non-trivial, control algorithm. The algorithm is called "Four-Counter Algorithm" for termination detection in an asynchronous atomic model.

Computation Model. A distributed system consists of a finite set of processes that are connected and communicate with the communication network, which is fully connected. Assuming the processes do not share a common global memory. Let $p_1, \ldots, p_n$ be n asynchronous processes. Each p_i has a local variable denoted $mode_i$ whose value is either active or passive. Initially, some processes, at least one, are in the active mode, while the others are in the passive mode. When a process p_i is active, it can execute local computations and send messages to the other processes. When a message reaches a process p_j, p_j instantaneously sets $mode_j$ to active. Messages are delivered reliably without any error in finite but arbitrary time that may or may not be in the sent order. The atomic model is a simplified model in that it takes only time to deliver a message to a destination process from a source process but the destination process does anything in it instantaneously, such as updating its internal state, on receipt of the message.

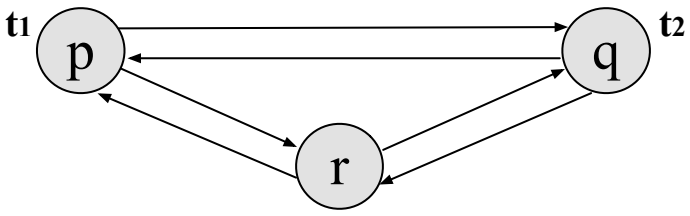


Figure 2.2: The initial state of the 3-processes, 6-channels and 2-tokens system

Fig. 2.2 describes an underlying distributed system consisting of three processes p, q, r and two tokens t_1 and t_2. The state of a process mainly depends on the set of tokens owned by the process. p_i has two actions: (1) consuming one of its tokens or sending the token to another process by putting the token into one of its outgoing channels if the process is active and (2) receiving a token from one of its incoming channels and becoming active in case it is in the passive mode. When a process does not hold any tokens, it may become passive and then remains passive unless it receives a message from another process. Initially, p is active and holds t_1, and q is passive, and holds t_2, while r is passive and does not have any tokens and all channels are empty.

Termination Detection Problem. It is very important to know when the distributed computation has terminated so that the result could be used or the next computation could be started. Intuitively, a distributed computation has terminated when all the processes are passive and all the channels are empty. The termination detection problem consists in designing an algorithm in order to detect the termination. Let $c[i, j]$ denote the channel from p_i to p_j and *emp* denote the empty channel. A distributed computation is said to be terminated if and only if:

For all i, j, $mode_i$ = passive and if there exists $c[i, j]$, then $c[i, j] = emp$.

This is called the "termination prediate."

Four-Counter Algorithm. The idea that underlies the algorithm is simple: it detects the termination by counting and comparing the number of messages that have been sent and received. The system is claimed to be terminated when the total number of messages sent is equivalent to the total number of messages received. Messages used in the underlying computation are called *computation messages*, and messages used for the purpose of termination detection (by Four-Counter Algorithm) are called *control messages*. The algorithm does not block any computation messages. Let us consider a simple version of the algorithm based on an inquiry-based principle. Each process p_i is required to count the number $sent_i$ of messages it has sent, and the number rec_i of messages it has received. Let us assume that there is an observer in charge of the termination detection. To start, the observer requests each process p_i to answer to let it know the pair $(sent_i, rec_i)$ by sending the pair to it. When the observer has all the pairs, it computes the total number S of the messages sent, and the total number R of the messages received. If $S = R$, it claims that the system has terminated. Otherwise, it starts its next inquiry.
At each process p_i:

- Receiving a computation message:

  $rec_i := rec_i + 1;$

- Sending a computation message:

  $sent_i := sent_i + 1;$

- Receiving a request from the observer:

  send the pair ($sent_i$, rec_i) to the observer.

At the observer:

repeat

$S := 0$; $R := 0$;
for each $i \in \{1, ..., n\}$ **do** send a request to p_i **end for**;
wait for an answer message $(sent_i, req_i)$ from each p_i;
$S := \sum_{1 \leq i \leq n} sent_i$; $R := \sum_{1 \leq i \leq n} rec_i$;
if $(R = S)$ **then** claim termination; **exit loop**
end if

// start the next inquiry.

endrepeat.

Chandy-Lamport Distributed Snapshot Algorithm (Chandy-Lamport Algorithm)

CLDSA [50] is the first and fundamental snapshot algorithm to record a consistent global state of a distributed system.

Underlying Distributed System. Channels are unbounded reliable queues (FIFO). The state of a channel is characterized by the sequence of in-trans messages sent along the channel and not yet received by the destination process. Fig. 2.3 shows a system that consists of three processes p, q, r and four channels, in which there are two channels from p to q.

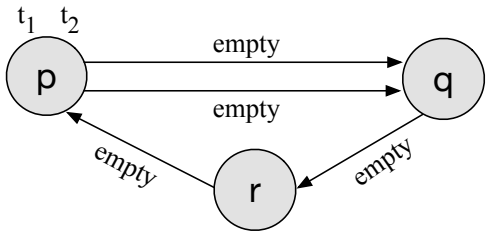


Figure 2.3: The initial state of a toke system with 3-processes & 4-channels

Chandy-Lamport Algorithm. Chandy and Lamport have proposed Chandy-Lamport algorithm by which processes can record their own states and the states of incoming channels such that the combination of all process states and all channel states forms a consistent global state. Chandy-Lamport algorithm guides each process when it should

record its own state and the state of each incoming channel by using a special message called *marker*. Each process can record its state at any time when it has not yet received any markers from other processes. These rules must be followed:

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

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

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

The algorithm will be terminated when each process has recorded its state and the states of all of its incoming channels. The global snapshot then is collected by combining those recorded process and channel states. An important aspect of the algorithm is that the algorithm runs concurrently with, but does not alter, the behavior of an underlying distributed system.

The Distributed Snapshot Reachability Property. Let s_1, s_* and s_2 be the state in which Chandy-Lamport algorithm is initiated (the start state), the snapshot taken, and the state in which Chandy-Lamport algorithm is terminated (the finish state), respectively. Although the snapshot s_* may not be identical to any of the global states that occur in the computation from s_1 to s_2, one desired property (called the distributed snapshot reachability property) Chandy-Lamport algorithm should satisfy is that s_* is reachable from s_1 and s_2 is reachable from s_*, whenever Chandy-Lamport algorithm is terminated. Note that s_1, s_2 and s_* are states of the underlying distributed system but not those of the underlying distributed system on which Chandy-Lamport algorithm is superimposed (UDS-CLDSA).

Checkpointing Algorithm

Checkpointing is an essential technique for fault tolerance distributed systems. It helps to reduce the amount of lost work by periodically saving the state of a process during the failure-prone execution. A failure system could restart its execution from the saved state upon to the failure. The saved state by a process is called local checkpoint and a global checkpoint is a set of local checkpoints, one from each process. However, an arbitrary set of local checkpoints may not form a consistent global checkpoint. A consistent global checkpoint is a global checkpoint such that no message that is sent by a process after taking its local checkpoint is recorded as a received message by another process in its

local checkpoint. There are two main approaches for checkpointing: coordinated checkpointing and uncoordinated checkpointing. In this case study, we specify and model check a coordinated checkpointing algorithm for mobile distributed systems with our approach.

Mobile Distributed Systems. A mobile computing system consists of the set of mobile hosts that can move and the set of mobile support stations that act as access points to connect mobile hosts to the rest of the network. Mobile support stations communicates with other mobile support stations by wired networks, but it communicates with mobile hosts by wireless networks. The location of a mobile host is represented by its current local mobile support station. The system as shown in Fig. 2.4 consists of three mobile support stations and seven mobile hosts. Because of the mobility of a mobile host, it may connect to different mobile support stations from time to time. A mobile host may voluntarily disconnect from the network and a disconnected mobile host is unreachable from the rest of the network. Wired networks provide reliable FIFO message delivers. Messages are delivered in the order sent and arbitrary, but limited time. A mobile host can pass messages in a reliable FIFO wireless channel to communicate with an mobile support station when it locates in the cover of the mobile support station (called *cell*).

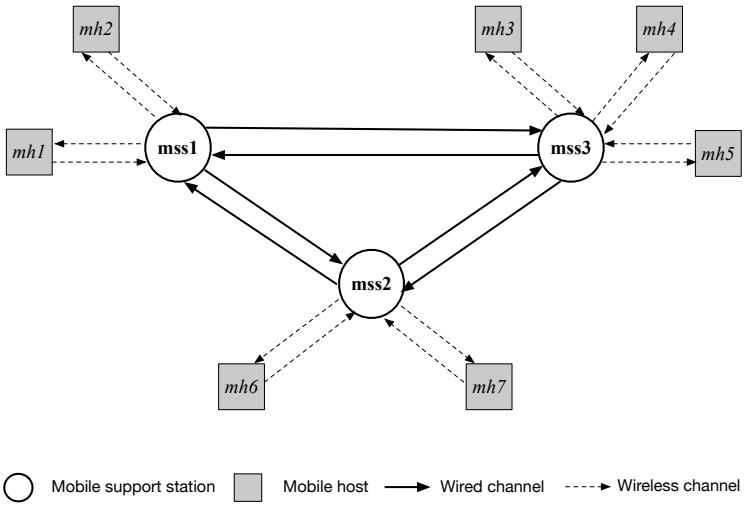


Figure 2.4: A mobile distributed system that consists of three mobile support stations and five mobile hosts.

Mutable Checkpointing Algorithm. Due to the mobility of mobile hosts and the limited capacity of wireless networks, it has raised some new issues in recording consistent global checkpoints, such as: (1) the communication delay and message complexity are increased because of the changes of the location of a mobile host; (2) the disk storage on a mobile host, which could be loss, theft, or physically damaged, cannot be considered as the stable storage and thus only minimum number of processes should be forced to take checkpoints, which need to be transferred to stable stores in mobile support stations through low bandwidth wireless channels; (3) a disconnected mobile host is not able to record its local checkpoints.

A checkpointing algorithm is executed in order to record a consistent global checkpoint. A checkpointing algorithm is superimposed on an underlying mobile distributed system (UMDS). There are two kinds of messages: computation underlying messages generated by the underlying distributed application and control messages generated by processes to advance checkpoints. Among several checkpointing techniques proposed, coordinated checkpointing is a commonly used technique, by which a process needs to synchronize their checkpointing activities in order to record a consistent global checkpoint. When a process records its local checkpoint, it asks (by sending control request messages) all relevant processes to take checkpoints. Cao and Singhal [13] have proposed a coordinated checkpointing algorithm, by which only the minimum number of processes have to take their checkpoints on the stable storage. The concept of "*mutable checkpoints*" is introduced. A mutable checkpoint is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into a tentative checkpoint at some points. It could be saved at anywhere, e.g, local storages in mobile hosts or stable storage. Intuitively, a process could start the algorithm at anytime when it is not in any checkpoint process. It takes its local checkpoint, and then sends checkpoint requests to its relative processes from which it has received messages. Requests made by a process can be inherited by other processes. A process may take a mutable checkpoint when it receives a computational message from a checkpointing process that has recorded its local state. The mutable checkpoint is turned to a tentative checkpoint if and only if the process is inherited a request message. Otherwise, it is discarded. In [13], the algorithms are described and explained in plaintext and also given in terms of pseudo-code.

2.1.2 Mobile Robot Algorithms

For the last two decades, the Distributed Computing community has been investigating what can be solved by a team of autonomous mobile robots. Following a different approach compared to the AI and Robotic communities, researchers started to propose formal models for these systems and design algorithms solving some predefined tasks. Theoretical research on distributed mobile robot focuses mainly on computability aspects; the goal is to determine whether a problem can be solved given some assumptions, such as synchrony, multiplicity detection and chirality. In this context, various models and algorithms have been proposed to solve various problems. There exist several different models, but they can be classified in two main classes: (i) discrete models in which movement are restricted on a graph [10, 11, 22] and (ii) continuous models in which entities move on a continuous space [34, 40, 64]. For both cases, a large variety of tasks have been considered such as gathering, pattern formation, scattering, flocking for continuous environments, and gathering, exploration, patrolling for discrete environments. For more details, we invite the interested reader to check the book [34] of Flocchini *et. al* that surveys many results (mostly on continuous models). In the discrete models, robots perform their activities in specific shape networks, such as rings and grids. They are observed at specific discrete locations. This thesis focuses on the ring discrete model. What and how problems can be solved by a group of autonomous mobile robots on ring shaped networks is a very interesting topic in the area, as shown by the large number of algorithms that have been

proposed. The authors in [10, 26, 27, 33, 44] have proposed algorithms to exploration on ring. Robot gathering on ring is solved in [11, 21, 23, 39, 41] and some other problems are solved in [22, 59]. There have been already some efforts to unify and formalize existing results [21, 22, 23, 33].

Computational Model

We considers the classic problems in the ring. The ring is *anonymous*, that is, there is neither node nor edge labeling. The robots are *identical*, *i.e.*, they are indistinguishable and all execute the same algorithm. Moreover, the robots are *oblivious* and *disoriented*, meaning that they have no memory of past actions, and they share no common orientation (no chirality). The robots cannot explicitly communicate, but have the ability to sense their environment and see the relative positions of the other robots, in their local coordinate system. When there is more than one robot, the node is called a *multiplicity* (or a *tower*). In some problems, such as robot gathering problem robots are assumed to have the global multiplicity detection. If robots have weak multiplicity detection, they can distinguish whether a node is empty, occupied by one robot, or more than one robot, but are not able to count the actual number of robots. With strong multiplicity detection, robots can count the exact number of robots in the multiplicity. In case they have no multiplicity detection, robots simply detect whether a node is empty or occupied by robots.

Robots follow a three-phase behavior: *Look*, *Compute*, and *Move*. During its Look phase a robot takes a snapshot of all robots' positions. The collected information (position of the other robots in the egocentric view) is used in the Compute phase during which the robot decides to move or stay idle. In the Move phase, the robot may move to one of the two adjacent nodes, as computed in the previous phase. The moves are assumed to be instantaneous which means that, during a Look phase, robots can be located on nodes only.

There are different types of synchronization for the robots: fully synchronous model (FSYNC), the semi-synchronous model (SSYNC) and the asynchronous model (ASYNC). We consider here the classic asynchronous ASYNC model [34]. The start and duration of each Look-Compute phases and the start of each Move phase of each robot are arbitrary and determined by an adversary. Note that it is possible for a robot to make a move based on a previously observed configuration which is not the current one anymore (*e.g.* if its Look phase occurred before the Move phase of another robot). A move that has been computed (during a Compute phase) but not yet executed (in the subsequent Move phase) is called a *pending move*.

The following part presents three mobile robot algorithms that solve three main problems on the ring shaped network: perpetual exploration, exploration with stop and gathering problems.

Perpetual Exploration

We consider the *exclusive perpetual exploration* of the *ring* and analyze one of the first algorithm proposed to solve this problem. More specifically we focus on the algorithm designed for *three robots*[1] by Blin *et al.* [10]. In the remaining of this part, we present successively the problem, and the algorithm under study. For each part, a more complete description can be found in the original paper [10].

Problem. The *perpetual* exploration problem requires each agent to visit each location (here, nodes of the ring) infinitely often. Moreover the *exclusive* nature of the exploration implies that two agents are not allowed to be on the same location at the same time; two robots cannot be on the same node and two robots cannot cross each-other on the same edge.

In order to verify the correctness of an algorithms, two properties have to be model-checked:

- The perpetual exploration property, and
- The mutual exclusion property.

where the former is a liveness property, while the latter is a safety property.

Algorithm. We recall here first some notations used to describe configurations and algorithms, and then present succinctly the 3-robot exploration algorithm.

Configuration encoding: In order to represent in a concise way any configuration of the system, we use the classical encoding as the sequence of occupied/free nodes of the ring (as in [10]). A configuration is an alternative (circular and non oriented) sequence of symbols R and F, indexed by integers: R_i stands for i consecutive nodes, each of them occupied by a robot, and F_j stands for j consecutive nodes free of any robot. For example the configuration of Figure 2.5 depicting 3 robots on a 10-nodes ring is encoded as (R_2, F_2, R_1, F_5) since there are 2 adjacent robots followed by 2 free nodes, followed by 1 robot, followed finally by 5 free nodes. Note that, due to the lack of orientation and origin, the very same configuration could also be encoded differently, for example with the sequence (R_1, F_2, R_2, F_5).

The configurations can be parametrized with integer variables when the size of the ring is unknown. For example, one can consider the configurations (R_2, F_2, R_1, F_z) in which there are two adjacent robots separated from the third robots by a gap of two empty nodes on one side, and z empty nodes on the side. Such notations allow to define generic algorithms for arbitrary size of ring.

Similar encoding will be used in the formal specification of the algorithm (see Section 5.1) in Maude. The only difference is that we use the number of edges between two

[1] It is natural to consider 3-robot algorithms, since, for non-trivial rings, any exploration algorithm requires at least three robots.

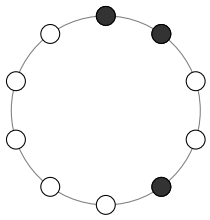


Figure 2.5: Configuration (R_2, F_2, R_1, F_5) with 3 robots on a 10-node ring.

robots, instead of counting the number of free nodes.

Move encoding: Each robot computes its next move based on the position of other robots. Therefore, designing an algorithm means giving the function that associates a move to any possible snapshot. A concise way of representing such algorithm is to write transition rules such as:

$$(R_2, F_2, R_1, F_z) \rightarrow (R_1, F_1, R_1, F_2, R_1, F_z)$$

Such a rule encodes the computed movement of each of the three robots when they took a snapshot corresponding to the configuration on the left. For example, in this case (see Figure 2.6), the isolated robot (corresponding to R_1) should not move, and among the two other robots (corresponding to R_2), only the furthest one (wrt. the isolated robot) should compute a move to go away. On Figure 2.6, the computed move is anti-clockwise.

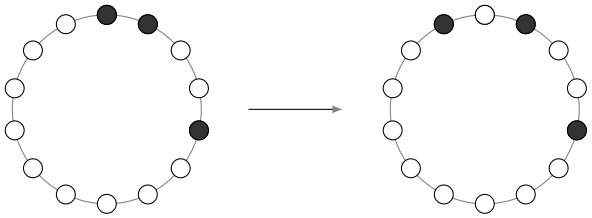


Figure 2.6: Rule RL1: $(R_2, F_2, R_1, F_z) \rightarrow (R_1, F_1, R_1, F_2, R_1, F_z)$.

Again, a similar encoding will be used in the formal specification of the algorithm (see Section 5.1). Each rule of the algorithm has a corresponding conditional rewriting rule in Maude.

Algorithm rules: The algorithm for three robots designed by Blin *et al.* [10] works in two phases. First there is a *Convergence Phase* which guarantees that starting from any initial configuration, the system reaches one of the three *legitimate*[2] configurations. Then during the *Legitimate Phase*, the system cycles between three configurations to explore perpetually the ring. The Legitimate Phase consist of the three rules RL1, RL2, and

[2]The terminology comes from the Self-Stabilization concept. One can understand such configurations as "good" configurations.

RL3 represented[3] on Figures 2.6, 2.7, and 2.8. One can "easily check" that applying successively these rules to the three robots indeed solves the perpetual exploration.

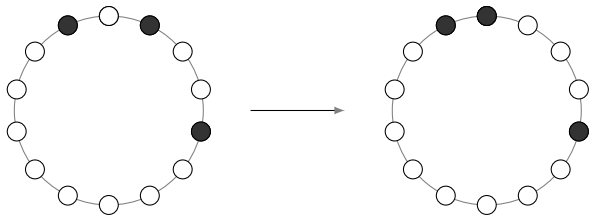


Figure 2.7: Rule RL2: $(R_1, F_1, R_1, F_2, R_1, F_z) \rightarrow (R_2, F_3, R_1, F_z)$.

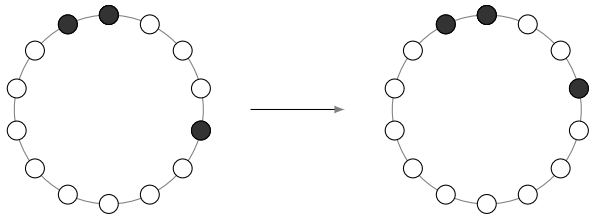


Figure 2.8: Rule RL3: $(R_2, F_3, R_1, F_z) \rightarrow (R_2, F_2, R_1, F_{z+1})$.

The set of rules for the Convergence Phase is omitted here, but can be found in [10].

Exploration with Stop

This part consider another problem of exploring, the exploration with stop an anonymous unoriented ring by a group of identical, oblivious, asynchronous mobile robots.

Problem. It is assumed that there may be more than one robot located at the same node. Each robot can distinguish whether a node is empty, occupied by one robot, or more than one robot. The problem of exploring with stop requires that within finite time and regardless of the initial placement of the robots, each node must be visited by at least one robot and the robots must be in a configuration in which they all remain idle.

Algorithm. We analyze the algorithm [33] by which, starting from any initial configuration of the k robots without towers, allows the robots to explore the entire ring and brings all robots to a configuration in which they all remain idle. The algorithm works following a sequence of three distinct phases: Set-Up, Tower-Creation, and Exploration. The purpose of the Set-Up phase is to transform an initial configuration into one from a predetermined set of configurations (called no-towers-final) with special properties. More precisely, in the Set-Up phase, the robots create a configuration where there is a single set of consecutive nodes occupied by robots, or two such sets of the same size (called blocks). The purpose of the Tower-Creation phase is to transform the no-towers-final configuration

[3]Pictures represent a ring of size 14, but the rules are defined for arbitrary size, as written in the corresponding captions.

created in the previous phase, into one from a predetermined set of configurations (called towers-completed) in which everything is prepared for exploration to begin. During the Exploration phase, the ring is actually being explored. The configuration reached upon exploration depends solely on the configuration at the beginning of this phase. The set of these special terminal configurations is uniquely identified, and once in a configuration of this type, no robots will make any further move.

Robot Gathering

We consider the classic gathering problem in the ring under the asynchronous scheduler (ASYNC). We analyze the most general algorithm that solves the problem for (almost) all initial configurations. In the remainder of this section, we succinctly present the problem, and the algorithm under study. For each part, a more complete description can be found in the original paper [21].

Gathering Problem. Robots in the system have the global weak multiplicity detection. The gathering problem requires each robot to terminate on the same node. The problem is solved if all robots are on the same location and there is no pending move.

Gathering Algorithm. The considered algorithm is said to be general in a sense that it solves the problem for all valid (*i.e.* without multiplicity) initial configurations outside of $NG \cup SP4$, where (1) NG is the set of non-gatherable configurations (such as periodic configurations), from which it is impossible to gather robots, and (2) $SP4$ is the "small" set of special configurations with 4 robots from which it is still unknown whether it is possible to gather robots (Bonnet *et al.* [11] gave a partial answer).

We analyze the gathering algorithm [21] for robot gathering. The algorithm executes these four phases sequentially:

1. Starting from an initial configuration without multiplicity, the algorithm executes a procedure MULTIPLICITY-CREATION that creates either one or two symmetric multiplicities.

2. A second phase named COLLECT consists in moving all but four robots in the previously created multiplicities.

3. A third phase called MULTIPLICITY-CONVERGENCE makes the two multiplicities to merge into a single one.

4. Finally the phase CONVERGENCE allows the remaining single robots to join the unique multiplicity, which concludes the gathering.

This is a short (partially incorrect) summary. In some *rare* cases, the sequence of four phases may be temporarily broken. (*e.g.* the system may go back to the COLLECT phase while executing the CONVERGENCE phase). But eventually all robots should gather at the same location.

The algorithm contains some specific subroutines for configurations with four or six robots. The precise algorithm is not given for four or six robots, but refers to other papers. At the moment, we decided not to include them in our analysis; as in the paper, they could be dealt separately.

Plaintext vs. *Pseudo-code:* the algorithms are described and explained in plaintext and also given in term of pseudo-code. We think that the pseudo-code version contains less ambiguities than the plaintext version. In (pseudo-)code, there is usually no place for interpretation; it is thus either correct or incorrect. Since our goal is to formally model-check, we believe that it makes more sense to base our analysis on the most formal available version. That is why we analyze the pseudo-code version of the algorithms.

This is certainly an arguable decision. Indeed, some of the errors, may or may not exist in the plaintext description of the algorithms. We can not conclude anything about the correctness of the plaintext algorithm since it is subject to interpretation.

2.2 Literature Review on Formal Verification of Distributed Systems

Distributed algorithms are both important and difficult. Because model checking is powerful and highly suitable, it is a favorite choice to formally verify these systems. Several specification languages and model checker, such as DVE, Spin, NuSMV, TLC, Maude, CafeOBJ and Elan support to specify and model check concurrent systems including distributed systems. Among them, Maude, CafeOBJ and Elan are based on rewriting logic. However, distributed algorithms is really complex, error-prone and hard to intentionally design. Model checking of distributed algorithms has not been approached in a structured way. Several research on formal specification and verification of such systems have been conducted [42][2][65][66][12] to address specific algorithms in a fixed computation environment. The authors in [65] deal with the problem of verification of asynchronous consensus algorithms. The authors propose a semi-automatic verification approach based on model checking technique. The authors' approach is similar to k-induction implemented in Symbolic Analysis Laboratory (SAL) [25]. The approach can deal with an arbitrary number of sessions made but needs to fix the number of processes. Up to 10 processes can be treated. The authors of [2] develop an extension of the linear temporal logic (LTL) model checker of Maude such that an extension of LTL called a fair linear temporal logic of rewriting (LTLR) is used as the logic for properties and properties expressed in LTLR can be model checked under fairness assumption. The most distinguished feature of their approach is to make it possible to model check a system to and/or from which entities are dynamically added and/or removed under fairness assumptions.

Grov *et al.* [36] & [35] use Maude to formalize and analyze Google's Megastore, which is a cloud data store that provides transactions and ensures serializability for certain classes of transactions. Those papers also focus on correctness analysis. In [36], they provide a precise formal model of Megastore given in [18]. The formal model gives more detail and precise description of Megastore than the original version. The paper shows how to

model, specify and model check Megastore in the rewriting-logic-based Real-Time Maude language and tool [58]. In [35], they extend Megastore to provide an extra consistency for transactions accessing data from multiple entity groups. The proposed extension, Megastore-CGC, is developed by using the formal specification language and analysis tool Real-Time Maude. The paper gives the Real-Time Maude specification of Megastore-CGC and show how Real-Time Maude can model check Megastore-CGC and estimate its performance.

Liu *et al.* [48, 49, 61, 62] give a long-term research efforts to the formalization of cloud-based transaction systems based on rewriting logic. The verification of these systems involves not only the consistency guarantees, but also the performance of the systems. In [61] & [62], they use Maude to model and analyzes Read Atomic Multi-Partition (RAMP) transition systems. In [61], the authors formalize RAMP transitions in rewriting logic and some proposed extensions, and used Maude model checking to analyze their correctness properties. In [62], the paper focuses on analyzing the performance of RAMP and its variations using statistical model checking with PVeStA. In addition, the paper proposes a new RAMP design alternative: RAMP- Faster that is expected to outperforms all others. Recently, in [48], the authors provide a formal executable specification of Walter in Maude. It then model checks the snapshot isolation and the parallel snapshot isolation properties for the algorithm. The paper also provides a parametric method to generate all initial states for given parameters and then performing model checking analysis to verify the snapshot isolation and the parallel snapshot isolation properties for all initial states for various parameter choices. Although the paper proposes a technique similar to us, but their technique does not take into account network topologies, while our approach takes into account different network topologies. The authors in [49] design a new distributed transition protocol, ROLA, that supports applications that only require read atomicity (RA) and prevention of lost update (PLU). ROLA is given with its formal verification. Namely, the protocol is formally specified in Maude and then model checked that it satisfies RA and PLU. The author analyze performance properties by using the statistical model checking. The performance of the protocol is compared with Walter protocol.

The authors in [46] proposes cache-based model checking, which concentrates on verifying a single process in a distributed system, which is a software model checking that model check directly an implementation, not a model specification. The tool is built of top of a Java model checker as its extension. It frees the limitation of stand-alone process by verify one process at a time and running other process in the another execution environment. The tool offers a scalable method to verify a system because the state space of one process is visited, but not the composite set of state space of all processes. Closely to our method, Meseguer *et al.* [53] proposes formal patterns for distributed systems. He tries to give a genetic and reusable pattern to design and specify distributed algorithms. Several patterns as theory transformations are given. Reflective logic and the meta level is used to describe one of these formal patterns.

Several studies are motivated by verification of snapshot algorithms. Among them, Gerard *et al.* [37] and Bruno *et al.* [12] consider Chandy-Lamport algorithm. The algorithm and its distributed snapshot reachability property were introduced early in the

original paper [51]. In [37], Chandy-Lamport algorithm is modelled in PROMELA, and then the model is simplified to be verifiable. However, only the underlying distributed system superimposed by Chandy-Lamport algorithm is modelled. The authors in [12] focus on developing snapshot algorithm with a formal proof that guarantees its correctness. Some existing snapshot algorithms, such as Chandy-Lamport algorithm and Lai-Yang are re-developed by using the Event-B framework and refinement. Starting with a model providing an abstract view of a system and its behavior, the model then is enriched more concretely by many refinement steps to derive the algorithms. Not only two, but many state machines are modelled. Their experiments are conducted on fixed networks.

Although the current model checking techniques allow us to detect counterexamples, they cannot guarantee that distributed algorithms surely enjoy desired properties because all possible cases may not be covered. Some abstraction techniques have been introduced to overcome this challenge. Many such techniques [15][8] have been proposed. In the Maude community, equational abstraction [54][52] is one such technique. Abstract logical model checking has been proposed to deal with infinity in the Maude community as well. Assuming that a distributed system has been specified by means of a rewrite theory $\mathcal{R} = (\triangle, E, R)$, with $(\triangle, E)$ an equational theory specifying the set of states as an algebraic data type, and R specifying the system transitions as a set of rewrite rules, the main technique of the equational abstraction consists on adding more equations, E', to get a quotient system specified by the rewrite theory $\mathcal{R}/E' = (\triangle, E \cup E', R)$. Such a system is called an equational abstraction of $\mathcal{R}$.

Meta-programming technique is powerful in the processing of other programs, which are treated as data. In [56], the authors developed a meta-program for debugging normal logic programs. In [20], a meta-interface mechanism (called a Quality Connector) for reducing the life-cycle costs of distributed real-time and embedded systems is proposed. Meta-programming technique is investigated to use in order to provide distributed real-time and embedded applications that can specify the qualities of service they require from their middleware. Although many researches have been conducted on application of meta-programming technique, as far as we know, the research on applying meta-programming to specification of distributed algorithms have not been conducted so far.

Due to the mobility aspect, mobile robot algorithms are often complex, arguably even more complex than classic distributed systems. This inherent difficulty explains probably the limited number of attempts in obtaining formal verifications. Formal, automatic techniques could help to increase the confidence of the existing algorithms/proofs as the works in [1, 5, 9, 19]. Courtieu *et al.* [19] and Balabonski *et al.* [5] formally proved the correctness of two gathering algorithms using the Coq proof assistant. In both cases, robots move on the continuous 2-dimensional plane. The difference lies in the timing model; the first paper considers the semi-synchronous (SSYNC) model while the second studies the fully-synchronous (FSYNC) model (and remove some other assumptions). The asynchronous model (ASYNC) is not considered in these papers since the gathering problem is generally not solvable in ASYNC in continuous space (except for trivial cases). Closely related to our current approach, Berard *et al.* [9] proposed a formal model to describe mobile robot algorithms in a discrete model with a finite set of possible robot

positions, under synchrony and asynchrony assumptions. The model then is transferred into the DVE language using DiVinE and ITS tools. The authors formally specifies the algorithm and then encode in Linear Temporal Logic (LTL) the properties that should be satisfied. The paper analyzes the perpetual exploration algorithm described in [10]. The algorithm, while being quite simple (only 3 robots on the ring and 8 rules), was refuted. They then give a correct version of the algorithm together with the formal proofs. These studies [5, 9, 19] basically consider either FSYNC or ASYNC without multiplicity assumption. The lack of multiplicity, of course, simplifies the model.

Recently, Aminof *et al.* [1] provided a general framework for modeling and verifying the systems in which multiple mobile robots evolves in a partially-known environment, e.g a robot in a ring may know about positions of other robots, but not the size of the ring. The authors apply formal methods to the parameterised verification problem in which it is the environment that is parameterised. Another parameterised verification is given in [63]. The paper presents a study to the verification problem of mobile robot algorithms in the parameterized case. Namely, the authors focus on formally verifying algorithms parameterized by the number of ring positions, assuming a fixed number of robots. The approach then is applied by using an SMT-solver to verify safety properties to some proposed algorithms.

2.3 Reflection and Meta-programming

This section introduces Rewriting logic and explains its important aspects as reflective logic, as well as introduces Metaprogramming, which is the main technique used in our method.

2.3.1 Rewriting Logic as a Reflective Logic

Rewriting logic is a natural model of computations for concurrency, parallelism and communication systems. Several specification languages based on rewriting logic, such as Maude, CafeOBJ and ELAN have been designed and implemented. Moreover, rewriting logic is a reflective logic that can be faithfully interpreted in its self. Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself. The essential of a reflective logic is that there is a universal theory $\mathcal{U}$ in which we can represent any other theories of the logic. In term of rewriting logic, we can represent any finitely presented rewrite theory $\mathcal{R}$ including $\mathcal{U}$ itself as a term $\overline{\mathcal{R}}$, any term t, t' in $\mathcal{R}$ as terms $\overline{t}$, $\overline{t'}$, and any pair ($\mathcal{R}$, t) as a term $\langle\overline{\mathcal{R}}, \overline{t}\rangle$. Since $\mathcal{U}$ can be represented in itself, we can achieve the following equivalence reflective tower with an arbitrary number of levels of reflections:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \to \langle\overline{\mathcal{R}}, \overline{t'}\rangle \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t}\rangle}\rangle \to \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t'}\rangle}\rangle \ldots$$

2.3.2 Meta-programming

Metaprogramming is a programming technique that is equipped with the ability to treat programs as data. A meta-program takes programs as inputs and perform some useful computations, such as analysing the program and transforming the program into another (Fig. 2.9). It requires to move from the object level to the meta-level, where programs are treated as data. Metaprogramming is necessary and also powerful with many important applications. Translators and debuggers, for instance, are such applications.

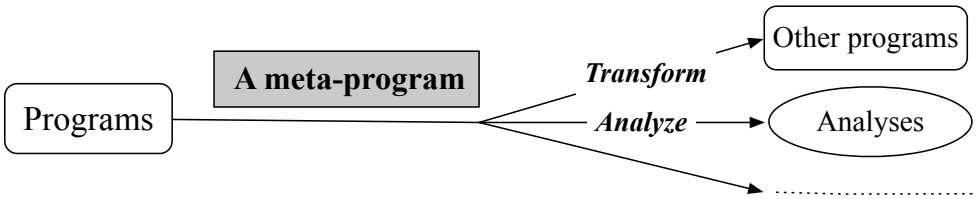


Figure 2.9: A meta-program takes programs as inputs and perform some useful computations, such as transforming and analyzing.

The language in which a meta-program is written is called the metalanguage. Because rewriting logic is a reflective logic, this makes specification languages based on rewriting logic, such as Maude able to provide meta-programming facilities. Metaprogramming is the main technique used in our method. Our method is based on rewriting logic meta-programming facilities.

Chapter 3

Maude and Meta-programming in Maude

This chapter first gives a brief introduction of Maude and shows how to specify a distributed system in Maude. It then explains how to use meta-progamming facilities in Maude.

3.1 Maude and Its LTL Model Checker

Maude [17] is a rewriting logic-based programming and specification language and equipped with a powerful system (or environment). Rewriting logic makes it possible to naturally specify dynamic systems, and the Maude system has a linear temporal logic (LTL) model checker. Therefore, Maude allows to model check if dynamic systems specified in Maude enjoy properties expressed in LTL.

The basic units of specifications and programs are modules. A module contains syntax declarations, providing a suitable language to describe a system. A module consists of sort, sub-sort, operator, variable, equation and membership declarations, as well as rewrite rule declarations. A sort denotes a set, corresponding to a type in conventional programming languages. For example, the sort `Nat` denotes the set of natural numbers. There is the relation among sorts that is a same as the subset relation. A sort is a subsort of another sort iff the set denoted by the former is a subset of the one by the latter, and the latter is called a supersort of the former. Let `Zero` and `NzNat` be the sorts denoting $\{0\}$ and the set of non-zero natural numbers, both of which are subsets of the set of natural numbers. `Zero` and `NzNat` are subsorts of `Nat` and `Nat` is a supersort of `Zero` and `NzNat`. An operator is declared as follows: $f : S_1 \dots S_n \to S$, where $S_1, \dots, S_n, S$ are sorts and $n \geq 0$. Note that `->` may be used instead of $\to$. $S_1 \dots S_n$, S and $S_1 \dots S_n \to S$ are called the arity, the sort and the rank of f. When $n = 0$, f is called a constant. f takes a sequence of n things of $S_1, \dots, S_n$ and makes something of S, where "things" and "something" are what are called terms and will be described soon. Operators denote functions or data constructors. Each variable has its own sort. Terms of a sort S are inductively defined as follows: (1) each variable whose sort is S is a term of S and (2) for each operator f whose

rank is $S_1 \ldots S_n \to S$ and n terms $t_1, \ldots, t_n$ of $S_1 \ldots S_n$, $f(t_1, \ldots, t_n)$ is a term of S. Note that when $n = 0$, f as it is is a term of S. An operator may contain underscores, such as `_+_ : Nat Nat -> Nat`. If that is the case, a different notation than $f(t_1, \ldots, t_n)$ is used. For example, if `X` is a variable of `Nat`, then `X + X` is a term of `Nat`. Use of operators containing underscores allows us to define context-free grammars. An equation declares that two terms are equal. A rewrite rule declares that a term changes to another one. Equations can be used to define functions, while rewrite rules can be used to define state transitions.

A simple example is used to briefly describe Maude and its LTL model checker. The simple system is a ring-shaped network consisting of four nodes whose identifications are 0, 1, 2, and 3 clockwise (see Figure 3.1). There is one mobile robot in the system, located at one node. Initially, the robot is located at node 0. We take two versions of the system into account: System 1 and System 2. System 1 has four transitions: if the robot is located at node N, then one of the four transition moves the robot at node $(N+1)$ mod 4. System 2 has four more transitions as well: if the robot is located at node N, then one of the four transitions moves the robot at node $(N-1)$ mod 4. We take two properties into account. One property is that the robot never visits node 4, and the other property is that the robot visits node 3 infinitely many times. The two properties are denoted `nv4` and `v3im`, respectively. `nv4` is a safety property, while `v3im` is a liveness property.

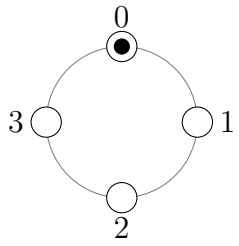


Figure 3.1: The initial configuration of the system.

The state in which the robot is located at N is expressed as term {`N`}. First we declared the following operators and equations:

```
sort NatMod4 .
ops 0 1 2 3 4 : -> NatMod4 [ctor] .
op inc : NatMod4 -> NatMod4 .
op dec : NatMod4 -> NatMod4 .
eq inc(0) = 1 . eq inc(1) = 2 . eq inc(2) = 3 . eq inc(3) = 0 .
eq dec(0) = 3 . eq dec(1) = 0 . eq dec(2) = 1 . eq dec(3) = 2 .
```

where `NatMod4` is a sort, `0`, `1`, `2`, `3` and `4` are constants of `NatMod4`, and `inc` and `dec` are operators that are defined in the following equations. The constants `0`, `1`, `2` and `3` of `NatMod4` are used to denote the node identifications. The constant `4` of `NatMod4` is used for expressing the property `nv4`. `ctor` stands for constructor, meaning that the operators

concerned are used to construct data. The operators `inc` and `dec` are the ordinary ones for natural numbers modulo 4.

The following sort and operator are declared:

```
sort Config .
op {_} : NatMod4 -> Config [ctor] .
```

As written, {N} denotes the state in which the robot is located at node N.

The four transitions in System 1 can be described as the following rewriting rule:

```
rl [rr] : {X} => {inc(X)} .
```

where `rr` is the label of the rule. Note that the rule has four instances for $N = 0$, 1, 2 and 3. The other four transitions in System 2 can be described as the following rewriting rule:

```
rl [lr] : {X} => {dec(X)} .
```

The system 2 is declared in the following module `SYSTEM`:

```
mod SYSTEM is
  sort NatMod4 .
  sort Config .

  ops 0 1 2 3 4 : -> NatMod4 [ctor] .
  op inc : NatMod4 -> NatMod4 .
  op dec : NatMod4 -> NatMod4 .
  op {_} : NatMod4 -> Config [ctor] .

  eq inc(0) = 1 . eq inc(1) = 2 . eq inc(2) = 3 . eq inc(3) = 0 .
  eq dec(0) = 3 . eq dec(1) = 0 . eq dec(2) = 1 . eq dec(3) = 2 .

  rl [rr] : {X} => {inc(X)} .
  rl [lr] : {X} => {dec(X)} .
endm
```

For System 1, the second rule is commented out or deleted.

To express the two properties in LTL, we need to prepare one proposition `(at N)` checking if the robot is located at node N in a given state. The proposition is declared in the following module `SYS-PROP`:

```
mod SYS-PROP is
  pr SYSTEM .

  inc SATISFACTION .
```

```
  subsort Config < State .
  op at_ : NatMod4 -> Prop .

  var X : NatMod4 .
  var C : Config .
  var P : Prop .

  eq {X} |= (at X) = true .
  eq C |= P = false [owise] .
endm
```

SATISFACTION is one module provided in the file model-checker.maude available in the Maude distribution. In the module, model satisfaction relation _|=_ and some more are declared. SATISFACTION is imported in the including mode. The first equation in the module says that (at X) holds in the state {X}. owise stands for otherwise. The second equation in the module says that (at X) does not hold in any other states.

The two properties are defined in the following module SYS-FORMULA:

```
mod SYS-FORMULA is
  inc SYS-PROP .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .

  ops nv4 v3im : -> Formula .

  eq nv4 = [] ~(at 4) .
  eq v3im = [] <> (at 3) .
endm
```

The operator []_ is the always temporal operator, and the <>_ is the eventually temporal operator. [] ~(at 4) says that the robot never visits node 4, and [] <> (at 3) says that the robot visits node 3 infinitely many times. The two properties are model checked as follows:

```
red in SYS-FORMULA : modelCheck({0},nv4) .
red in SYS-FORMULA : modelCheck({0},v3im) .
```

The Maude LTL model checker concludes that System 1 satisfies both properties, while System 2 satisfies nv4 but not v3im. The counterexample shown is as follows:

```
result ModelCheckResult: counterexample({{0},'rr}, {{1},'rr} {{2},'lr})
```

saying that once the robot moves to node 1 from node 0, the robot always moves clockwise when it is located at node 1 and always moves counterclockwise when it is located at node 2, never visiting node 3. If we assume Strong Fairness of transitions to model check v3im, then even System 2 enjoys v3im, an extended version of the Maude LTL model checker that facilitates model checking liveness properties under fairness has been developed [2].

3.2 Specifying an Underlying Distributed Systems as a Module

A distributed system is formalized as a state machine, which is specified in Maude as a module. A state machine consists of a set of states, some of which are initial states, and a binary relation over states. The definition is as follows:

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

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

Elements (denoted $s \rightarrow s'$*) of the binary relation are called state transitions, where* $s \rightarrow s'$ *says that* s *can change to* s'*.*

The reachable states of a state machine are inductively defined as follows: (1) each initial state is reachable and (2) for each reachable state s and each state transition $s \rightarrow s'$, s' is reachable. A state predicate p is an invariant property of the state machine iff $p(s)$ holds for all reachable states s of the state machine.

A sort and the set denoted by the sort are interchangeably used in this thesis. We often use associative-commutative collections as key data structures. Such collections are called soups. Let *Soup* be the sort for soups and *Elt* be the sort for their elements. *Elt* is declared as a subsort of *Soup* meaning that an element is treated as a singleton soup that only consists of the element. We use associative-commutative binary operators as data constructors of soups. The juxtaposition operator `_ _` is used as such a binary operator in this thesis. Let e_1, e_2, e_3 be terms of *Elt*. The term $e_1\ e_2\ e_3$ represents the soup that consists of the three elements. Since the juxtaposition operator is associative and commutative, there are other terms, such as $e_3\ e_2\ e_1$ and $e_2\ e_1\ e_3$, that exactly represent the same soup. Each element $e_i (i = 1, 2, 3)$ is treated as the singleton soup that only consists of e_i. We also use a constant of *Soup* as the empty soup that is treated as an identity of the associative-commutative binary operator. For *Soup*, *empSoup* is used as such a constant in this thesis and then $e_1\ e_2\ e_3$ *empSoup*, e_3 *empSoup* e_2 *empSoup* e_3, etc. represent exactly the same soup as denoted by $e_1\ e_2\ e_3$. The sort and constructor for such soups are declared as follows:

```
subsort Elt < Soup .

op empSoup : -> Soup [ctor] .
op _ _ : Soup Soup -> Soup [ctor assoc comm  id: empSoup].
```

The following part shows how to formalize and specify a UDS. Let us consider the system shown in Fig. 2.2 as an example.

State Expressions

Let `Pid`, `Mode` & `Token` be the sorts for process identifiers, process modes & tokens, `p`, `q` & `r` be the constants of `Pid`, `active` & `passive` be the constants of `Mode` and `t1` & `t2` be the constants of `Token`. Let `TknSet` be the sort for token soups. Local states of processes and channels are expressed as what are called observable components that are name-value pairs. Let `OCom` be the sort for observable components. The local state of each process whose identifier is p is expressed as the observable component `<`p`:` md`,`ts`>`, where p is the name, "md`,`ts" is the value, md is a process mode and ts is a token soup. Let `Msg` be the sort for messages and the supersort of `Token`. Thus, tokens are messages. Let `Chan` be the sort for message soups. The local state of each channel from a process p to a process q is expressed as the observable component `[`p`,`q`:` ms`]`, where "p`,`q" is the name and ms is the value, which is a message soup. Global states of the system are expressesd as soups of observable components for which `Config` is used as the sort.

```
sorts Pid Msg Token Mode
MsgSet OCom Config .
subsort Token < TknSet .
subsort Token < Msg .
subsort Msg < Chan .
subsort OCom < Config .

ops active passive : -> Mode .
ops t1 t2 : -> Token .
ops p q r : -> Pid .

op empTknSet : -> TknSet [ctor] .
op _ _ : TknSet TknSet -> TknSet [ctor assoc  comm id: empTknSet] .

op empChan : -> Chan [ctor] .
op _ _ :   Chan Chan -> Chan [ctor assoc comm id: empChan] .

op <_:_,_> : Pid Mode TknSet -> OCom [ctor] .
op [_,_:_] : Pid Pid Chan -> OCom [ctor] .

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

Initial States

In this system, there is only one initial state in which the process p is active and holds the token t_1, and the process q is passive, and holds the token t_2, while r is passive and does not have any tokens and all channels are empty. The initial state is expressed as a term of `Config`.

```
op init : -> Config .
eq init = (<p: active, t1>)(<q: passive, t2>)(<r: passive, empTknSet>)
([p, q: empChan])([p, r: empChan])([q, p: empChan])
([q, r: empChan])([r, p: empChan])([r, q: empChan])
```

where `init` equals the term expressing the initial state.

State Transitions

An action performed by a process is specified as a rewrite rule. One rewrite rule is as follows.

```
rl [sndToken] :
(<P: active, T TS> [P, Q: CS] BC) => (<P: active, TS> [P, Q: T CS] BC) .
```

where `P`, `Q`, `T`, `TS`, `CS` and `BC` are variables of `Pid`, `Pid`, `Token`, `TknSet`, `Chan` and `Config`, respectively. `sndToken` is the label given to the rewrite rule. The rule `sndToken` says that if `P` owns `T` and has an outgoing channel to `Q`, then `P` can put `T` into the channel. By substituting `P`, `Q`, `T`, `TS`, `CS` and `BC` with `p`, `q`, `t1`, `empTknSet`, `empChan` and

```
(<q: passive, t2>)(<r: passive, empTknSet>)
([q, p: empChan])([p, r: empChan])([q, r: empChan])
([r, p: empChan])([r, q: empChan])
```

respectively, the left-hand side of the rule `sndToken` is the same as the term `init` and then can be applied to the term. If so, the term changes to the following:

```
(<p: active, empTknSet>)(<q: passive, t2>)
(<r: passive, empTknSet>)
([p, q: t1])([q, p: empChan])([p, r: empChan])
([q, r: empChan])([r, p: empChan])([r, q: empChan])
```

In this way, state transitions are specified as rerwrite rules. The other actions can be specified likewise.

Let `SYS` be the module in which the system is specified. We will interchangeably use "state transitions" and "rewrite rules" (or simply "rules") in the rest of the thesis.

3.3 Meta-program in Maude

In Maude, metaprogramming has a logical, reflective semantics [17] because Maude is designed and implemented based on the fact that rewriting logic is a reflective logic. Maude strongly supports metaprogramming since all essential functionalities have been implemented in the functional module `META-LEVEL`. This module includes the modules `META-MODULE` and `META-TERM`.

Terms are only data structures available in Maude and programs/specifications in Maude are composed of modules. Thus, it is necessary and sufficient to be able to build modules as terms in metaprograms in Maude. To build a module as a term, we need to represent its components, such as operator and rewrite rule declarations, as terms. Terms representing modules, operator declarations, rewrite rule declarations, etc. are called their metarepresentations.

In the module `META-TERM`, sorts are metarepresented as data in the sort `Sort`, which is a subsort of the sort `Qid` of quoted identifiers. For example, `'Bool` and `'Nat` are terms of this sort. Maude terms are metarepresented as elements of the sort `Term` for terms. The sort `Term` is a supersort of the sorts `Constant` for constants and `Variable` for variables, which are subsorts of the sort `Qid`. Constants are quoted identifiers that contain the constant's name and its type separated by a '.', e.g, `'true.Bool`. Similarly, variables contain the variables' names and their types separated by a ':', e.g, `'X:Bool`. Arguments of an operator are expressed as a list of terms for which the sort `TermList` that is a supersort of the sort `Term` is prepared. A list of terms is built with the constructor `_,_`. For example, the term `(< P : active, t1 >` in the module `SYSTEM` is meta-represented in the following term of the sort `Term`:

```
'<_':_',_>['P:Pid,'active.Mode,'t1.Token]
```

where `'<_':_',_>` is the metarepresentations of the operator `(<_:_, _>)` and `'P:Pid, 'active.Mode,'t1.Token` is a list of terms and its sort is `TermList`. The term list is constructed by one variable `'P:Pid` and two constants `'active.Mode` and `'t1.Token`.

The module `META-MODULE` imports the module `META-TERM`. Maude modules are metarepresented as a term of the sort `Module` of modules. A functional module and system module are expressed as the following operators:

```
op fmod_is_sorts_._ _ _ _endfm : Header ImportList SortSet SubsortDeclSet
OpDeclSet MembAxSet EquationSet -> FModule [ctor] .

op mod_is_sorts_._ _ _ _ _endm : Header ImportList SortSet SubsortDeclSet
OpDeclSet MembAxSet EquationSet RuleSet -> SModule [ctor].
```

where `Header`, `ImportList`, `SortSet`, `SubsortDeclSet`, `OpDeclSet`, `MembAxSet`, `EquationSet` and `RuleSet` are the sorts for the metarepresentations of the name of the module, imported submodules, sort declarations, subsort declarations, operator declarations, membership declarations (that are not used in the thesis), equation declarations, and rule declarations, respectively. The sorts `FModule` and `SModule` are subsorts of the sort `Module`. They denote functional modules and system modules, respectively.

Maude is equipped with rich metaprogramming facilities to make it possible to do so and some more. Module `META-LEVEL` provides many useful and efficient built-in functions for moving among reflective tower levels, such as `upModule`, `upSorts`, `upTerm` and `downTerm`. We can use the function `upTerm` to obtain the metarepresentation of a term and use the function `downTerm` to recover the term. The functions `upModule` takes as arguments

the metarepresentation of the name of the module M and a Boolean b and returns the metarepresentation of the module M. If b is *true*, the metarepresentations of the module imported by M are also generated. Given a name *name*, such as a module name, a sort name and an operator name, its metarepresentation is the quoted ID `'`*name*, which is a term of the sort `Qid`. We can get the metarepresentation of the module `SYS` a by using the function `upModule` as follows:

```
upModule('SYS, false) .
```

We can write a meta-program in Maude by importing the functional module `META-LEVEL`. A meta-program can be implemented as a function that has modules as its arguments. Given a module whose name is M, let us consider to change the name to M`-FCA` and add a new sort `Obs-state` and a constant `terminate` of the sort to the module. To this end, we will define a meta-program as the following function `modify` in the module `MODIFY`.

```
mod MODIFY is
  pr META-LEVEL .
  op modify : Module -> Module .
  op reName : Module Qid -> Module .
  op addSort : Module SortSet -> Module .
  op addOpe :  Module OpDeclSet -> Module .

  var M : Module .
  eq modify(M) = addOpe(addSort(reName(M, 'FCA),
  'Obs-mode), op 'terminate : nil -> 'Obs-mode [none].).
  ...
endm
```

where `Module` is the sort for modules. `M` is a variable of `Module`. For a given module `M`, `reName`, `addSort` & `addOpe` are operators that changes the name to `M-FCA`, adds the sort `Obs-mode` to the module and add the operator declaration `op terminate : -> Obs-mode .` to the module. The term `op 'terminate : nil -> 'Obs-mode [none] .` is the metarepresentation of the operator declaration. "..." are some other variable and equation declarations.
Computing (or reducing) the term `modify(upModule( 'SYS, false))`, we obtain the module that is made by adding the sort `Obs-mode` and the constant `terminate` of `Obs-mode` to `SYS` and whose name is changed to `SYS-FCA`.

Although the above example is very simple, it introduces our idea to specify a CA as a meta-program in Maude.

Chapter 4

Specifying and Model Checking Control Algorithms at the Meta Level

This chapter proposes a new approach to specifying and model checking control algorithms. Meta-programming is the main technique used in our approach. A control algorithm is specified as a meta-program. The approach is explained by applying it to specifying and model checking a termination detection algorithm. We then give our idea on generation of all possible initial states that fulfill some constraints so as to make it more likely to detect subtle errors lurking in control algorithms. Two case studies are given at the end of the chapter.

4.1 Specifying Control Algorithms at the Meta Level

4.1.1 Meta-programs as Formal Specifications

Because a control algorithm runs concurrently with the underlying distributed system, the underlying distributed systems on which control algorithms are superimposed (UDS-CA) is considered as two layers: the underlying distributed systems as a core layer and the working of the algorithm as the cover layer. The cover layer monitors the behavior of the core. The system is the combination of the two layers. Given the formalization of an underlying distributed system and the working of the algorithm, it is possible to obtain the formalization of the UDS-CA from them.

The fundamental of our method is that an algorithm is specified as a parameterized specification, where parameters are the specifications of underlying distributed systems. Given the specification of an underlying distributed system, the specifications of the UDS-CA are generated by the parameterized specification. The implementation of the idea is, however, problematic at the object level because a specification of an underlying distributed system that is considered as a program at the object level need to be treated as input data to the parameterized specification. We solve the problem by moving from

the object level to the meta level and therefore meta-programming is used. Indeed, an algorithm is specified as a meta-program that takes a specification of an underlying distributed system as an input and generates the specification of the UDS-CA as shown in Fig 4.1.

Figure 4.1: A control algorithm is specified as a meta-program

Although the idea is generic, we use Maude and meta-programming in Maude to implement the idea. An underlying distributed system is specified as a module. The meta-programming is implemented based on the working of the algorithm such that it takes the module as its meta-representation, computes and returns the module that is the specification of the UDS-CA. Let us call it the meta-program T. As explained in Section 3.2, the specification of a system includes three parts:

1. The syntax part defining all notations to describe the system, in which sorts and operators are declared;
2. The initial part defining initial states of the system; and
3. The transition part defining the behavior of the system.

It is, therefore, convenient and possible to divide T into three sub-programs T_{State}, T_{Init} and T_{Trans} to deal with each above part. Later, `T`, `T-State`, `T-Init` and `T-Trans` may be used instead.

T_{State}

We may need to change some of the notations for a control algorithm and/or add some more notations so as to describe the UDS-CA. For example, the UDS-CA may need to use control messages that are different from computation messages used by an underlying distributed system and/or use extra components to save some information observed by the control algorithm. T_{State} takes a syntax part, modifies it and returns a new set of syntax declarations in which all notations to describe the UDS-CA are included.

```
op T-State : Module -> Module .
eq T-State(M) = setSyn(M, modifySyn(getSyn(M))) .
```

where `setSyn` gets the set of syntax declarations of a module `M`, `modifySyn` modifies this set and `setSyn` replaces the current set of syntax declarations of `M` by the modified one.

T_{Init}

An initial global state of the UDS-CA needs to include not only the initial state of the underlying distributed system, but also the initial values that only depend on the control algorithm. T_{Init} takes an initial state of the underlying distributed system, changes its form by adding the latter initial values to the initial state.

```
op T-Init : Module -> Module .
op T-Init-Ops : OpDeclSet -> OpDeclSet .
op T-Init-Eqs : EquationSet -> EquationSet .
eq T-Init(M) = setOpsEqs(M, T-Init-Ops(getOps(M)), T-Init-Eqs(getEqs(M))) .
```

where `getOps` (`getEqs`) gets the set of operators (equations) in `M`. `T-Init-Ops` (`T-Init-Eqs`) converts each constant (equation) that defines an initial state of the underlying distributed system into another that defines the corresponding initial state of the UDS-CA. `setOpsEqs` replaces sets of operators and equations in `M` by the ones obtained by the conversions.

T_{Trans}

A control algorithm may preserve all behaviors of an underlying distributed system, but processes in the system need to do some more actions to perform the control algorithm. The rules of the UDS-CA could be separated into two parts: (1) the UDS&CA part that includes those specific to an underlying distributed system but modified to perform the control algorithm and (2) the control algorithm part that includes those independent from the underlying distributed system. T_{Trans} takes all rules of an underlying distributed system, modifies them and returns all rules of the UDS&CA part as well as rules of the CA part.

```
op T-Trans : Module -> Module .
op T-Trans : Module RuleSet -> RuleSet .
eq T-Trans(M) = setRls(M, t-Trans(M, getRls(M))).
```

where `getRls` gets the set of rules in `M` and `setRls` replaces it by the rules set of both the UDS&CA and CA parts.

T

T is the composition of the three sub-programs:

```
eq T(M) = T-Trans(T-Init(T-State(M))).
```

We give here a practical view of our approach by using Four-Counter algorithm as an example.

State Expression: Each process in the underlying distributed systems on which Four-Counter algorithm is superimposed (UDS-FCA) needs to observe the system locally to

operate Four-Counter algorithm. The pieces of information to be observed by each process are are the numbers of messages sent and received by it and the flag telling if it has sent the two numbers to the observer.
The observer has one of the following three modes:

- *sleep*: it is sleeping (means that Four-Counter algorithm has not yet started);
- *request*: it has started executing Four-Counter algorithm by requesting each process to send it the two above numbers;
- *terminate*: it claims that the system has terminated.

Let `Obser-mode` be the sort for the three modes.

```
sort Obser-mode .
ops request terminate sleep : -> Obser-mode .
```

The pieces of information owned by the observer are a soup pss of process statuses, the number tn of processes in the system, the number x of processes that has sent the two numbers to the observer and the observer mode om, expressed as `(`pss`,` tn`,` x`,` om`)` called global observations. pss stores the process statuses (essentially the numbers of messages sent and received by each process; let `PStatus` be the sort for process statuses) sent by processes to the observer. Let `Global-obser` be the sort for global observations.

```
subsort   PStatus < PStatSet.

op (_: _, _, _) : Pid Nat Nat Nat -> PStatus [ctor] .

op empPStatSet : -> PStatSet [ctor] .
op _ _ : PStatSet PStatSet -> PStatSet [ctor assoc comm id: empPStatSet] .

op _, _, _, _ : PStatSet Nat Nat Obser-mode ->  Global-obser [ctor] .
```

A global state of the UDS-FCA is expressed as `Base-state(`cfg`) Local-OB(`$lpss$`) GlobalOB(`$gobs$`)`, where cfg is a configuration of the underlying distributed system, $lpss$ is a soup of process statuses (referred as the local observation later), storing the pieces of information, such as the numbers of messages sent & received by each process, for processes to perform Four-Counter algorithm (let `PStatSet` be the sort for such soups) and $glbs$ is the global observation, called a meta-configuration. Let `Meta-config` be the sort for it.

```
op Base-state(_) Local-OB(_) Global-OB(_) : Config PStatSet Global-obser ->
   Meta-config [ctor] .
```

Initial States: The initial state of the UDS-FCA for the underlying distributed system shown in Fig. 3 is expressed as follows:

```
op initial : -> MConfig .
eq initial = Base-state((< p: active, t1 >)(< p: passive, t2 >)
(< r: passive, none >) ([p, q: empChan])([p, r: empChan])
([q, p: empChan]) ([q, r: empChan])([r, q: empChan])([r, q: empChan]))
Local-OB((p, 0, 0, 0) (q, 0, 0, 0) (r, 0, 0, 0))
Global-OB(empPStatSet, 3, 0, sleep)
```

State Transitions: Each process preserves its actions in the underlying distributed system but needs to count the numbers sent and received and send them to the observer if required. The rule `sndToken` is modified as follows:

```
rl[sndToken] :
Base-state(< P, active, T ; TL > [P, Q, CS] BC)
Local-OB((P, N1, N2, N3) LB) Global-OB(GB)
=>
Base-state(< P, active, TL > [P, Q, T CS] BC)
Local-OB((P, N1 + 1, N2, N3) LB) Global-OB(GB) .
```

The other rules in the underlying distributed system are modified likewise.

The observer (1) starts Four-Counter algorithm by changing its mode to `request` from `sleep` and (2) judges if the system has terminated when all processes have already sent the numbers of messages sent and received to the observer. (2) is split into two cases: (2.1) the two numbers of messages sent and received are equal and (2.2) the two number are not. (2.1) is described as the following rule:

```
crl[terminate] :
Base-state(BC) Local-OB(LB) Global-OB(LB1, N1, N2, request)
=>
Base-state(BC) Local-OB(LB) Global-OB(LB1, N1, N2, terminate)
if (N1 == N2) and compare(LB1) .
```

where `compare` counts and compares the numbers of messages sent and received.
(1) and (2.2) are described likewise.

We show now how to implement Four-Counter algorithm as T.

The function `T-State`. In addition to the sorts and operators used in an underlying distributed system, we need to use some more sorts, such as `PStatus` and `Meta-config`, and some more operators, such as those used in (*p*: *sms*, *rms*, *f*) and `Base-state(`*cfg*`)` `Local-OB(`*lpss*`)` `GlobalOB(`*gobs*`)`. Those new sorts and operators are added by `T-state`, which has already been introduced. For example, the sort `Obser-mode` and the three constants of the sort are added to a module `M` as follows:

```
addOpe(addSort(M, 'Obs-mode),
op 'sleep : nil -> 'Obs-mode [none] .
op 'request : nil ->  'Obs-mode [none] .
op 'terminate : nil -> 'Obs-mode [none] .) .
```

The function `T-Init`. Let us suppose that there are n processes $\mathtt{p}_1$, ..., $\mathtt{p}_n$. Let cfg_0 be the initial configuration of an underlying distributed system. Then, the initial state of the UDS-FCA is as follows:
`Base-state(`$\mathtt{cfg}_0$`)`
`Local-OB((`$\mathtt{p}_1$`, 0, 0, 0) ... (`$\mathtt{p}_n$`, 0, 0, 0))`
`GlobalOB(empPStatSet,` n`, 0, sleep)`
This conversion is done by `genInitial`.

```
op genIntial : Config -> Meta-config .
eq genIntial(I) = Base-state(I) Local-OB(InitLocal(I))
                  Global-OB(InitGlobal(I)).
```

where `InitLocal` and `InitGlobal` initialize the local and global observations.
By convention, a term representing an initial state is referred by a constant. For example, we declare an equation "`eq init =` $\mathtt{cfg}_0$`.`", where `init` becomes a constant of `Config`. Such an equation is converted to another one such that `init` is a constant of `Meta-config` and $\mathtt{cfg}_0$ becomes what has already been described. Let us suppose `cfg2mcfg` be in charge of the conversion of `init`. `T-Init-Eqs` is defined as follows:

```
eq T-Init-Eqs(eq T = T' [AtS]. EqS) = if (getType(T) == 'Config) then
(eq cfg2mcfg(T) = upTerm(genIntial(downTerm(T', empConfig))) [AtS].)
T-Init-Eqs(EqS) else (eq T = T' [AtS].) T-Init-Eqs(EqS) fi.
eq T-Init-Eqs(none) = none .
```

where "`eq T = T' [AtS] .`" is a meta-representation of an equation, namely a term of the sort `Equation`. If the equation is used to define a constant as an initial state of an underlying distributed system, the equation is converted to the one used to define a constant as an initial state of the UDS-FCA.

The function `T-Trans`. `T-Trans` first constructs the rules in the underlying distributed system&FCA part from the rules for the underlying distributed system. We have described how to convert the rule `sndToken`. The one of the equations that define `T-Trans` in charge of this conversion is as follows:

```
ceq cl-Trans(M, rl T => T' [AtS] . RlS) =
(rl 'Base-state`(_`) Local-OB`(_`) Global-OB`(_`)
[T, '__['LB:ListPair,'_`,_`,'_`,_['P:Pid,'N1:Nat,'N2:Nat,'N3:Nat]], 'GB:Global-obser]
 =>
'Base-state`(_`)Local-OB`(_`)Global-OB`(_`)[T', '__['LB:ListPair,'_`,_`,'_`,_[
'P:Pid,'_+_['N1:Nat,'s_['0.Zero]],'N2:Nat, 'N3:Nat]],'GB:Global-obser] [AtS] .)
cl-Trans(M, RlS)
if (checkPart(T, T') == 'send) .
```

where "`rl T => T' [AtS] .`" is the meta-representation of a rule and `checkPart` judges what a rule belongs to based on the left- & right-hand sides of the rule. The equations conducts the conversion as mentioned. If the rule `rl T => T' [AtS].`" is `SendToken`, then

the equation returns the meta-representation of the rule obtained by the conversion from the rule. In the meta-representation, for instance, `'_+_['N1:Nat, 's_['0.Zero]]` is the meta-representation of the term `N1 + (s 0)` where `N1` is a variable of `Nat` and `s_` is the successor function of natural numbers.

Lastly, `T-Trans` adds to the module all the remaining rules in the FCA part, which are constructed regardless of any underlying distributed system. For example, it adds the above `terminate` rule.

The function `T`. `T` just composition of the three functions as described.

4.1.2 Model Checking at the Meta Level

In our approach, the model checking is conducted at the meta-level as well. For the metarepresentation of a system module in which a state machine is specified, Maude makes it possible to search its reachable states for states such that some requirements are satisfied, which is done by the operator `metaSearch`. Given the metarepresentation M of a module, a starting term t_1, a pattern term t_2, a condition term c, the kind k of search, and a bound value b and a natural number n, the search `metaSearch(`M`,`t_1`,`t_2`,`c`,`k`,b,`n`)` searches the reachable state space from t_1 in a breadth-first manner with the maximum depth b of the search for at most n states that match t_2 such that c holds. When t_1 is an initial state of a state machine and the negation of a state predicate p is expressed as t_2 and c, the search conducts the model checking that p is an invariant property of the state machine.

```
op metaSearch :  Maude Term Term Condition Qid Bound Nat ~>
ResultTriple? [special(...)] .
```

The result of the function is a term of the sort `ResultTriple?`.

```
sorts  ResultTriple ResultTriple?
subsort ResultTriple < ResultTriple?
op failure : -> ResultTriple? [ctor] .
op {_,_,_} : Term Type Substitution -> ResultTriple [ctor] .
```

The result can be a failure denoted as a constant `(failure).ResultTriple?` of the sort `ResultTriple?` that means any such patterns are not reachable from the start term, or a term of the sort `ResultTriple`, which is a subsort of the sort `ResultTriple?`, consisting of the searched term, the type of the searched term and the substitution that matches the searched term with the pattern.

We demonstrate how to use `metaSearch` by model checking that UDSs-FCA for some underlying distributed system enjoy the property (called the termination property) that whenever the observer claims that the underlying distributed system has terminated, it has surely done. To this end, we define the predicate `unTerm` judges if an underlying distributed system has terminated or not:

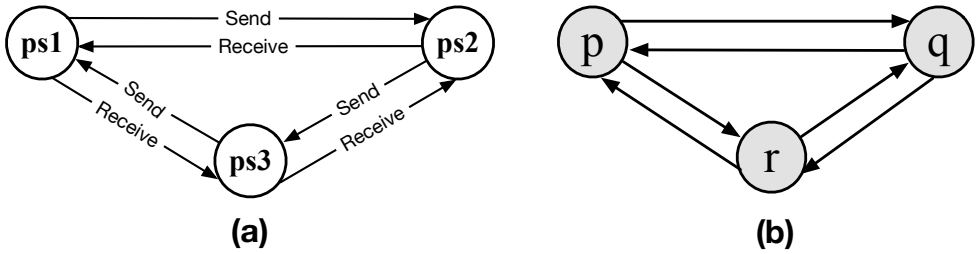


Figure 4.2: The state-transition diagram of a process.

```
eq unTerm(Base-state(BC) Local-OB(LB) Global-OB(OB))
= (not passive(BC) or not checkMsg(BC)).
```

where `passive` returns `true` if all processes are passive and `false` otherwise, and `checkMsg` returns `true` if all channels are empty and `false` otherwise. `unTerm`, therefore, returns `false` iff all processes are passive and all channels are empty, and `true` otherwise.
The model checking is performed as follows:

```
metaSearch(T(upModule('SYS, false)), 'initial.MConfig, 'MC:MConfig,
unTermCon('MC:MConfig) = 'true.Bool, '*, unbounded, 0) .
```

where `unTermCon` is the meta-representation of `unTerm`, `'*` means that all reachable states from the given initial state (including the initial state) are candidates, `unbounded` means that all the reachable states are traversed and `0` means that the search tries to find all states that matches a variable of `MConfig` such that `unTerm` holds.

4.1.3 Experiments

We conduct model checking experiments for two underlying distributed systems. The first is the system specified in Section 4.3.1 (shown in Fig. 2.2) and the second is another one in which each process has three statuses ps1, ps2 and ps3. For the second system, a process may change its status when it sends or receives a message; it changes its status to either ps1 or ps3 when its status is ps2 and it is not able to send a message when its status is ps3. A process turns to passive after it sends a message or its status is ps3. The state transitions of a process are depicted in Fig. 4.2(a) and we make a model for the concrete system in Fig. 4.2(b) with three processes and the initial state in which one is active. An advantage of our method is demonstrated by that we do not need to specify the UDS-FCA for each underlying distributed system. All what we need to do is specifying only underlying systems. The two underlying systems are specified as the modules `SYS1` and `SYS2`, respectively, and `T` takes each module as an input, generating `T(SYS1)` and `T(SYS2)` as the specifications of the two UDS-FCAs.

As the results of our model checking, counterexamples are found. This implies that the Four-Counter algorithm does not work correctly. The counterexample for `SYS1` is depicted as shown in Fig. 4.3. Intuitively, the observer cannot conclude from S = R that the computation has terminated. This is due to the asynchrony of communication. All

processes may not receive or reply to the request from the observer at the same time. According to the counterexample, p receives a request after it sends t1 to q and turns to passive. It sends $(p, 1, 0)$ back. q receives a request before it receives the token t1 from p. It sends $(q, 0, 0)$ back. It then sends t1 to p and t2 to r. r receives a request after it receives and comsumes t2 and then it sends $(r, 0, 1)$ back. The observer counts the numbers of the messages sent and received when it has already received all the answers from the three processes. Because $S = 1$ and $R = 1$, the observer claims that the system has terminated while the computation has not been actually terminated because the channel from q to p is carrying t1.

Counterexamples were also found for `T(SYS2)`.

4.2 Generating and Model Checking All Possible Initial States

We have illustrated in Section 4.1.2 how to conduct model checking experiments of a control algorithm for concrete underlying distributed system. Note that even for a fixed number of each kind of entities, such as processes, there may be multiple initial states and some initial states may lead to a counterexample for a property, while some may not. It is better to model check as many as possible initial states as shown in Fig. 4.4(a). However, it is time-consuming and mistake-prone to create all possible initial states of a system and model check them by human beings.

4.2.1 Generating All Possible Initial States

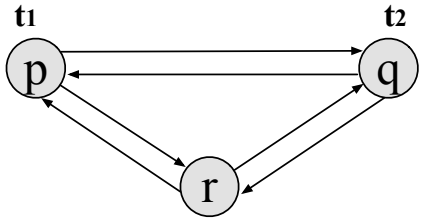
It suffices to use some parameters, such as the number of processes, to systematically generate all initial states of a system such that some conditions are fulfilled, for example, that the network is fully connected. The idea is depicted in Fig. 4.4(b). For a system that has 3 processes fully connected and 3 tokens, for example, all possible initial states are shown in Fig. 4.5. For another system that has 3 processes partially connected and 4 channels, some possible initial states are shown in Fig. 4.6. Note that all processes play the same role in these systems.

For the underlying systems of Four-Counter algorithm, for example, the function to generates all such initial states could be implemented as follows:

```
op InitGen : Nat Nat -> ConfigSet .
```

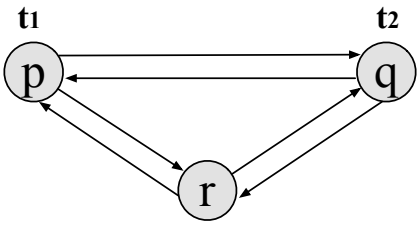
where `ConfigSet` is the sort for system configuration soups.

`InitGen` takes the number of processes and the number of the tokens and returns all possible initial states of the system. Note that we have supposed that the underlying system network is fully connected.



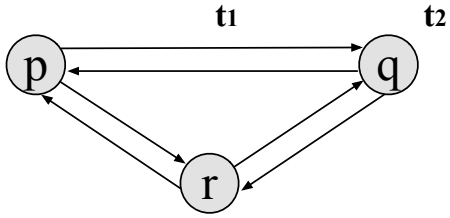
(a)

Global-OB(empPStatSet, 3, 0, sleep)



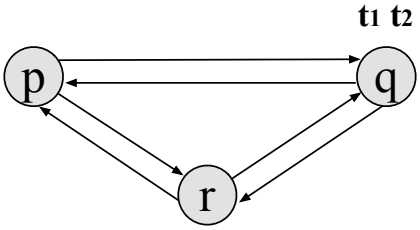
(b)

Global-OB((q, 0, 0, 0), 3, 1, request)



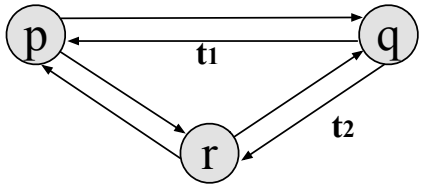
(c)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
3, 2, request)



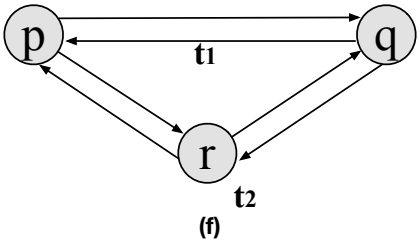
(d)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
3, 2, request)



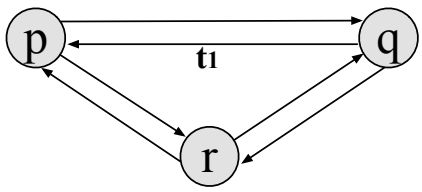
(e)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
3, 2, request)



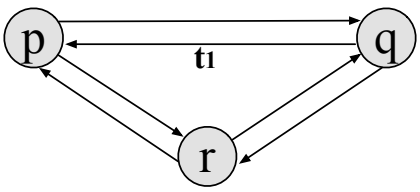
(f)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
3, 2, request)



(e)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
(r, 0, 1, 0), 3, 3, request)



(f)

Global-OB((q, 0, 0, 0) (p, 1, 0, 0),
(r, 0, 1, 0), 3, 3, terminate)

Figure 4.3: The counterexample found

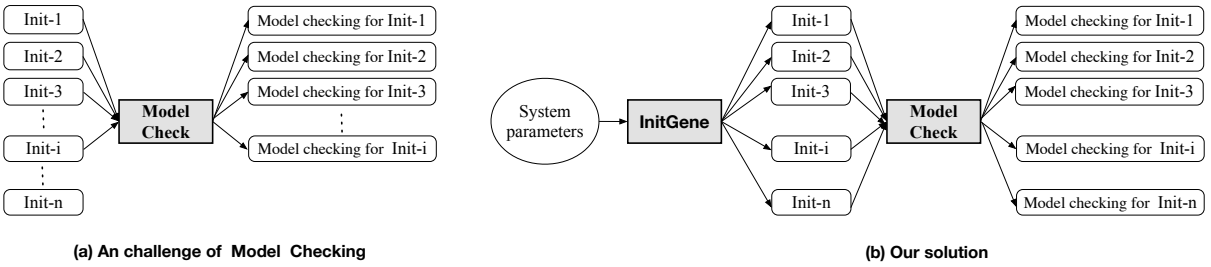


Figure 4.4: A challenge of model checking and our solution to model check *control algorithms*. Init-1 ... Init-i ... Init-n are initial states that represent concrete system. The function **InitGene** takes some system parameters of an underlying distributed system as inputs and returns all possible initial states of the underlying distributed system.

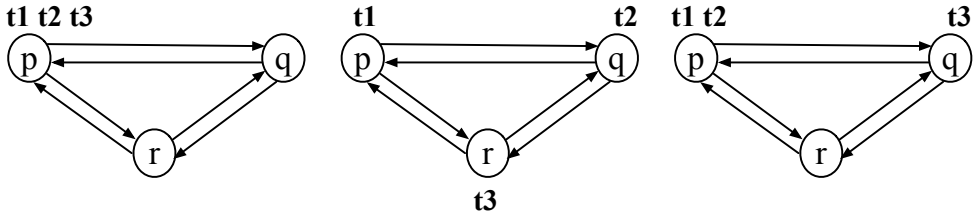


Figure 4.5: All possible initial states of a fully connected system.

4.2.2 Model Checking All Possible Initial States

Once all possible initial states of a system have been generated, the next to do is to conduct model checking experiments for all of them with `checkAll` defined as follows:

```
op checkAll : Nat Nat -> CounterSet .
op checkList : ConfigSet -> CounterSet .
eq checkAll(N1, N2) = checkList(InitGene(N1, N2)) .
eq checkList(C Cs) = modelCheck(upTerm(init(C))) and checkList(Cs) .
eq checkList(empConfigSet) = empCounterSet .
```

where `N1` & `N2` are variables of `Nat`, `C` is a variable of `Config`, `Cs` is a variable of `ConifgSet`, `CounterSet` is the sort for counterexample soups, `modelCheck` conducts model checking for one configuration (which is defined in the same way as described in Section IV-B) and `checkList` conducts model checking for a configuration set.

`checkAll` first calls `InitGen` to make all initial states for given parameters and then conducts model checking for all of them. An algorithm is more likely to enjoy a desired property if no counterexample is found for any of those initial states. In case time taken is considered and it is sufficient to detect only one counterexample, `checkList` could be modified such that we can stop model checking as soon as a counterexample is found.

4.2.3 Experiments

For Four-Counter algorithm, we conduct experiments for six underlying systems, which are described by the numbers of processes and tokens as shown in Table. 4.1. The number

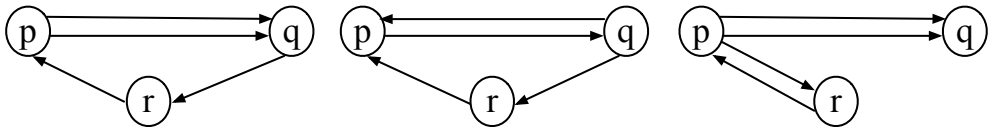


Figure 4.6: Some possible initial states of a partially connected system.

of all possible initial states for each system is presented in the fourth column. Thanks to the result of `checkAll`, we know that counterexamples are not found for any initial states. Let us consider the system that includes three processes and two tokens. Some possible initial states of the systems are shown in Fig. 4.7.

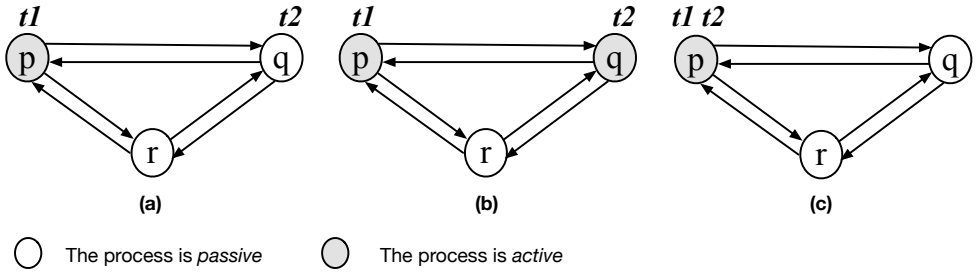


Figure 4.7: Some initial states of the system with 3 processes & 2 tokens.

We have shown in the previous section that a counterexample is found when we model check the initial state **(a)**. This means that the algorithm does not work correctly. However, model checking for the initial state **(b)**, there is no counterexample found. Generally, an counterexample may appear in some initial states, but not in some others. Assuming we only model check for **(b)**, but not **(a)**, the counterexample is not found and the error is not detected. The experimental results are shown in Table. 4.1. The experiments were conducted on a computer that carried a 2.9GHz Intel Xeon E5-4655 processor with 256GB of RAM. The fifth column presents the numbers of initial states from which the counterexample is found. Moreover, the times taken to find the first of these initial states are shown in the *time taken* column.

In summary, checking all possible initial states of a system increases the capacity of counterexample detection or improves the confidence in the correctness of an algorithm. Another example to demonstrate this point is shown in Section 4.3.2.

4.3 Case Studies

We apply our approach to specifying and model checking Chandy-Lamport algorithm and Checkpointing algorithm, which have already been described in Section 2.1.1. The first case study demonstrates that our approach makes it faithful to model check a property that involves both an underlying distributed system and the underlying distributed system on which Chandy-Lamport algorithm is superimposed (the distributed snapshot reachability property). The second case study shows the usefulness of our idea to specifying and

Table 4.1: Results of the nine model checkings for Four-Counter Algorithm

System	**The number of Processes**	**The number of Tokens**	**All Initial states**	**Counter-Initial States**	**Time-Taken (Minutes)**
Sys 1	2	2	3	1	0.001
Sys 2	3	2	5	2	5.563
Sys 3	3	3	14	5	35.867
Sys 4	3	4	38	16	49.765
Sys 5	4	2	7	2	62.378
Sys 6	4	3	19	7	158.459
Sys 7	5	2	12	4	204.845
Sys 8	5	4	72	28	421.642
Sys 9	6	3	31	10	642.851

model checking all possible initial states.

4.3.1 Specification and Model Checking of Chandy-Lamport Algorithm

The Chandy-Lamport distributed snapshot algorithm (Chandy-Lamport algorithm)[51] is the first distributed snapshot algorithm. The algorithm should satisfy distributed snapshot reachability property as follows. Let s_1, s_* and s_2 be the state in which Chandy-Lamport algorithm initiates (the start state), the snapshot taken, and the state in which Chandy-Lamport algorithm terminates (the finish state), respectively. Although s_* may not be identical to any of the global states that occur in the computation from s_1 to s_2, one desired property Chandy-Lamport algorithm should satisfy is that s_* is reachable from s_1 and s_2 is reachable from s_*, whenever Chandy-Lamport algorithm terminates. Note that s_1, s_2 and s_* are states of the underlying distributed system but not those of the underlying distributed system on which Chandy-Lamport algorithm is superimposed (UDS-CLDSA). The distributed snapshot reachability property relates to two different systems: an underlying distributed system and the UDS-CLDSA. It is not straightforward to express the property in typical existing temporal logics, such as linear temporal logic (LTL) and computation tree logic (CTL). This is because the semantics of such a logic is defined for a single system formalized as a Kripke structure (an extension of a state machine). As a result, few research on model checking Chandy-Lamport algorithm and any other distributed snapshot algorithms have been conducted. Model checking Chandy-Lamport algorithm means model checking UDS-CLDSAs in this thesis. To the best of our knowledge, there are only two case studies in which Chandy-Lamport algorithm has been model checked [7][57]. In [7], SPIN is used to model check that a system (in which there are two processes and one channel from one process to the other) on which Chandy-Lamport algorithm is superimposed enjoys a property that is different from the distributed snapshot reachability property. In [57], the authors have specified

UDS-CLDSAs and model checked that they satisfy a property, which they claim is the distributed snapshot reachability property in Maude. However, the property is not exactly the same as the distributed snapshot reachability property. It is encoded in the Maude search command and does not reflect the informal description of the property originally given in [51]. In addition, their way to model check is to compare the numbers of solutions obtained by three search experiments. Hence it is not straightforward to construct a counterexample when the property is not fulfilled.

The Specification of an Underlying Distributed System

The system shown in Fig. 4.8 is used as an example. The state of a process only depends on the set of tokens owned by the process. Each process has two actions: (1) sending a token to another by putting the token to one of its outgoing channels if it holds the token and (2) receiving a token from one of its incoming channels if any.

State Expression. Let `Pid`, `PState` & `Token` be the sorts for process identifiers, process states & tokens, `p`, `q` & `r` be the constants of `Pid`, and `t1` & `t2` be the constants of `Token`. Let `TknSet` be the sort for token soups. Let `Msg` be the sort for messages and the supersort of `Token`. Thus, tokens are messages. The state of a channel is expressed as a queue of messages for which the sort `MsgQueue` is used. The empty queue of messages is denoted as `empQ`. A queue of n messages $m_0, m_1, \ldots, m_{n-1}$ in this order is denoted as m_0 | m_1 | ... | m_{n-1} | `empQ`. Process states and channel states are expressed as observable components for which the sort `OCom` is used: `(p-state[`p`]:` ts`)` for process, where `p-state[`p`]:` is the name, p is a process and ts is the value and a token soup and `(c-state[`p`,`q`,`n`]:` ms`)` for channel, where `c-state[`p`,`q`,`n`]:` is the name, p & q are the source and destination processes, n is a natural number used to identify the channel because there may be more than one channel from p to q and ms is the value that is a message queue. For this system, one channel from p to q is identified by 0 and the other is by 1. The system state is expressed as the soup of the observable components for which the sort `Config` is used.

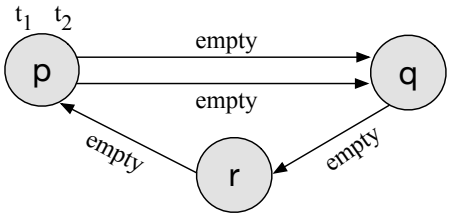


Figure 4.8: The initial state of a token system

```
sorts Pid Msg Token PState MsgQueue OCom Config .
subsort Token < PState .
subsort Token < Msg .
subsort OCom < Config .
```

```
ops t1 t2 : -> Token .
ops p q r : -> Pid .

op empPState : -> PState [ctor] .
op _ _ : PState PState -> PState [ctor assoc comm id: empPState] .

op empQueue : -> MsgQueue [ctor] .
op _|_ : Msg MsgQueue -> MsgQueue [ctor] .

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

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

Initial States. In this system, there is only one initial state in which p holds the two tokens t_1 and t_2, the other processes do not and all the channels are empty. Let the constant `initial` of `Config` equal the following term:

```
(p-state[p]: t1 t2) (p-state[q]: empTknSet) (p-state[r]: empTknSet)
(c-state[p, q, 0]: empQ) (c-state[p, q, 1]: empQ)
(c-state[q, r, 0]: empQ) (c-state[r, p, 0]: empQ) .
```

State Transitions. The two actions of a processes are described as follows:

```
rl[sndToken] :
(p-state[P]: T PS) (c-state[P, Q, N]: CS) BC =>
(p-state[P]: PS) (c-state[P, Q, N]: enq(CS, T)) BC.

rl[recToken] :
(p-state[P]: PS) (c-state[Q, P, N]: T | CS) BC =>
(p-state[P]: T PS) (c-state[Q, P, N]: CS) BC .
```

where where `P`, `Q`, `T`, `PS`, `CS` and `BC` are variables of `Pid`, `Pid`, `Token`, `TknSet`, `MsgQueue` and `Config`, respectively. `enq` is a standard function for queues, taking a queue q and an element e and putting e into q at bottom.
Let `SYS-TK` be the module in which the system is specified.

Specifying Chandy-Lamport Algorithm as a Meta-program

Let `cl` be the meta-program. Its three sub-functions are `cl-State`, `cl-Init` and `cl-Trans`.

The function `cl-State`**.** Among messages in the UDS-CLDSA are not only data messages but also *markers*. Moreover, to describe the UDS-CLDSA, we need to express more information, such as the start state, the snapshot, the finish state and the information to control behaviours of the algorithm. Thus, we need to use new notations to express the states of the system. For example, we need to use a new sort `MMsg` to denote tokens and markers. The sort for the states of the UDS-CLDSA is `MConfig` and a state is expressed as follows:

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

where *bc*, *sc*, *ssc*. *fc* and *ctl* are terms that represent an underlying distributed system state, a start state, a snapshot, a finish state and the information to control the algorithm, respectively.
`cl-State` modifies a given set of notations for an underlying distributed system, generating a set of notations for the UDS-CLDSA. The function `cl-State` will take into account the syntax part of the module. All what to do is replacing all notations for describing an underlying distributed system by all notations for describing the UDS-CLDSA. It also replaces the sorts of constants and variables in the input module by the corresponding sorts for a UDS-CLDSA, e.g replacing `MsgQueue`, which is the sort for the states of channels in an underlying distributed system, with `MMsgQueue`, which is the sort for the states of channels in the UDS-CLDSA. To this end, we define several functions. For example, the following function to change the sort of a constant of the sort `Config`, which is the sort for states of an underlying distributed system, to be `MConfig`, which is the sort for states of the UDS-CLDSA.

```
op reConfigToMConfig : Constant -> Constant .
ceq reConfigToMConfig(C) = qid(string(getName(C))  + "." + "MConfig")
   if getType(C) == 'Config .
```

The function takes as an argument a constant, and changes the sort of the constant to `MConfig` if the sort of the constant is `Config`.

The function `cl-Init`**.** Each initial state of the UDS-CLDSA can be obtained from an initial state of an underlying distributed system. Specifically, for each initial state *s* of an underlying distributed system, the corresponding initial state of the UDS-CLDSA is *base-state(s) start-state(empBConfig) snapshot(empBConfig) finish-state(empBConfig) control(t)*, where *empBConfig* denotes the empty soup of the observable components and *t* is a term in which all information to control behaviours of Chandy-Lamport algorithm is initialized. This is done by the function named `genIntial`. For a given initial state `BC` of an underlying distributed system, `genInt` generates the corresponding initial state of the UDS-CLDSA as follows:

```
op genInit : Config -> MConfig .
eq genInit(BC) = base-state(BC) start-state(empConfig) snapshot(empConfig)
finish-state(empConfig) control(InitCtlConfig(BC)).
```

where the function `InitCtlConfig` initializes values for all control information components.

The function `cl-Init` uses `genInit` to convert each equation for initial states of an underlying distributed system to one for those of the UDS-CLDSA. The function `cl-Init` is defined as follows:

```
eq cl-Init(M) = setEqSet(M, cl-Init(M, getEqs(M))).
eq cl-Init(M, eq T = T' [AtS] . EqS) =
  if (getType(T) == 'Config)  then  (eq reConfigToMConfig(T) =
  upTerm(genIntial(downTerm(T', empConfig)))  [AtS] .) cl-Init(M, EqS)
  else (eq T = T' [AtS] .) cl-Init(M, EqS) fi .
```

where the function `getEqs` gets the set of equations of the module and the function `setEqSet` replaces all current equations of the module by another set of equations.

The function `cl-Trans`. `cl-Trans` first constructs the rules in the UDS&CLDSA part based on those in an underlying distributed system. From the rule `sndToken`, for example, the following one is generated:

```
rl : base-state((p-state[P]: T PS)(c-state[P, Q, N]: CS) BC)
=>
base-state((p-state[P]: PS)(c-state[P, Q, N]: enq(CS, T)) BC) .
```

The function `cl-Trans` deals with this case as follows:

```
ceq cl-Trans(M, rl T => T' [AtS] . RlS) =
(rl 'base-state[T] => 'base-state[T'] [AtS] .)  cl-Trans(M, RlS)
if (checkPart(T) == 'UDS&CLDSA1) .
```

The function `cl-Trans` then generates all rules in the CLDSA part.

The function `cl`. `cl` combines the three functions together.

```
eq cl(M) = cl-Trans(cl-Init(cl-State(M))) .
```

Model Checking of the Distributed Snapshot Reachability Property

The existing model checking approach in [57] uses Maude search command to model check a property for Chandy-Lamport algorithm. The result of the model checking is based on the numbers of solutions obtained by three search experiments. The disadvantage of this method is that it is difficult to find a counterexample in case the algorithm does not satisfy the property. Moreover, the property they have used is not exactly the same as the distributed snapshot reachability property. A faithful formal definition of the property has been given in [29]. It is checked that Chandy-Lamport algorithm is terminated in the UDS-CLDSA, but it is checked that some states of an underlying distributed system

are reachable from some others in the underlying distributed system but not the UDS-CLDSA. Accordingly, the property involves two systems, an underlying distributed system and the UDS-CLDSA.

We propose a new model checking method by which a counterexample is shown when the algorithm does not enjoy a property. Our method directly deals with the faithful formal definition of the property. The essential idea of our method is that `metaSearch` is used to find any taken snapshots that satisfy the *unReachable* condition as follows: either the snapshot is not reachable from the start state or the finish state is not reachable from the snapshot. If there is no such snapshot taken then the algorithm is likely to satisfy the property. Since the *unreachable* predicate is not built-in Maude, we define it. Given a state machine M and two states s_1 and s_2, s_2 is said to be *unreachable* from s_1 iff s_1 cannot go to s_2 by any state transition steps in M. We implement this predicate by using `metaSearch`. Assuming that `M`, `T1` and `T2` are the metarepresentation of the module that specifies M, the term that expresses s_1 and the term that expresses s_2, respectively, we define the predicate *unreachable* as follows:

```
op unReachable : Module Term Term  -> Term .
eq unReachable(M, T2, T1) = if metaSearch(M, T1, T2, nil,'*, unbounded, 0)
== (failure).ResultTriple? then 'true.Bool else 'false.Bool fi .
```

The model checking is performed as follows:

```
metaSearch(cl(upModule('SYS-TK, false)), 'initial.MConfig,
'__['MC:MConfig,'start-state['SC:Config],'snapshot['SSC:Config],
'finish-state['FC:Config]], '_!=_['FC:Config, 'empBConfig.Config]
= 'true.Bool /\ unReCon(upModule('SYS-TK, false), 'SC:Config,
'SSC:Config, 'FC:Config) = 'true.Bool, '*, unbounded, 0)
```

Termination is detected with respect to (wrt) `cl(upModule('SYS-TK, false))`, the UDS-CLDSA, while reachability is checked wrt `upModule('SYS-TK, false)`, the underlying distributed system. `unReCon` takes the meta-representations of an underlying distributed system, a start state `SC`, a snapshot `SSC` and a final state `FC` and use `unReachable` to check if `SSC` is not reachable from `SC` wrt the underlying distributed system or `FC` is not reachable from `SSC` wrt the underlying distributed system. This model checking approach shows counterexamples if any.

Experiments

First, we take the "token system" described in Section 4.3.1. Secondly, we consider the systems in which each process has three statuses ps1, ps2 and ps3, and there are three messages m_1, m_2 and m_3. A process may change its status when it sends a message. The state-transitions of a process are depicted in Fig. 4.9. We call such systems the "switch-message" systems. Lastly, we consider systems called the "multi-token" systems, in which there are one or more tokens. Processes exchange tokens to others or may consume them. For each underlying distributed system, we only need to specify the underlying distributed

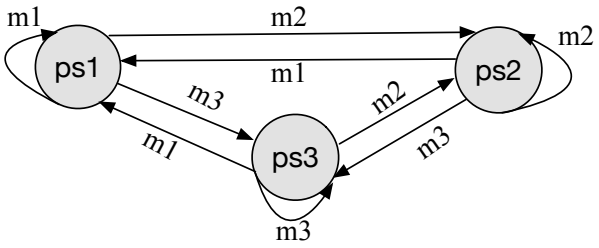


Figure 4.9: The state-transition diagram of a process.

Table 4.2: A comparison of performance between the two approaches

System	**The number of Processes**	**The number of channels**	**The number of tokens**	**Time taken by the existing approach (Minutes)**	**Time taken by the proposed approach (Minutes)**
Sys 1	2	2	1	0.0036	0.0006
Sys 2	2	2	3	0.1472	0.0393
Sys 3	2	3	2	28.7403	0.5373
Sys 4	3	3	2	96.0883	2.3606
Sys 5	3	4	1	0.9459	0.0405
Sys 6	3	4	2	245.9765	5.9551
Sys 7	3	5	2	985.5643	52.6838
Sys 8	4	8	1	589.3522	32.1800

system and `cl` automatically generates the specification of the UDS-CLDSA. There is no counterexample found.

We compare the performance of our proposed method with the existing one [57]. To this end, we consider to model check the distributed snapshot reachability property for the "multi-token" systems, which have been specified in the existing study. We conduct model checking for eight different systems. We take into account the time to perform the model checking. Experiments were conducted on a computer that carries a 4GHz Intel Core i7 processor with 32GB of RAM. The results are shown in Table 4.2. A system is described by the three factors that are the number of processes, the number of channels and the number of tokens. The experiment time of the system, however, depends on not only these numbers but also its topology.

The results of experiments have shown that our proposed approach surpasses in performance the existing one. The existing method uses three search commands to find: (1) all snapshots taken, (2) all snapshots taken such that the snapshot is reachable from the start state and (3) all snapshots taken such that the finish state is reachable from the snapshot, while our proposed method uses only one search to find all snapshots taken such that the *unReachable* condition is satisfied.

Our model checking shows that Chandy-Lamport algorithm satisfies the distributed

snapshot reachability property for the underlying distributed system used since a counterexample was not found. However, to demonstrate an advantage of our approach that a counterexample is found when the algorithm does not satisfy a property, we propose to check a property that is not satisfied by Chandy-Lamport algorithm. Conveniently, we use a property that is opposed to the distributed snapshot reachability property. We call it UDSR property as follows. Let s_1 be the start state, s_2 be the terminated state, and s_* be the snapshot, then s_* is not reachable from s_1, or s_2 is not reachable from s_*. To model check, we will find any snapshots taken such that the snapshot is reachable from the start state and the finish state is reachable from the snapshot (called *reachable* condition). To this end, we define a predicate `ReCon` for this *reachable* condition. We then define the search to conduct the model checking the UDSR property for Chandy-Lamport algorithm. A counterexample is found and shown.

4.3.2 Formal Analysis of a Checkpointing Algorithm

Checkpointing algorithm [13] requires each p_i of mobile hosts to store the following pieces of information (let n be the total number of mobile hosts and mobile support stations):

- R_i: an array of bits at p_i such that its size is n. $R_i[j] = 1$ means that p_i has received computation messages from p_j in the current checkpointing interval.

- csm_i: an array of checkpoint numbers at p_i such that its size is n. $csm_i[j]$ represents checkpoint numbers of p_j that p_i knows.

- $weight_i$: a nonnegative number whose maximum value is 1. It is used to detect the termination of the algorithm.

- $trigger_i$: a pair (pid, $imun$) at a component p_i; pid is the identification of the last checkpoint initiator for which p_i has processed the request and $imun$ is the initiator's csm.

- $sent_i$: a Boolean at p_i that is set to 1 if p_i has sent messages in the current checkpoint.

- $cp\text{-}state_i$: a Boolean at p_i that is set to 1 if p_i is in the current checkpoint.

- *old-csm*: a number that stores the csm of the current tentative checkpoint.

- CP_i: 4-tuple: (A, B, C, D) where A is the mutual checkpoint of p_i, B is its own R_i before it takes mutual checkpoint, C is the $trigger$ which is associated with the current checkpoint and D is its own $sent_i$ before taking the mutual checkpoint.

- Mr: a soup that stores the information of csm_i and R_i for a component (a mobile host or a mobile support station) to decide whether it should inherit a checkpoint request or not.

Note that 0 and 1 are used as the Boolean values in the specification.

A component could start Checkpointing algorithm at anytime when it is not in any checkpoint process. It takes its local checkpoint, increases its $csm_i[i]$, sets $weight_i$ to 1 and $cp\text{-}state_i$ to 1 and stores its own identifier and the $csm_i[i]$ to $trigger_i$. It then sends checkpoint requests to all of its relative components j such that $R_i[j]$ =1 (meaning that the component from which it has received messages). Each request carries the trigger of the initiator and the portion of weight of the initiator, whose weight is decreased by an equal amount. When a component receives a checkpoint request, it checks whether it needs to inherit the request. If it does not, it sends a reply to the initiator. Otherwise, it deals with the request: either takes a new local checkpoint or makes the mutual checkpoint tentative, sends a reply message to the initiator and then propagates the request to all its relatives. When a component sends a computation message, it adds more information to the message: its own $csm_i[i]$ and $trigger_i$ if it is in a checkpoint process. Each component that receives a computation message checks whether it needs to take a mutual checkpoint or not. A mutual checkpoint is turned to a tentative checkpoint when the component receives a checkpoint request. Otherwise, the mutual checkpoint will be discharged. When an initiator receives all replies from its relative components, it announces the termination of the checkpoint process and sends a broadcast message to all components in the system.

The Specification of an Underlying Mobile Distributed System (UMDS)

State Expression. There are three observable components in the systems: mobile hosts, mobile support stations and channels. Let `Mid`, `MidSet` & `Mssid` be the sorts for mobile identifiers, soups of mobile identifiers & station identifiers and `Id`, which is a super sort of `Mid` and `Mssid` be the sort for the union set of mobile and station identifiers. The connection state of a mobile host is either *disC* denoting that it is disconnected from the network or the MSS's identifier to which it is connected. The sort `MState` is used for the connection states of an mobile host.

A message is expressed as (p, q, d), where p is the source process, q is the destination process, and d is data carried by the message. Let `Data` & `Msg` be the sorts for data & messages.

```
sort Msg .
op (_,_,_) : Id Id Data -> Msg [ctor] .
```

The sort `MsgQueue`, queues of messages, is used for the states of channels. The three observable components are: <*mb*: *loc*, *cos*> for mobile hosts, where the *name mb* is a mobile identifier and the value "*loc*, *cos*" is its local state and its connection state, {*mss*: *mbl*, *ms*} for stations, where the name *mss* is its identifier and the value "*mbl*, *ds*" is the set of mobile identifiers, which are connected to it and *ms* is the queues of incoming messages before processing, and [*sp*, *ds*: *st*] for channels, where the name "*sp*, *ds*" is composed of two parameters *sp* and *ds* that are its source and destination processes, respectively, and its state. The sorts `Mobile`, `Station` and `Channel` are used to denote mobile, station and channel observable components, respectively. The state of a system is a soup of observable components and the sort for it is `Config`.

```
sorts Mobile Station Channel Config .
subsort Mobile Station Channel < Config .

op <_:_,_> : Mid LState MState  -> Mobile [ctor] .
op {_:_,_} : Mssid MidSet MsgQueue -> Station [ctor] .
op [_,_:_] : Id Id MsgQueue -> Channel [ctor] .

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

State Transition. A mobile host could do three kinds of actions: (1) sending messages, (2) receiving messages and (3) disconnecting or reconnecting to the network. The following rule specifies the action when a mobile host sends a computation message to another mobile host located in another cell by sending the message to its local station: in what follows, `MB1, MB2` $\in$ `Mid`, `MSS1, MSS2` $\in$ `Mssid`, `CS1, CS2` $\in$ `MState`, `D` $\in$ `Data`, `C` $\in$ `Config`, `LMS` $\in$ `MsgQueue` and `MS` $\in$ `Msg` are variables of those sorts.

```
crl [sendMobile] :
<MB1, D CS1, MSS1> <MB2, CS2, MSS2>[MB1, MSS1, LMS] C
=>
<MB1, CS1, MSS1> <MB2, CS2, MSS2> [MB1, MSS1, enq(LMS, (MB1, MB2, D))] C
```

A mobile support station can extract a message from one of its incoming channel if any and then puts it at the end of its message queue, which is specified as the following rule:

```
rl [receiveStation] :
{MSS1, IDL1, LMS1} [ID, MSS1, MS | LMS] C}
=>
{MSS1, IDL1, enq(LMS1, MS) } [ID, MSS1, LMS] C} .
```

The other actions can be specified likewise.

Let `SYS-MB` be the module in which the system is specified.

Specifying Checkpointing Algorithm

Let `cp` be the meta-program as the formal specification. It has three sub-functions `cp-State`, `cp-Init` and `cp-Trans`.

The function `cp-State`. There are three kinds of control messages: "request," "reply" and "broadcast" messages, which contain information needed to be exchanged by components to perform Checkpointing algorithm. `Msg-request`, `Msg-reply` and `Msg-broadcast` are used as the sorts for those messages. Because a computation message sent by a component p_i needs to carry more information, namely $csm_i[i]$ and $trigger_i$, a new form of computation messages, which are called meta-computation messages, is utilized, for which the sort `Mdata` is used. Both control messages and meta-computation messages are

called meta-messages, for which the sort `M-msg` is used. Let `M-msgQueue` be the sort for meta-message queues. Thus, the expressions of the local states of mobile, station and channel are different from the ones of a UMDS. The sorts `mobile` , `station` and `channel` are replaced by the sorts `M-mobile`, `M-station` and `M-channel`, respectively. Each mobile (or station) observable component needs to to be added some pieces of information to perform Checkpointing algorithm. Such revised observable components are called meta-observable components and expressed in the form of `m-pro(`$ms\ ar\ cs\ we\ tr\ se\ cs\ ol\ cp\ ms1\ ms2\ cS$`)`, in which ms is an element of `Meta-state`, which is a super sort of `M-mobile` and `M-station`, ar, cs, we, tr, se, cs, ol and cp are R_i, csm_i, $weight_i$, $trigger_i$, $sent_i$, $cp\text{-}state_i$, $old\text{-}csm$ and CP_i, respectively, $ms1$ and $ms2$ are soups of messages that the component has sent and received, and cS is a list of tentative checkpoints. The meta-observable component of a channel is in the form of `m-chan(`cn`)`, in which cn is an element of `M-channel`. A global state of a UMDS on which Checkpointing algorithm is superimposed (UMDS-CP) is expressed as a soup of meta-observable components. The corresponding sort is `M-config`. Some notations used in the specification of a UMDS, such as `Mid`, `Id` and `Msg`, are still used in the specification of the UMDS-CP. However, some notations are no longer used, such as `Channel` that is replaced with `M-channel`. `cl-State` needs to get rid of all unused notations and adds all the other necessary notations for describing the UMDS-CP.

The function `cp-Init`**.** Each initial state of a UMDS is in the form $<mh_1>,\ldots,<mh_n>$, $\{mss_1\},\ldots,\{mss_m\}$, $[cn_1],\ldots,[cn_k]$, where each $<mh_i>$, $\{mss_j\}$ and $[cn_k]$ is an observable component as described. Each initial local state of mobile host (or mobile support station) in the UMDS-CP is the same as those of the UMDS, but we need to add all the other necessary meta-observable components that have the appropriate initial values. The corresponding initial state of the UMDS-CP is $m\text{-}pro(<mh_1>\ MC_1),\ldots,m\text{-}pro(<mh_n>\ MC_n)$, $m\text{-}pro(\{mss_1\}\ SC_1),\ldots,\ m\text{-}pro(\{mss_m\}\ SC_m)$, $m\text{-}chan([cn_1]),\ldots,m\text{-}chan([cn_k])$, where $MC_1,\ldots,\ MC_n$ and $SC_3,\ldots,\ SC_n$ are terms in which control information is initialized. This is done by `initial` as follows:.

```
op initial : Ocom -> Meta-com.
eq initial(MB) =  pro(MB,  controlI-nit(MB)).
eq initial(MSS) =  pro(MSS, control-Init(MSS).
eq initial(CN) =  meta-chan(CN) .
```

where `control-Ini` initializes all control information.

The function `cp-Init` uses `initial` to convert each equation for initial states of a UMDS to one for those of the UMDS-CP.

The function `cp-Trans`**.** The rules of the UMDS-CP are classified into two parts: UMDS&CP and CP. In the UMDS&CP part, for example, the following is one such rule:

```
crl [sendMobile1] :
{m-pro(<MB1, D CS1, MSS1> R1 CSM[(MB1, N1) LPC] W1 TR1 (Sent: N2)
(CP-state: 1) OLD1 CP1 CM1 CM2 CL1)
```

```
m-pro(<MB2, CS2, MSS2> R2 CSM2 W2 TR2 S2 CS2 OLD2 CP2 CM3 CM4 CL2)
m-chan([ MB1, MSS1, LMS ]) C}
=>
{m-pro(< MB1, CS1, MSS1 > R1 CSM[(MB1, N1) LPC] W1 TR1 (Sent: 1)
(CP-state: 1) OLD1 CP1 ((MB1, MSS1, (MB1, MB2, D)) CM1) CM2 CL1)
m-pro(<MB2, LCS2, MSS2 > R2 CSM2 W2 TR2 S2 CS2 OLD2 CP2 CM3 CM4 CL2)
m-chan([ MB1, MSS1, put(LMS, M-msg((MB1, MB2, D) (MB1, N1) TR1))]) C}  .
```

The rule is generated from the rule `sendMobile` of the UMDS. When a mobile host sends the messages to its local mobile support station, it changes the value of *sent* to "1" (expressed as `(Sent: 1)` in the right-hand side of the above rule, while it is `(Sent: N2)` of the left-hand side, where `N2` may not be 1), which means that the mobile host has sent messages. Such a message contains not only the computation information, but also the information about its *csm* and *trigger*, which are attached as pairs `(MB1, N1)` and `TR1`. `cp-Trans`. deals with this case as follows:

```
ceq cl-Trans(M, rl T => T' [AtS] . RlS) =
(rl downtern(modifyrls(upTerm(T)), empM) =>
downtern(modifyrls(upTerm(T)), empM) [AtS] .) cl-Trans(M, RlS)
if (checkPart(T, T') == 'MUDS) .
```

where `modifyrls` modifies a state of a UMDS as described. For instance, `< MB1, D CS1, MSS1 >` changes to `m-pro(< MB1, D CS1, MSS1 > R1 CSM[(MB1, N1) LPC] W1 TR1 (Sent: N2) (CP-state: 0) OLD1 CP1 CM1 CM2 CL1)`. `< MB1, D CS1, MSS1 >` is kept and `R1 CSM[(MB1, N1) LPC] W1 TR1 (Sent: N2) (CP-state: 0) OLD1 CP1 CM1 CM2 CL1` is not dependent on the state of the UMDS.

`cl-Trans` then adds all rules of the CP part.

The function `cp`. The function `cp` combines the three functions.

```
eq cp(M) = cp-Trans(cp-Init(cp-State(M))) .
```

Model Checking

A checkpointing algorithm needs to guarantee that any recorded global checkpoint is consistent. A consistent checkpoint must contain no *orphan* message that is sent by a process after taking its local checkpoint, but recorded as a received message by another process in its local checkpoint. This is called "*orphan-consistent property.*" This property is used in our model checking to model check for the algorithm.

To model check the property, we define the predicate `checkInCons` that judges if there exists an orphan message in a given state of a UMDS-CP:

```
eq checkInCons(MC) =  checkOrphanMsg(getCP(MC)) .
```

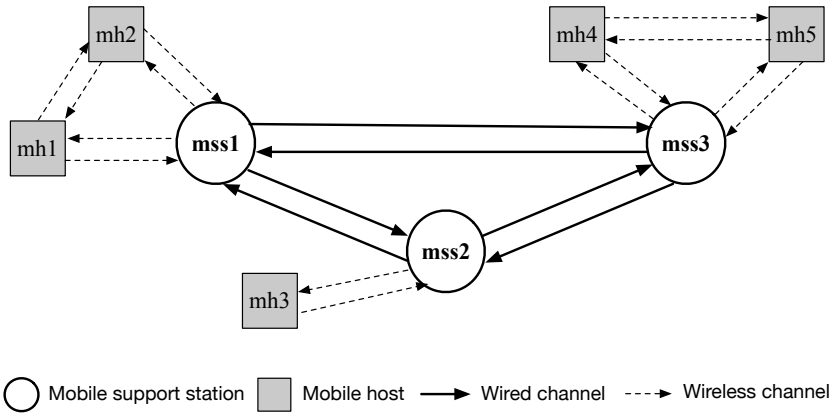


Figure 4.10: A mobile distributed system consists of three mobile support stations and five mobile hosts, in which mobiles directly communicate with other mobiles in the same cell.

where `getCP` gets the current checkpoint of a system, `checkOrphanMsg` checks whether there exists an *orphan* message.
The model checking is performed as follows:

```
red metaSearch(cp(upModule('SYSTEM, false)), 'initial.Meta-config,
'MC:Meta-config, TermCon('MC:Meta-config) /\ InCons('MC:Meta-config),
'*, unbounded, 0) .
```

where **TermCon** and **InCons** are the meta-representations of the conditions to check the termination of the algorithm and "*orphan-inconsistent*" condition.

Experiments

We have conducted model checking for two UMDSs. In the first one, a mobile host is not allowed to send messages directly to other mobile hosts, but to its local mobile support station. The UMDS as shown in Fig. 2.4 is an example. The second system allows mobile hosts to send messages directly to the other mobile hosts that are located in the same cell with it. The system as shown in Fig. 4.10 is an example.

As the results of the model checking, counterexamples are found. Analyzing these counterexamples, we find a minor error in the pseudo-codes of Checkpointing algorithm, which is as follows. If a component p_i propagates the request to all components on which it depends, it may result in a large number of redundant system messages because some of these components may have already received the request messages from other components. The designers of Checkpointing algorithm solve this problem by adding some information saved in the structure Mr. If p_i knows (by checking Mr) that p_k on which it depends has received the request from another component with $req\text{-}csm \geq csm_i[k]$ ($req\text{-}csm$ is saved in Mr), then p_i does not send the request to p_k. The idea is implemented in the procedure `prop-cp` in the pseudo-codes. p_i only sends the request to p_k iff $R_i[k]$ is equal to 1 (meaning that p_i depends on p_k) and $csm_i[k]$ is greater than $Mr[k].csm$ (meaning that p_k has not

Table 4.3: Results of the six model checkings for the algorithm

System	**MSSs**	**MHs**	**All Initial states**	**Counter Initial States 1**	**Time Taken 1 (Seconds)**	**Counter Initial States 2**	**Time Taken 2 (Seconds)**
Sys 1	3	3	2	2	0.507	N/C	0.000
Sys 2	3	5	3	3	3.617	N/C	0.000
Sys 3	4	4	10	10	5.867	3	65.633
Sys 4	4	6	16	16	19.453	3	265.986
Sys 5	5	5	148	148	22.341	52	305.657
Sys 6	5	8	186	186	57.559	74	562.393

received the request from other components). Unfortunately, the algorithm does not work correctly. Let us consider the following case. Based on Checkpointing algorithm, $cms_i[k]$ is initially set to 0. Assuming that p_i has received a computation message from p_k ($R_i[k]$ is set to 1). p_i then starts Checkpointing algorithm. It sets $Mr[k].csm$ to 0. Because p_i depends on p_k and p_k has not received the request, p_i should send the request to p_k. However, according to Checkpointing algorithm, the request is sent iff $R_i[k]$ is equal to 1 and $csm_i[k]$ is greater than $Mr[k].csm$. Thus, p_i does not send to the request to p_k because both $csm_i[k]$ and $Mr[k].csm$ are equal to 0.

The above counterexample appears in model checking of any initial states of a system. However, we will show another error that demonstrates the usefulness of our method to generate and model check all possible initial states of a system. We prepare `cp-checkAll` for the first underlying system. We have conducted experiments for six instances of the system that are described by the numbers of mobile support stations (MSSs) and mobile hosts (MHs) as shown in Table. 4.3. The second column presents the number of mobile support stations, the third column presents the number of mobile hosts and the number of all possible initial states is shown in the fourth column[1].

Considering a very simple mobile system with 4 mobile support stations and 4 mobile hosts, there are many possible initial states of the system and some of them are as shown in Fig. 4.11. Let us just concentrate on the two initial state **(a)** and **(b)** and take into account two mobile hosts $mb1$ and $mb4$. When $mb1$ sends a message to $mb4$, there is only one route in **(a)** to pass the message from $mb1$ to $mb4$ through $mss1$, $mss2$, $mss3$ and $mss4$, while there are two routes to pass the message in **(b)** through either $mss1$, $mss2$, and $mss4$, or $mss1$, $mss3$ and, $mss4$. Assuming that $mb1$ has received computation messages from some other components, but not yet from $mb4$. It starts the first checkpointing process, records its local checkpoint and asks all its relative components to take their local checkpoints. After receiving all replies from these components, it

[1]The experiments focus on the working of the algorithm for different wired channels among mobile support stations. mobile hosts are simply assigned to mobile support stations such that at leat one mobile host is connected to a mobile support station and mobile support stations that have more channels are given more priority to have mobile hosts assigned.

terminates the checkpoint process and sends broadcast messages to all components in the system including $mb4$. The messages are passed to $mb4$ by the route $mss1$, $mss2$ and $mss4$. It may happen that while the broadcast message is on the way to $mb4$, $mb4$ sends a computation message to $mb1$ and after that $mb1$ starts the second checkpoint process. It thus sends the checkpoint request to $mb4$. We assume that the request is delivered to $mb4$ with the different routes $mss1$, $mss3$ and $mss4$. When $mb4$ receives the request, it takes its local checkpoint and sends a reply to $mb1$. The broadcast message now arrives at $mb4$. According to the pseudo-code of the algorithm, $mb4$ terminates the checkpoint process by setting $cp\text{-}state_{mb4}$ to 0. This is an error because the broadcast message corresponds to the first checkpoint, but not the current checkpoint. It occurs because there are two different routes to pass a message from $mb1$ to $mb4$. The checkpoint request of the second checkpoint process arrives before the broadcast message. This counterexample is found when we model check the initial state as shown in **(b)**. However, this counterexample does not occur in the initial state **(a)** because there is only one route from $mb1$ to $mb4$. The broadcast message definitely arrives at $mb4$ before the checkpoint request. The error is very simply and can be simply fixed by checking whether the broadcast cast message corresponds to the current checkpoint of a component. When a component receives an broadcast message, it compares its own *trigger* and the message *trigger*. If they are the same, it terminates its checkpoint process, and otherwise it refuses the message. However, the point is that an counterexample may occur in some initial configurations, but may not in the others. Thus, if we only conduct model checking for configurations as in **(a)**, the counterexample cannot be found. It is almost impossible for humans to predict all cases in advance. Fortunately, by generating all possible initial states, our method makes it possible to detect counterexamples lurking in the UMDS-CPs for the property concerned. Said differently, it increases the counterexample detection ratio or the correctness confidence of an algorithm.

Table. 4.3 shows the results of the model checking experiments. Experiments were conducted on a computer that carries a 2.9GHz Intel Xeon E5-4655 processor with 256GB RAM. The fifth column of the table shows the number of initial states from which the counterexample of the first error is found and N/C denotes that the counterexample is not found in any initial state. The next column presents the time taken to find the counterexample. In the same meaning, two last columns present the number of initial states and the time taken for the counterexample of the second error.

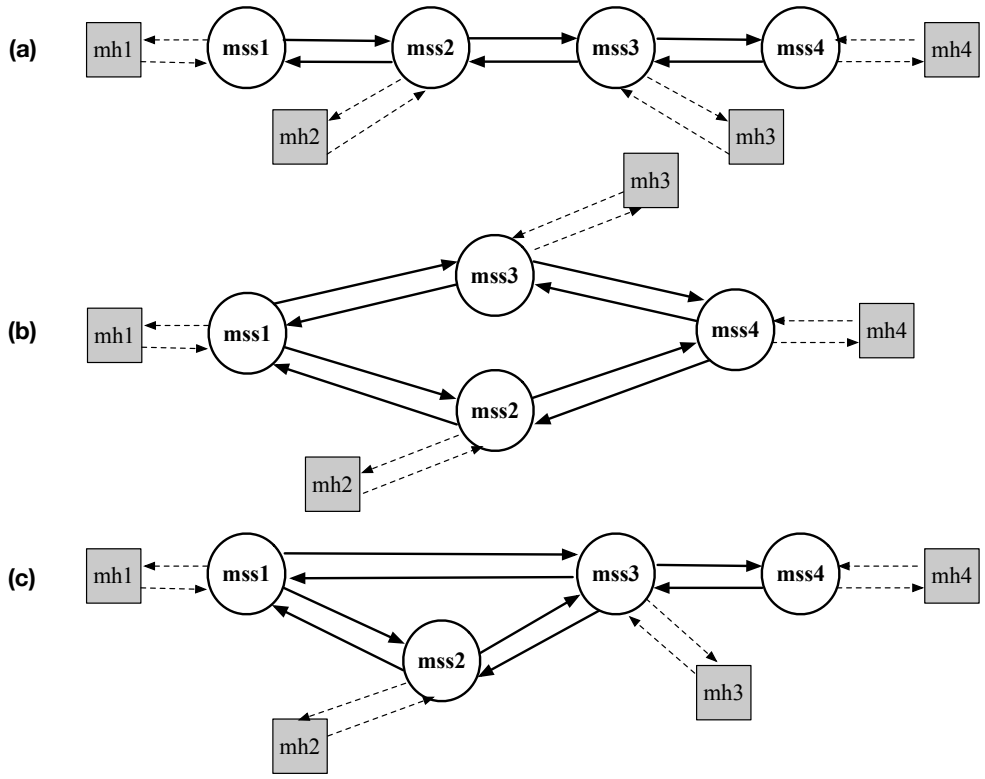


Figure 4.11: Some possible initial states of a mobile distributed system consisting of four mobile support stations and four mobile hosts.

Chapter 5

Formally Analyzing Mobile Robot Algorithms in Maude

This chapter shows how to model, specify and model check mobile robot algorithms in Maude. We restrict our attention to discrete models only, and more specifically to the ring topology. About timing assumption, we consider the more general asynchronous model ASYNC. We have demonstrated in [30, 31] that Maude allows us to specify distributed algorithms/systems more succinctly than others. For instance, it supports *associative* and *commutative* operator attributes that are very necessary to concisely specify distributed algorithms/systems. To describe the model, we use state machines. A distributed mobile robot system is formalized as a state machine and then the state machine is specified in Maude as a module. We then use the Maude LTL model checker to formally verify an algorithm enjoys some desired properties. Namely, we analyze a perpetual exploration algorithm and a gathering algorithm, which have been already described in Section 2.1.2.

5.1 Specification and Model Checking of a Perpetual Exploration Algorithm

We consider the *exclusive perpetual exploration* of the *ring* and analyze one of the first algorithm proposed to solve this problem. More specifically we focus on the algorithm designed for three robots by Blin *et al.* [10]. This section describes our way to formalize the system and specify it in Maude.

5.1.1 System Specification

The system in which robots operate under the control of the algorithm will be modeled and then specified in order to model check the algorithm. The system is specified in our system module EXPLORATION, which defines the behavior of the system. We consider first how to express the states of the system and then the state transitions for the system.

State Expressions

Robots are denoted as r1, r2, ..., and the sort is `Robot`. In the Compute phase, a robot, based on the perceived configuration, makes a decision to stay idle or move to one of its adjacent nodes: either the node on the right or the node on the left. *Pending moves* are denoted as `L`, `R`, and `nil` corresponding respectively to moving to the right, moving to the left, and staying idle, and the sort is `Pending`.

Since the algorithm works on a ring shape network, our modeling of the system respects the ring. Although the ring is an anonymous ring without orientation and the robots on the ring are anonymous, for model checking purpose, we name the robots and number the nodes of the ring. For a ring of size n, nodes are labeled from 0 to $n-1$ following an arbitrary clockwise ordering. Each robot is given a different name. We want to note that this does not affect the fact that the ring and the robots are anonymous since our implementation of the rules for the algorithm recognizes the robots are identical, and likewise with the nodes.

Each robot is located in one node of the ring. When a node is occupied by robots, the node is called a non-empty node. Otherwise, it is called an empty node. We actually represent the ring by the set of all non-empty nodes. Each such node is denoted as

`<r,d,p>`, where `r` is the name of the robot, `d` is the label of the node, and `p` is the pending move of the robot. The corresponding sort is `Node`.

The ring is denoted as a commutative and associative set of these non-empty nodes and the sort is `Ring`. Rings without any robots are called empty rings and we use `empR`, a constant of the sort `Ring`, for them.

```
subsort Node < Ring .
op <_,_,_> : Robot Nat Pending -> Node [ctor] .
op empR : -> Ring [ctor] .
op __ : Ring Ring -> Ring [ctor assoc comm id: empR] .
```

where `Nat` is the sort for natural numbers.

We define the sort `Size` for the size of a ring. This sort is a super-sort of the sort `Nat`. The constant `size`, an element of the sort `Size`, is used for the size of a ring. The configuration of a system is denoted as `{R}`, where `R` is an element of the sort `Ring`, and the sort is `Config`. The system is described by the configuration and the size of the ring.

```
op {_} : Ring -> Config .
op size : -> Size .
```

Some examples of how to describe a system are showed in Figure 5.1.

In any initial configuration of the system, there is no two robots located on the same node and the pending move of any robots is nil.

We define two important concepts: *interval* and *order*. Given two robots r1 and r2 located respectively on the nodes n1 and n2, we define the intervals between the two robots as the number of edges between n1 and n2. Since the environment is a ring, for each pair of robots there are two different intervals, (1) the clockwise interval which counts

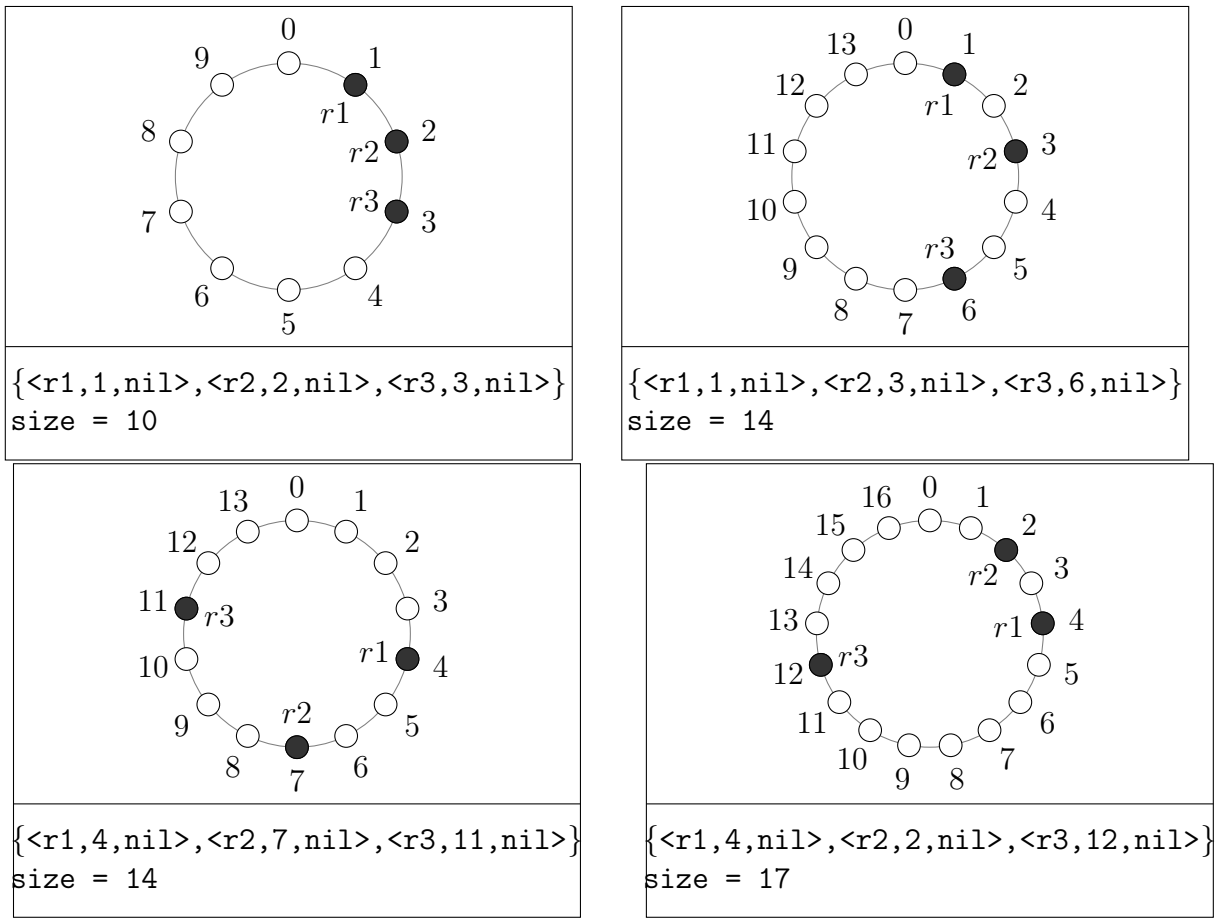


Figure 5.1: Describing a system by the configuration and the size of the ring

edges from n1 to n2 following the clockwise orientation, and (2) the counter-clockwise interval, which counts edges in the other direction.

The latter concept takes into account the order of robots on the ring. Since the algorithm under study works for three robots, we define the notion only for this case. It is possible to extend the notion for arbitrary number of robots. Given three robots r1, r2, and r3 located respectively in the node n1, n2, and n3, the three robots are said to be in the right (or clockwise) order if and only if n2 is between n1 and n3 on the ring according to the clockwise orientation (*e.g.* the three first configurations of Figure 5.1). Conversely, the three robots are said to be in the left (or counter-clockwise) order if and only if n2 is after n1 and before n3 on the ring according to counter-clockwise orientation (*e.g.* the last configuration of Figure 5.1). We calculate the interval of two robots and the order of three robots by using the following functions:

```
op order : Nat Nat Nat -> Pending .
op interval : Nat Nat Pending -> Nat .
```

The function order(N1, N2, N3) returns the order of three robots located in the nodes N1, N2 and N3. The order can be `L` (left order), `R` (right order), or `nil` in case the three

robots are not in any order.[1] Function interval(N2, N1, M) returns the interval of the two robots located on N1 and N2. The third parameter M, which can be `L`, `R`, or `nil`, is used to determine which way we calculate the interval.

State Transitions

All robots execute the same algorithm and they do not have the ability to distinguish themselves from others. The algorithm is given as the set of rules as presenting in Section 2.1.2. A robot first takes a look at the system in the Look phase to capture the snapshot of the system, which contains the positions of all robots on the ring. Then based on the snapshot and following the set of the rules of the algorithm, it decides the next movement in the Compute phase. The pending move will be executed in the Move phase. The state transition for the system is conducted from the rules of the algorithm. The rules of the algorithm will be implemented as rewriting rules in Maude. The following rewriting rules do not identify robots and also nodes. They totally depend on the positions of all robots on the ring. This ensures that the ring and robots are considered anonymous. Since the Compute phase uses the snapshot of the system taken in the Look phase as input and a robot does not perform any movements during two phases, we combine the two phases in the one called the Look-Compute phase in which a robot takes the snapshot of the system and calculates the movement. A robot decides to take a snapshot of the system and calculate a movement only when it dose not hold a pending move. The pending move is stored in the third parameter of the notation of a non-empty node as `<r,d,p>`. We separate the set of rewriting rules for the system into two sets: the set of rules for the Look-Compute phases corresponding to the set of the rules of the algorithm and the set of rules for the Move phase.

The rules for the Look-Compute phase. Each of the following rewriting rule corresponds to one rule in the set of rules of the algorithm. For each rule, we keep the same name as in the original paper [10] to easily match them. In the following part, `R1`, `R2`, and `R3` are variables of the sort `Robot`; `N1`, `N2`, and `N3` are variables of the sort `Nat`; and `M`, `M1`, `M2`, and `M3` are variables of the sort Pending.

1. The rewriting rule corresponding to the rule RL1 given on Figure 2.6:

```
crl [RL1] : { <R1,N1,nil> <R2,N2,M2> <R3,N3,M3> }
            => { <R1,N1,change(M)> <R2,N2,M2> <R3,N3,M3> }
            if M := order(N1,N2,N3) /\ interval(N2,N1,M) == 1
                                    /\ interval(N3,N2,M) == 3 .
```

where the function `change(M)` returns the opposite value of `M`. If `M` is `L` then it returns `R`, if `M` is `R` then it returns `L`, and `nil` otherwise.

The lefthand side and the conditional part of the rule encodes the initial configuration of the rule RL1 (*i.e.* the left picture of Figure 2.6). The initial configuration contains

[1] When two robots are located on the same nodes, there is no order.

three robots such that, given an orientation (clockwise or counterclockwise), the two first robots are at distance 1 (*i.e.* neighbors) and the third robot is at distance 3 from the second one; both distances/intervals being computed in the same orientation.

The topmost robot of Figure 2.6 corresponds to robot `R1` of the Maude rule. According to the rule RL1, this robot has to move if it takes a snapshot of this configuration. In Maude, it is specified by having no pending move initially (the parameter `nil` on the lefthand side), while having a pending move (`change(M)` on the righthand side). The direction of the pending move is chosen to match the direction of the move in rule RL1.

Since the two other robots are not supposed to move in RL1; the parameters `M2` and `M3` respectively are not updated in the conditional rule.

Note that the rule RL1 could have equivalently be written by exchanging the roles of the first and third robot, as proposed below. We choose the previous specification to match as closely as possible the rules given in the original paper.

```
crl [RL1] : { <R1,N1,M1> <R2,N2,M2> <R3,N3,nil> }
            => { <R1,N1,M1> <R2,N2,M2> <R3,N3,M> }
            if M := order(N1,N2,N3) and interval(N2,N1,M) == 3
                                    and interval(N3,N2,M) == 1 .
```

2. The rewriting rule corresponding to the rule RL2 given on Figure 2.7.

```
crl [RL2] : { <R1,N1,M1> <R2,N2,nil> <R3,N3,M3> }
            => { <R1,N1,M1> <R2,N2,change(M)> <R3,N3,M3> }
            if M := order(N1,N2,N3) /\ interval(N2,N1,M) == 2
                                    /\ interval(N3,N2,M) == 3 .
```

3. The rewriting rule corresponding to the rule RL3 given on Figure 2.8.

```
crl [RL3] : { <R1,N1,M1> <R2,N2,M2> <R3,N3,nil> }
            => { <R1,N1,M1> <R2,N2,M2> <R3,N3,change(M)> }
            if M := order(N1,N2,N3) /\ interval(N2,N1,M) == 1
                                    /\ interval(N3,N2,M) == 4 .
```

4. The rewriting rule corresponding to the rule RC1.

```
crl [RC1] : { <R1,N1,M1> <R2,N2,M2> <R3,N3,nil> }
            => { <R1,N1,M1> <R2,N2,M2> <R3,N3,change(M)> }
            if M := order(N1,N2,N3) /\ interval(N2,N1,M) == 1
                                    /\ interval(N3,N2,M)> 4
               /\ (interval(N3,N2,M) < interval(N1,N3,M)) .
```

5. The rewriting rule corresponding to the rule RC2.

```
crl [RC2] : { <R1,N1,M1> <R2,N2,M2> <R3,N3,nil> C }
            => { <R1,N1,M1> <R2,N2,M2> <R3,N3,M> C }
            if M := order(N1,N2,N3) /\ (interval(N2,N1,M)> 0)
                                    /\ (interval(N3,N2,M)> 1)
               /\ (interval(N3,N2,M) == interval(N1,N3,M)) .
```

6. The rewriting rule corresponding to the rule RC3.

```
crl [RC3] : { <R1,N1,M1> <R2,N2,nil> <R3,N3,M3> C }
            => { <R1,N1,M1> <R2,N2,change(M)> <R3,N3,M3> C }
            if M := order(N1,N2,N3)
               /\ (interval(N3,N2,M) > interval(N2,N1,M))
               /\ (interval(N1,N3,M) > interval(N3,N2,M))
               /\ (interval(N2,N1,M) > 1) .
```

7. The rewriting rule corresponding to the rule RC4.

```
crl [RC4] : { <R1,N1,nil> <R2,N2,M2> <R3,N3,M3> C }
            => { <R1,N1,change(M)> <R2,N2,M2> <R3,N3,M3> C }
            if M := order(N1,N2,N3)
               /\ (interval(N2,N1,M) == 1)
               /\ (interval(N3,N2,M) == 1) .
```

This rule is more subtle than other rules and the formalization from the original rule is less straightforward. The initial configuration (lefthand side of the rewriting rule) is symmetrical; three robots are adjacent to each other. The commutativity of the sort `Ring` and our notions of order and interval guarantee that this specification is conform to the original rule; it is possible that either one or two robots compute a move.

8. The rewriting rule corresponding to the rule RC5.

```
crl [RC5] : { <R1,N1,M1> <R2,N2,M2> <R3,N3,nil> C }
            => { <R1,N1,M1> <R2,N2,M2> <R3,N3,M> C }
            if M := order(N1,N2,N3)
               /\ (interval(N2,N1,M) == 1)
               /\ (interval(N3,N2,M) == 2) .
```

The rules for the Move phase. Each robot may move to its adjacent node on the left, its adjacent node on the right or stay idle. This movement is based on its pending move. In the following, the function `moveL(N)` is to move the robot located on the node N to the adjacent node on the left and the function `moveR(N)` is to move the robot located on the node N to the adjacent node on the right .

1. If the stored pending move is L, the robot will move to the adjacent node on the left.

```
rl [RL-Lpending] : { <R1,N1,L> C } => { <R1,moveL(N1),nil> C } .
```

2. If the stored pending move is R, the robot will move to the adjacent node on the right.

```
rl [RL-Rpending] : { <R1,N1,R> C } => { <R1,moveR(N1),nil> C } .
```

5.1.2 Model Checking

In order to verify the correctness of an algorithm, two properties have to be model-checked:

- The perpetual exploration property, and
- The mutual exclusion property.

where the former is a liveness property, while the latter is a safety property.

The perpetual exploration property guarantees that each robot visits infinitely often each node. It is a liveness property which ensures that something good eventually happens. The mutual exclusion property ensures that no two robots are located on any node at any given time. This property is a safety property, which guarantees that something bad never happens. Maude is equipped with an LTL model checker [17, 38]. These two properties can be expressed in the LTL used by Maude.

State Predicates

We define the state predicates `perexp` and `mutual` which are used to specify the two properties as LTL formulas. The two predicates are specified in the module `EXPLORATION-PREDS`, which protects the module `EXPLORATION` and includes the module `SATISFACTION`. The sort `Config` is chosen as our kind for states and declared as sub-sort of the sort `State`.

```
mod EXPLORATION-PREDS is
   pr EXPLORATION .
   inc SATISFACTION .
   subsort Config < State .
   ...
 endm
```

where '`...`' indicates the part in which the syntax and semantics of the state predicates are specified. The specification of predicate `perexp(R, N)` is as follows:

```
op perexp : Robot Nat -> Prop .
eq { < R, N, M > C } |= perexp(R, N) = true .
eq { C }             |= perexp(R, N) = false [owise] .
```

where R is a variable of the sort Robot, N a variable of the sort Nat, and M a variable of the sort Pending. The predicate perexp(R, N) is true in the configuration S if and only if the robot R is located in the node N in S, as <R,N,M>, and false otherwise.

The mutual predicate is specified as follows:

```
op mutual :  -> Prop .
op checkMutual : Config -> Bool .
op checkMutual1 : Robot Nat Config -> Bool .

eq { C } |= mutual = checkMutual({ C }) .
eq checkMutual({ empR }) = false .
eq checkMutual({ < R1, N1, M1 > C }) =
   checkMutual1(R1, N1, { C }) or checkMutual({ C }) .

eq checkMutual1(R1, N1, { empR }) = false .
eq checkMutual1(R1, N1, { < R2 , N2, M2 > C }) =
    (N1 == N2) or checkMutual1(R1, N1, { C }) .
```

where R1 and R2 are variables of the sort Robot, N1 and N2 are variables of the sort Nat, M1 and M2 are variables of the sort Pending, and C is a variable of the sort Ring.

The value of the predicate mutual depends on the result of the function checkMutual which is false if and only if there are no two robots located in the same node and true otherwise.

Property Specifications as LTL Formulas

The perpetual exploration property and the mutual exclusion property will be specified as LTL formulas. The LTL formula saying that the robot r eventually visits the node 0, eventually visits the node 1, ..., and eventually visits the node n-1 is as follows:

```
([]<> (perexp(r, 0))) /\ ([]<> (perexp(r, 1)))
   /\ ([]<> (perexp(r, 2)))   /\  ...
   /\ ([]<> (perexp(r, n-2))) /\ ([]<> (perexp(r, n-1))) .
```

where n is the size of the ring.

We define a function named perexpGen as the following function to automatically generate this formula:

```
op perexpGen : Robot Nat -> Formula .
eq perexpGen(R, 0)  = ([]<> (perexp(R, 0))) .
ceq perexpGen(R, N) = ([]<> (perexp(R, N)))
                        /\ perexpGen(R, sd(N, 1)) if N > 0 .
```

where R and N are variables of the sort Robot and Nat respectively.

The perpetual exploration property is satisfied if and only if the LTL formula perexpGen(r,n) is satisfied for all robots r in the system of a n-node ring:

```
perexpGen(r1,n) and perexpGen(r2,n) and ...  and perexpGen(rk,n) .
```

where k is the number of robots in the system.

The mutual exclusion property is expressed as the following LTL formula:

```
[]~ (mutual)
```

This formula says that mutual predicate is always false, meaning that the mutual exclusion will never happen.

5.1.3 Experiments and Counterexample

The module EXPLORATION specifying the system in which robots execute the algorithm has been given. In the module EXPLORATION-PREDS, which protects the module EXPLORATION, the two predicates and their semantics have been defined. The two properties have been specified as LTL formulas. All requirements to perform the model checking are satisfied. We define a new module, called EXPLORATION-CHECK. The module EXPLORATION-CHECK imports the module MODEL-CHECKER, which supports LTL model checking. We then model check the two given LTL formulas specifying the two properties for a given initial configuration. An initial configuration is defined as the constant initial of the sort Config in the module EXPLORATION-CHECK. First, we perform the model checking for the ring with size 10.

```
op initial : -> Config .
ops r1 r2 r3 : -> Robot .
eq initial = { < r1, 1, nil > < r2, 2, nil > < r3, 3, nil > } .
 eq size = 10 .
```

We are now ready to model check the two properties. We use the key operator modelCheck, which takes a state and the LTL formula and returns either the Boolean true if the formula is satisfied, or a counterexample when it is not satisfied. The first property to check is perpetual exploration:

```
red modelCheck(initial, perexpGen(r1, 9)) .
red modelCheck(initial, perexpGen(r2, 9)) .
red modelCheck(initial, perexpGen(r3, 9)) .
```

The second property to check is mutual exclusion:

```
red modelCheck(initial, []~ (mutual)) .
```

For the ring with of 10, we can define all possible initial configurations of the system. It takes less than 30 seconds to model check both properties for all initial configurations.

For the ring of size 10, there is one initial configuration for which the model checker finds a counter example for each formula; this is the initial configuration with three adjacent robots. This means that all the formulas specifying the two properties are not satisfied. The counter example is shown as follows:

```
reduce in M-CHECK : modelCheck(initial, []~ mutual) .
rewrites: 284597 in 89ms cpu (91ms real) (3163946 rewrites/second)
result ModelCheckResult: counterexample(...)

reduce in M-CHECK : modelCheck(initial, perexpGen(r1, 9)) .
rewrites: 284597 in 89ms cpu (91ms real) (3163946 rewrites/second)
result ModelCheckResult: counterexample(...)

reduce in M-CHECK : modelCheck(initial, perexpGen(r2, 9)) .
rewrites: 284597 in 89ms cpu (91ms real) (3163946 rewrites/second)
result ModelCheckResult: counterexample(...)

reduce in M-CHECK : modelCheck(initial, perexpGen(r3, 9)) .
rewrites: 284597 in 89ms cpu (91ms real) (3163946 rewrites/second)
result ModelCheckResult: counterexample(...)
```

where '...' is the following counter-example:

```
{{< r1,1,nil > < r2,2,nil > < r3,3,nil>},'RC4}
{{< r1,1,L > < r2,2,nil > < r3,3,nil >},'RC4}
{{< r1,1,L > < r2,2,nil > < r3,3,R >},'RL-Lpending}
{{< r1,0,nil > < r2,2,nil > < r3,3,R >},'RC5}
{{< r1,0,L > < r2,2,nil > < r3,3,R >},'RL-Rpending}
{{< r1,0,L > <r2,2,nil > < r3,4,nil >},'RC2}
{{< r1,0,L > < r2,2,R > < r3,4,nil >},'RL-Rpending}
{{< r1,0,L > < r2,3,nil > < r3,4,nil >},'RL1}
{{< r1,0,L > <r2,3,nil > < r3,4,R >},'RL-Rpending}
{{< r1,0,L > < r2,3,nil > < r3,5,nil>},'RL2}
{{< r1,0,L > < r2,3,R > < r3,5,nil >},'RL-Lpending}
{{< r1,9,nil > < r2,3,R > < r3,5,nil >},'RC2}
{{< r1,9,L > < r2,3,R > < r3,5,nil >},'RL-Lpending}
{{< r1,8,nil > < r2,3,R > < r3,5,nil >},'RL2}
{{< r1,8,nil > < r2,3,R > < r3,5,L >},'RL-Lpending}
{{< r1,8,nil > < r2,3,R > < r3,4,nil >},'RL-Rpending}
{{< r1,8,nil > < r2,4,nil > < r3,4, nil >},deadlock}
```

The scenario of the counter example is depicted in Figure 5.2. Looking at the counter example, we can recognize that a collision situation occurs, in which there are two robots located in the same node at the same time. This shows that the mutual exclusion property is not satisfied.

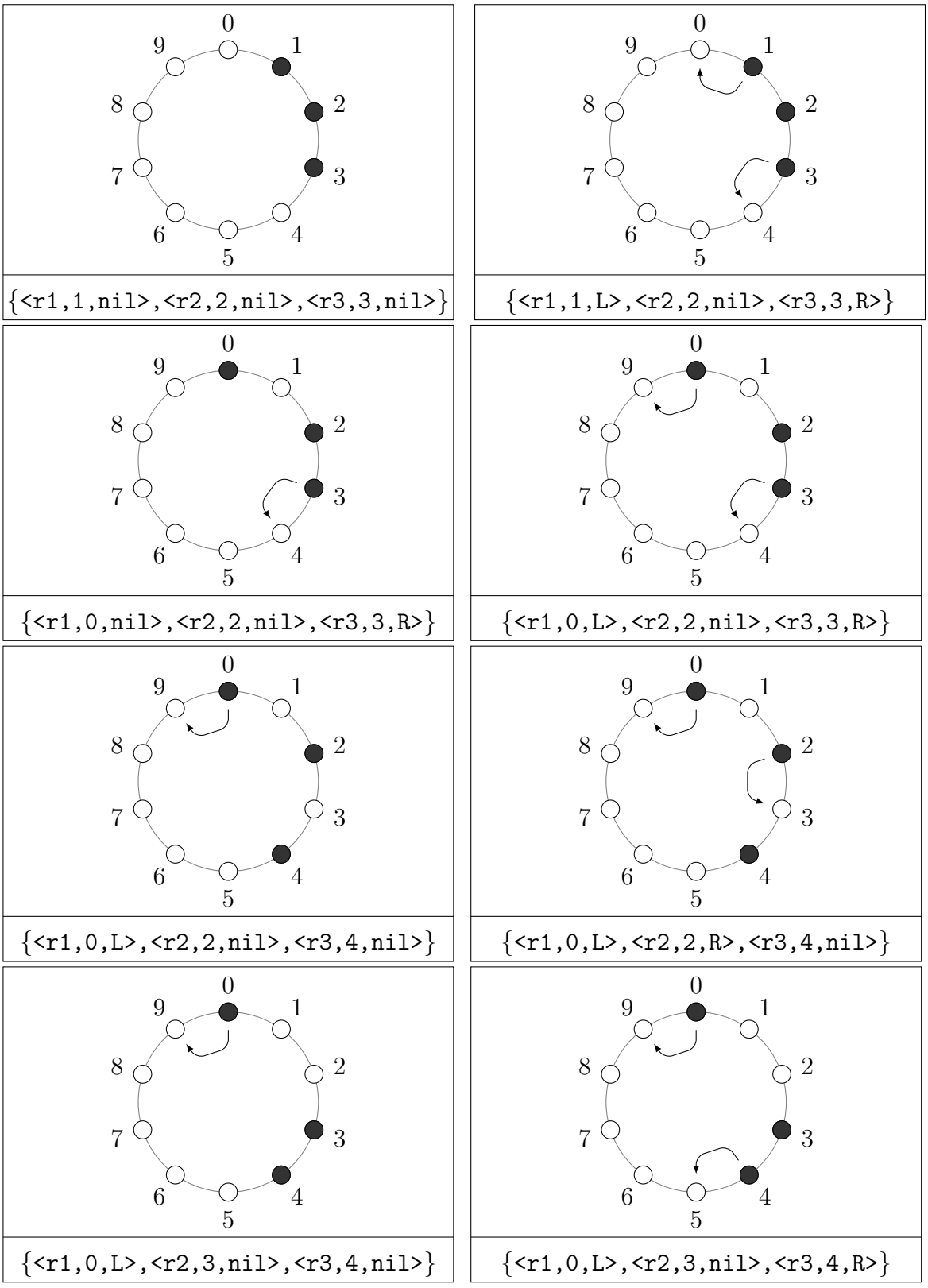
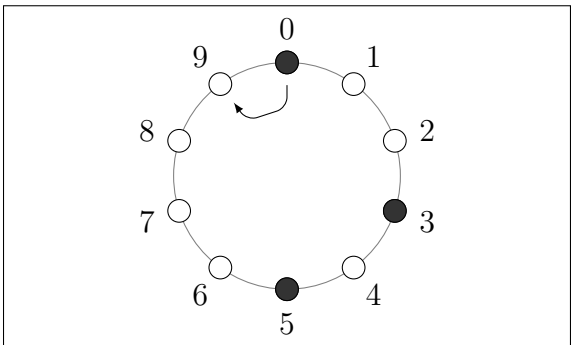
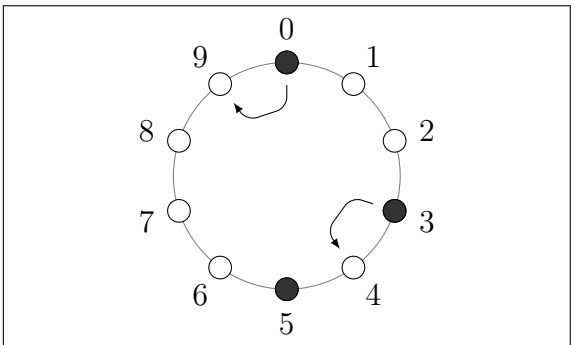


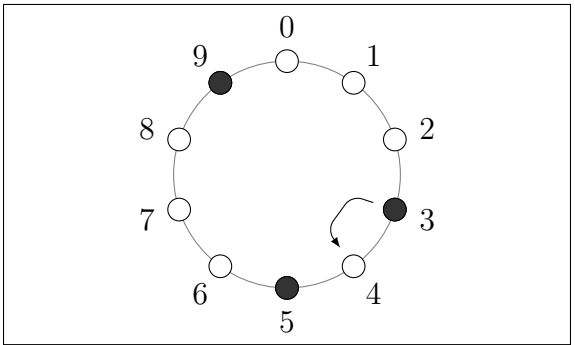
Figure 5.2: Scenario of the counter example



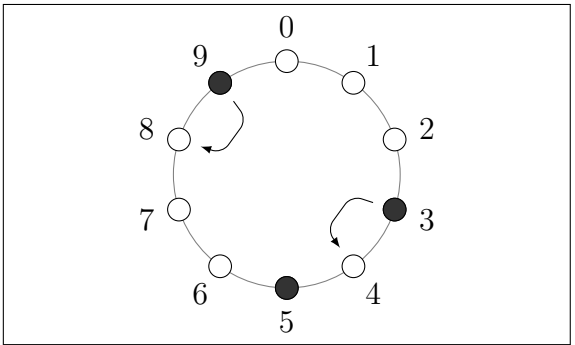
{<r1,0,L>,<r2,3,nil>,<r3,5,nil>}



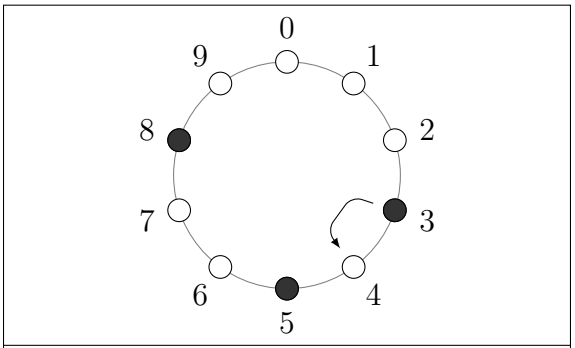
{<r1,0,L>,<r2,3,R>,<r3,5,nil>}



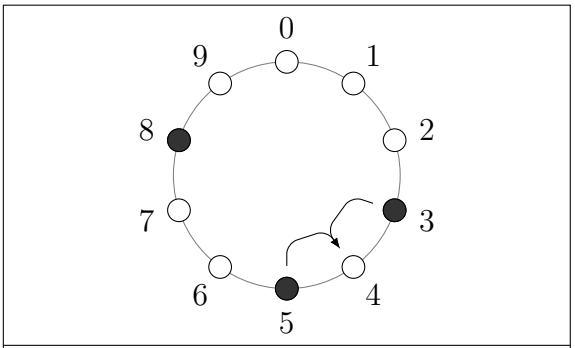
{<r1,9,nil>,<r2,3,R>,<r3,5,nil>}



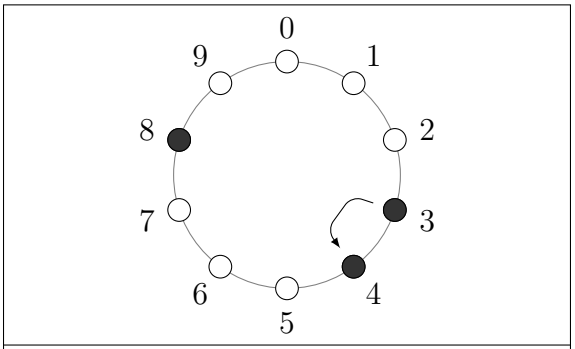
{<r1,9,L>,<r2,3,R>,<r3,5,nil>}



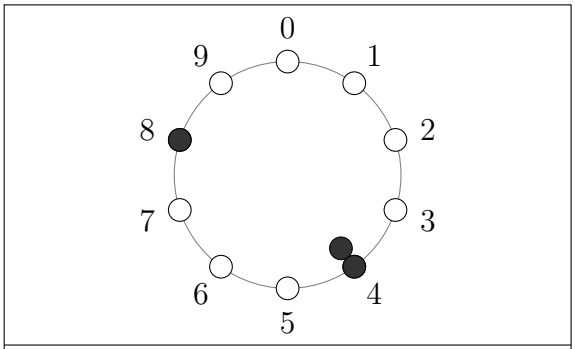
{<r1,8,nil>,<r2,3,R>,<r3,5,nil>}



{<r1,8,nil>,<r2,3,R>,<r3,5,L>}



{<r1,8,nil>,<r2,3,R>,<r3,4,nil>}



{<r1,8,nil>,<r2,4,nil>,<r3,4,nil>}

Figure 5.2: Scenario of the counter example (continued)

The perpetual exploration property is also not satisfied since the two robots collide. No robot, after that, can move anymore.

The counter example states that the two properties are not satisfied, and conclusively, the algorithm does not work correctly. While being outside the scope of this paper, it is worth mentioning that the algorithm can be fixed by changing the rule RC5. Unfortunately, it is not possible to deduce the required modification directly from the counter example.

Although the counter-example has been found, we still conduct some more experiments for other sizes of rings. In our experiments, it takes less than 5 minutes to model check the first property and less than 30 second to model check the second property for rings of size up to 20. Computations were executed on a 4GHz Intel Core i7 processor with 32GB of RAM.

We have described how to specify and model check a mobile robot algorithm in Maude. The model checker found counterexamples showing that the analyzed algorithm is not correct since it does not satisfy the two properties.

5.2 Model Checking of Robot Gathering

We propose a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions. Robots are assumed to have the weak multiplicity detection capacity. We then use the Maude LTL model checker to formally verify an algorithm for robot gathering problem on ring enjoys desired properties. We focus on the gathering problem and analyze the algorithm proposed by D'Angelo *et al.* [21] as a case study. As the result of the model checking, counterexamples have been found. We detect the sources of unforeseen design errors. We, furthermore, give our explanations on these errors. Unavoidably, multiplicity and asynchrony make arduous to formalize the systems. We pay much attention to this problem and solve it in our model.

5.2.1 Formal Model

For these systems, a state of the system is called a configuration. A configuration is described in terms of a view starting from any robot and traversing the ring in one arbitrary direction. When a robot wakes up, it takes the snapshot of the current configuration of the system, and computes a move (called a computed move) based on this snapshot. The computed move is either staying idle or moving to one of its adjacent nodes. In the latter case, it moves to the adjacent node, eventually. In the following part, Maude notation is used to describe state machines. We consider how to express a state and how to describe an event as a state transition.

State Expressions

We denote a robot as a pair $\langle I, P \rangle$, where I denotes the size of the interval, an interval is a maximal set of empty consecutive nodes, between it and the next robot, and P denotes

the computed move. The value of P could be *nil*, *fc* or *fc-* (*fc* stands for *f*ollowing the *c*onfiguration). *nil* means that the robot has no pending move (*i.e* last computed move was idle). *fc* (resp. *fc-*) means that the robot has a pending move to the adjacent node located after it (resp. before it) following the direction of the configuration. Initially, the computed move of each robot is *nil*. If P is *nil*, the robot may be activated and will compute a new move and update P accordingly. If P is not *nil*, the robot may be activated and will execute the move and update P to *nil* after the move. We use the sort `Pending` to denote computed moves and the sort `Pair` to denote pairs. They are expressed by the following operators that are constructors as specified with `ctor`.

```
op <_, _> : Int Pending -> Pair [ctor] .
```

The sort `Int` is used for denoting integers. The operator `<_, _>` is used to construct `Pair`. For $c_1 \in$ `Int`, $c_2 \in$ `Pending`, `<` c_1, c_2 `>` $\in$ `Pair`.

A configuration is expressed as a sequence of pairs. It contains the information about the locations of all robots and their states. The corresponding sort is `Config`. Configurations are defined by the following operators.

```
subsort Pair < Seq .
op empS : -> Seq [ctor] .
op _ _ : Seq Seq -> Seq [ctor assoc id: empS] .
op {_} : Seq -> Config [ctor].
```

where the sort `Seq` is used for sequences of pairs. `empS` denotes the empty sequence of pairs. `Seq` is a supersort of `Pair`, which means that each `Pair` is treated as the singleton sequence only consisting of the pair. The juxtaposition operator `_ _` is used to construct non-trivial sequences of pairs. For c_1, $c_2 \in$ `Seq`, c_1 $c_2 \in$ `Seq`. The juxtaposition operator `_ _` is associative as specified with `assoc`, and `empS` is an identity of the operator specified with `id: empS`.

A configuration is of the form $\{\_\}$ of a sequence. States of the system are expressed as terms of the sort `Config`. A term of a sort S is a variable of S or $f(t_1, \ldots, t_n)$ if f is an operator declared as $f : S_1 \ldots S_n \rightarrow S$ $(n \geq 0)$ and $t_1, \ldots, t_n$ are terms of $S_1, \ldots, S_n$. If f has any underscores $\_$, such as $\langle \_, \_ \rangle$, then a different notation than $f(t_1, \ldots, t_n)$ is used, such as $\langle I, P \rangle$ that is a term of the sort `Pair`, where I is term of the sort `Int` and P is a term of the sort `Pending`. Constructor terms are those consisting of constructors and variables. Ground term are those having no variables. Ground constructor terms hence are those composed of constructors only and no variables . Ground constructor terms of the sort `Config` express concrete states of the system. For example, the initial configuration of the system as shown in Fig. 5.3(a) could be expressed as the view starting from the robot r in clockwise order, $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. Let us assume that robot r_1 is activated, takes a look at the configuration, and computes a move. Assuming that the move is to move to the node located after it, the system reaches the configuration as shown in Fig. 5.3(b). It is expressed as $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. If the robot r_1 is activated again, it executes the move; the system becomes as shown Fig. 5.3(c)

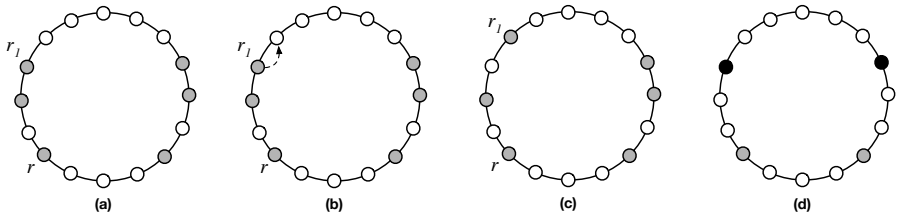


Figure 5.3: Some configurations. A dashed arrow represents a pending move. Black nodes represent multiplicities.

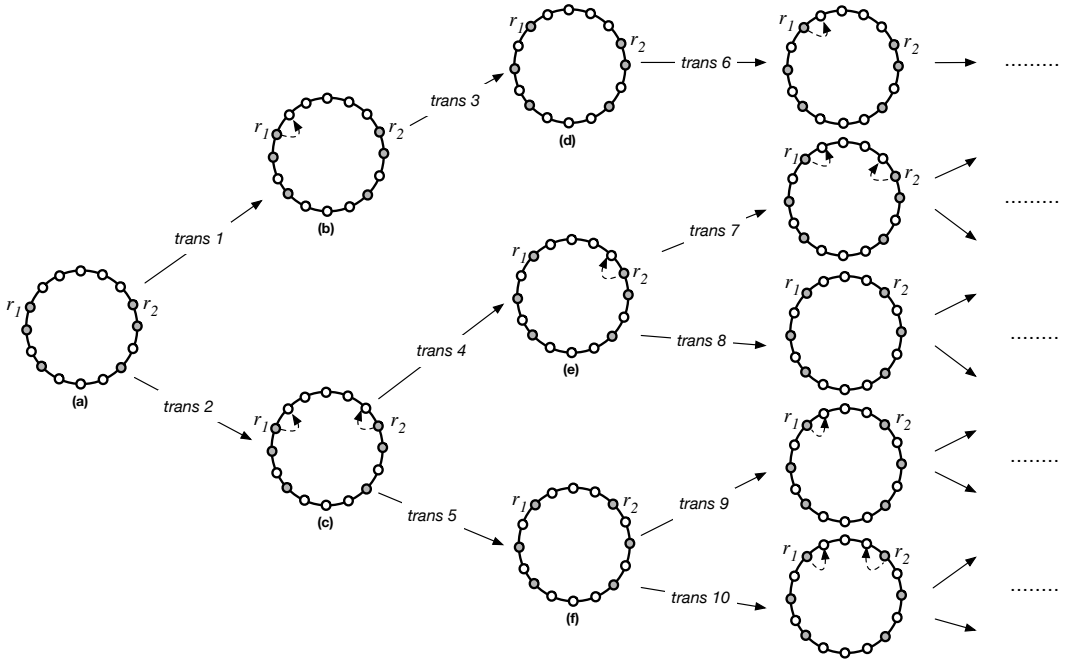


Figure 5.4: A transition graph of one specific initial configuration (a)

which is expressed as $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. A robot is not allowed to look at the second element of the pairs of other robots and it calculates a move based on its own view of the system. For example, the view of robot r_1 in Fig. 5.3(a) could be either $\{\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\}$ or $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\}$ without knowing anything about clockwise order. It is worth noting that this allows us to guarantee that robots have no sense of direction and do not know the pending moves of other robots.

Due to the multiplicity assumption, it is possible that a robot moves to a node that is occupied by other robots. The node is, or becomes a multiplicity. There may be more than one of such robots in one multiplicity. Since the robots are anonymous, we can denote all of them by a pair $\langle I, P\rangle$ in which the value of I is set to the negative of the additional number of robots located on the multiplicity (-3 indicates 3 additional robots, which means a multiplicity of 4 robots). Note that this notation allows us to represent the exact number of robots in multiplicities. But robots do not have access to

this information; they can only know if there is a multiplicity (*i.e.* a negative number). Our encoding allows a simple conversion to consider global strong multiplicity detection. For instance, the configuration as shown in Fig. 5.3(d) assuming that there are two robots in each multiplicity, is expressed as $\{\langle 2, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 2, nil\rangle\langle 3, nil\rangle\}$. We use this encoding to match as closely as possible the definitions introduced in [21].

State Transitions

Because the *Compute* phase uses the snapshot of the system taken in the *Look* phase as input and a robot does not perform any movements during two phases, to model the system, we combine the two phases into one called the *Look-Compute* phase in which a robot takes the snapshot of the system and computes a move. When either (1) a robot takes the snapshot of the system and then computes a move, or (2) a robot executes its pending move, the current configuration of the system changes to another. Such changes are called a state transition (or a transition). A transition is expressed as a pair (l, r), where l and r are configurations.

Let us examine the following scenario. Given an initial configuration as shown in Fig. 5.4(a) and assumed that both robots r_1 and r_2 are allowed to move, it may happen that only one robot (assuming r_1) looks at the system and computes a move, or both r_1 and r_2 do. In the former case, the configuration of the system is transferred to the one as shown in Fig. 5.4(b). The transition is named $trans1$ and expressed by the pair $(\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle$, $\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle)$. In the latter case, the configuration of Fig. 5.4(c) is established. The configuration of Fig. 5.4(d) is obtained after r_1 in the configuration of Fig. 5.4(b) executes its pending move. The configurations of Fig. 5.4(e) and Fig. 5.4(f) are obtained from the configuration of Fig. 5.4(c). The graph in Fig. 5.4 shows possible transitions from the initial configuration. A sequence of transitions starting from an initial configuration, e.g $trans1$, $trans3$, $trans6$, ..., is called a possible execution. There may exist more than one execution from a given initial configuration.

We describe the actions of robots as transition rules. A transition rule is described in the form of a rewrite rule. Each rewrite rule is defined only over `Config` that does not have any sub-sorts and in the form L ⇒ R such that L only consists of constructors and variables. We give here a simple example to explain how an action can be expressed as a transition rule. The following transition rule describes the action corresponding to a robot executing its pending move when there is no multiplicity in the system.

```
crl [fc-pending] : {S1 < I1, P >  < I2, fc >  S2}  =>
{S1 < I1 + 1, P > <  I2 - 1, nil > S2}
if nonMul({S1 < I1, P > < I2, fc > S2}).
```

where `S1`, `S2` ∈ `Seq`, `I1`, `I2` ∈ `Int` and `P` ∈ `Pending` are variables of those sorts; The function `nonMul` returns `true` when the configuration has no multiplicity and `false` otherwise.

The above rule is a conditional writing rule and the condition is specified in the `if`

part. The rule then will be applied if the condition is satisfied. The configuration {S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2} expresses any state such that the robot ⟨ I2, fc ⟩ holds a pending move fc and the robot before it is ⟨ I1, P ⟩. Such a state may have some more robots before and after the two robots that are expressed as $S1$ and $S2$, respectively. The ground constructor term $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expresses the state as shown in Fig. 5.4(b). There is no multiplicity in this configuration. The left-hand side of the above rewrite rule *fc-pending* matches this ground term by substituting *S1, I1, P, I2* and *S2* with $\langle 1, nil\rangle$, 0, nil, 5 and $\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle$, and the rewrite rule can be applied to the term, changing it to $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expressing the state as shown in Fig. 5.4(d). In this way, a rewrite rule expresses a set of state transitions.

To make T, a set of transitions (s, s'), from rewrite rules L ⇒ R, let σ be a substitution from the variables in L to appropriate constructor terms, $\sigma(L)$ is a constructor term, but $\sigma(R)$ is not necessarily, and then it is necessary to reduce $\sigma(R)$ with equations. Let $nf(t)$ be the term obtained by reducing t with equations. So, $(\sigma(L), nf(\sigma(R)))$ is a state transition obtained from L ⇒ R. Let $\sigma(L) \Rightarrow nf(\sigma(R))$ be called a ground instance of L ⇒ R.

Definition 5.2.1 (TR_{RS})**.** Let TR_{RS} be the set of all ground instances of the all transition rules.

Formal Model

We formalize a mobile robot system as a state machine. The state machine includes the set of all possible states as the set of all ground constructor terms S_{RS}, the set of initial states I_{RS} and the binary relation over states T_{RS}. I_{RS} is a subset of S_{RS} such that for each state $s \in S_{RS}$, there is no multiplicity and the configuration does not belong to $NG \cup SP4$. T_{RS} is the binary relation over S_{RS} made from TR_{RS}.

Definition 5.2.2 (M_{RS})**.** The state machine formalizing a mobile robot system is M_{RS}, where

1. S_{RS} is the set of all ground constructor terms whose sorts are *Config*;
2. I_{RS} is a subset of S_{RS} such that $(\forall s \in I_{RS})$ (numMul(s) = 0) and (not ng&sp4(s));
3. T_{RS} is the binary relation over S_{RS} defined as follows:
   {(l, r) | l ⇒ r ∈ TR_{RS}}.

The function numMul counts the number of the multiplicities in the system and the function ng&sp4 returns *true* when a configuration is in $NG \cup SP4$ and *false* otherwise.

We specify this formal model in Maude specification languages. Note that our specification is coherent [17].

5.2.2 Model Checking

Model checking is a verification technique that explores all possible system states and checks whether a desired property that should be satisfied by an algorithm is satisfied. The desired property is required to be formally expressed. A model checker then verifies whether the formula is satisfied for all possible executions. If the formula is not satisfied, a counterexample is found. We specify the formal model of the system in Maude. We then apply Maude LTL model checker to formally verify the algorithm. The original paper [21] has given very important lemmas, such as *Lemma 5, 6, and 7*, that state properties that need to be satisfied at the end of each phase. These lemmas are used to model check the algorithm. We have formally expressed these lemmas as LTL formulas [38]. We give here the formalization of *Lemma 6* as an example. *Lemma 6* states a property that must be satisfied at the end of the COLLECT phase. Namely, it states that the configuration obtained at the end of the COLLECT phase contains two multiplicities and satisfies a condition (called located condition) that the configuration needs to be in some specific configurations in which robots are located in some specific locations. Note that the COLLECT phase can start only if the initial configuration is symmetric or at one step from specific symmetric configurations. To model check this lemma, we expressed it as an LTL formula. We define the atomic propositions `endOfColl` and `coll` as follows:

```
C |= endOfColl = checkOfColl(C) .
C |=  coll = checkColl(C) .
checkColl(C) = checkAllowedSym(C) and (numMul(C) == 2) and
checkCondition(C) .
```

The function `checkOfColl` checks whether a system state C is at the end of the COLLECT phase. The atomic proposition `endOfColl`, thus, is true if and only if the COLLECT phase has just finished. The function `checkAllowedSym` checks whether a configuration is an allowed symmetric configuration. The function `checkCondition` returns `true` when the configuration satisfies the located condition and `false` otherwise. The atomic proposition `coll`, thus, is `true` when the configuration is an allowed symmetric configuration containing two multiplicities and satisfies the located condition and `false` otherwise. .
The mathematical notation `==` stands for equivalence
The lemma then is formally expressed as an LTL formula as follows.

```
lemma6 =  [] (endOfColl -> coll)  /\  <> endOfColl .
```

where `[]` stands for always (globally) and `<>` stands for eventually (in the future).

Intuitively, the formula states that it is always true that the phase COLLECT will finally terminate and whenever the phase has been just over, then `coll` is true. This means that the `Lemma 6` is satisfied at the end of the COLLECT phase.
This formula is used to conduct the model checking for the algorithm. To model check, we separate the formula into two sub-formulas and model check them separately in order to easily detect the source of errors (if some are found). Namely, we use the two following formulas:

```
lemma6-1 = <> endOfColl .
lemma6-2 = [] (endOfColl -> coll)  .
```

5.2.3 Experiments and Counterexamples

As the result of the model checking, counterexamples are found. A counterexample is of the form of a possible execution: it includes all visited states and the sequence of transition rules applied. This helps to analyze counterexamples to detect the source of the detected errors. We present two counterexamples as follows:

The first one results from the model checking of the formula lemma 6-1.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-1) .
rewrites: 89524096 in 40088ms cpu (40173ms real) (2233163 rewrites/second)
result ModelCheckResult: counterexample(
     {{< 0,nil > < 1,nil > < 0,nil > < 3,nil>
       < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
     {{< 0,nil > < 1,nil > < 0,nil > < 3,nil > < 0,fc >
       < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
     {{< 0,nil > < 1,nil > < 0,nil > < 4,nil > < -1,nil >
       < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo2}
     {{< 0,nil > < 1,nil > < 0,nil > < 4,fc- > < -1,nil >
       < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
     {{< 0,nil > < 1,nil > < -1,nil > < 5,nil > < -1,nil >
       < 1,nil > < 0,nil > < 0,nil >},deadlock})
```

The counterexample shows that the system falls into a deadlock state. Indeed, the configuration $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 0, nil\rangle\}$ belongs to the kind of configurations that need to be handled by the COLLECT phase. However, the COLLECT phase could not deal with it. Thus, the system is not able to reach a valid state at the end of the COLLECT phase. Analyzing this counterexample, we detect the error, which is described later in Section 4.1. This also means that the algorithm fails in gathering all robots in the same location.

The second counter-example results from the model checking of the formula lemma 6-2.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-2) .
rewrites: 15982060 in 7064ms cpu (7114ms real) (2262435 rewrites/second)
result ModelCheckResult: counterexample(
   {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
     < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
   {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
     < 0,fc > < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
   {{< 0,nil > < 1,nil > < 0,nil > < 4,nil >
     < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo1}
   {{< 0,nil > < 1,nil > < 0,fc- > < 4,nil >
```

```
  < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
{{< 0,nil > < 0,nil > < 1,nil > < 4,nil >
  < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'ofColl})
```

This counterexample occurs because the state $\{\langle 0, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle \langle 0, nil\rangle\langle 0, nil\rangle\}$ is at the end of the COLLECT phase, but the atomic propositions *coll* returns *false*. Specifically, at end of the COLLECT phase, the configuration does not contain two multiplicities and satisfy the located condition.

Since counterexamples are found, the lemma does not hold. The remainder of this section reports on some errors that we have found. In addition, we also give our opinions about the origins of these errors.

Omission of Special Cases

We report here the error that corresponds to the first counterexample. The error occurs when the algorithm deals with symmetric configurations with two multiplicities. Single robots in such configurations should move such that the configuration eventually reaches a symmetric one with (i) size nodes occupied, (ii) two multiplicities, (iii) two robots adjacent to these multiplicities, and (iv) other robots in specific locations. However, it is not straightforward to design a strategy for these robots. Two examples are shown in Fig. 5.5. For the configuration of Fig. 5.5(a), the two symmetric robots are expected to move such that either the configuration of Fig. 5.5(b) or Fig. 5.5(c) is obtained. In the same way, the configurations of Fig. 5.5(e) and Fig. 5.5(f) are obtained from the configuration of Fig. 5.5(d).

Unfortunately, a counterexample is found. Our model checking detects that the algorithm does not work correctly for the configuration of Fig. 5.5(d). The two single robots do not move as expected. The same configuration is obtained instead of the configurations of Fig. 5.5(e), 5.5(f). Indeed, the algorithm only works correctly for configurations in which there are two single robot adjacent to the two multiplicities, *e.g.* the configurations similar to the one of Fig. 5.5(a), but it does not work for configurations such as the one of Fig. 5.5(d).

This error occurs because some cases are missing in the algorithm. For instance, the configurations as the configuration of Fig. 5.5(d) are not considered. That is why the algorithm is not able to deal with them. This would be very difficult to detect without the help of an automatic tool. The error leads to the fact that the system will fall to a deadlock state and all robots in the system are unable to locate at the same location. This error could be fixed by finding a strategy to deal with these case. However, we need to take care of the fact that the new strategies may be conflicting with exiting strategies.

Design Errors (difficult to detect by mathematical proof)

This part reports errors of a different kind compared to the one of Section 4.1. One of the main issues to be handled by the algorithm concerns symmetric configurations in which two symmetric robots are supposed to move symmetrically. Let us explain the idea with

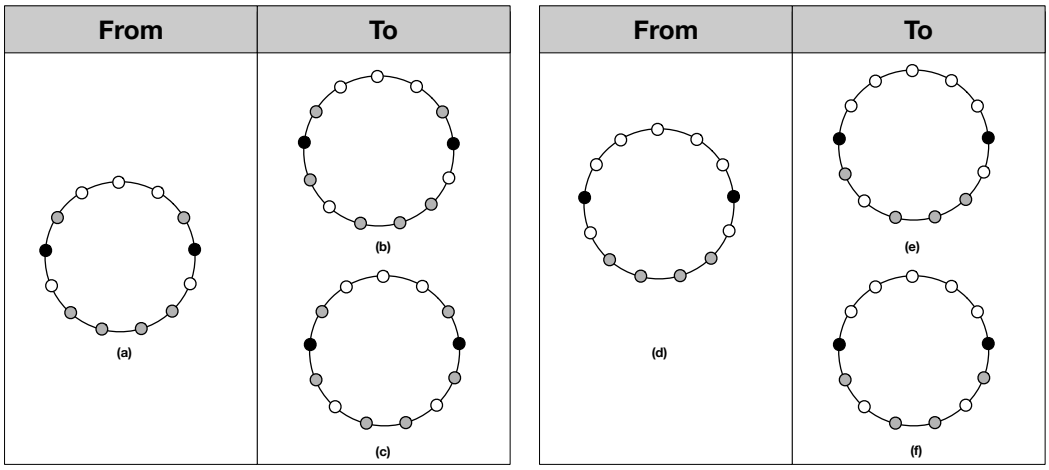


Figure 5.5: Expected executions for the two specific symmetric configurations with two multiplicities.

an example. For the symmetric configuration as shown in Fig. 5.6(a), the two symmetric robots r and r_1 are allowed to symmetrically move. It may happen that both r and r_1 move and the configuration of Fig. 5.6(c) is obtained. It may also happen that only r moves, while r_1 already computed the move, but has not yet moved (means that it holds a pending move) or r_1 has not yet performed its Look-Compute phase and the configuration of Fig. 5.6(b) is obtained. This configuration is an asymmetric configuration that may contain a possible pending move (only robot r_1 knows if it has already computed its move; other robots do not know it). The procedures CHECK-REDUCTION and PENDING-REDUCTION are designed to deal with this case. The robots in such configuration are supposed to detect this situation and the robot r_1 is expected to move in order to reach the configuration of Fig. 5.6(c). It is not so difficult to design procedures that let robots in these configurations to recognized these configurations and move as expected. However, the problem is when these procedures are included in the entire program.

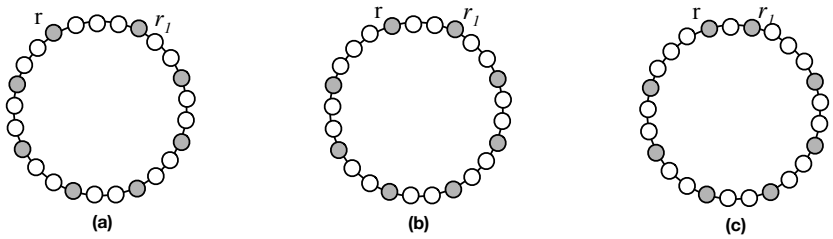


Figure 5.6: Three configurations where (a) is some initial configuration, (b) is the configuration obtained if only one robot moves, and (c) the one obtained if both symmetric robots move.

As written in [21], the procedure CHECK-REDUCTION also recognizes some other configurations where PENDING-REDUCTION should not be applied. One of them is the configuration depicted in Fig. 5.7(b). The authors use this configuration as an ex-

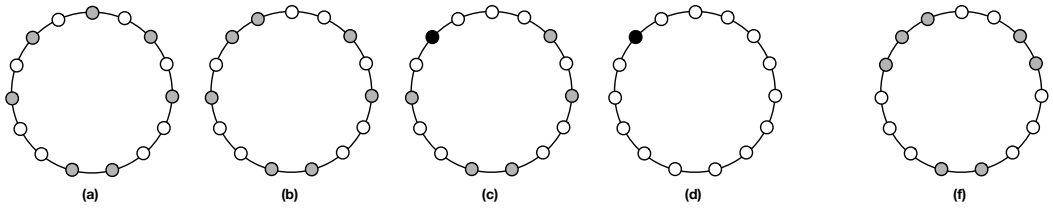


Figure 5.7: Five configurations. The gathering algorithm should follow the sequence (a), (b), (c), (d), but the configuration (f) is actually obtained from (b).

ample to explain the algorithm (Figure 3 in [21]). The algorithm is expected to work as follows: since the configuration of Fig. 5.7(a) is symmetric with one robot on the axis, the robot on the axis has to move, obtaining the configuration of Fig. 5.7(b). This configuration is asymmetric and and contains only one *supermin*[2]. There is one important point emphasized by authors: the robots can recognize that there are no pending moves in this configuration. Therefore, the unique supermin is reduced until a multiplicity is created. In this example, the configuration Fig. 5.7(c) is obtained, and then all robots join the unique multiplicity one-by-one, until achieving the gathering as in Fig. 5.7(d). However, the algorithm actually works as follows: from configuration of Fig. 5.7(b), the robot which is supposed to create a multiplicity does not move. Instead two other robots move and we obtain the configuration of Fig. 5.7(f) (instead of Fig. 5.7(c)). Checking each step of the execution and the transition rules applied, we discover that the source of this error: Robots do not recognize the correct type of the configuration of Fig. 5.7(b), which is classified into a group named W7 that performs the procedure CHECK-REDUCTION and PENDING-REDUCTION. When executing the procedure CHECK-REDUCTION on this configuration, robots incorrectly categorize it as an asymmetric configuration with possible pending moves. Then the PENDING-REDUCTION is executed while it should not be for this configuration. Thus, the two symmetric robots decide to move. The configuration of Fig. 5.7(f) is obtained. This means that the CHECK-REDUCTION and PENDING-REDUCTION procedures are not correct. This error would be very difficult to detect manually. Indeed, the procedures are correct for the configurations for which they are supposed to be used. It is only incorrect because it is also applied to configurations that are not supposed to use these procedures[3].

The second error is in the COLLECT phase. When entering into this phase, the configuration of the system belongs to one of three categories. One of them is called COLL-A-1. They are asymmetric configurations with one multiplicity that can be obtained from symmetric configurations and while satisfying also some other conditions. To simulate how the algorithm works, the authors give the scenario depicted in Fig. 5.8. The configuration of Fig. 5.8(b) is in COLL-A-1. It is at one move from the symmetric configuration of Fig. 5.8(a). When the configuration is in COLL-A-1, the algorithm checks whether the

[2]The definition of *supermin* is given in [21]. One characteristic of a *supermin* is that it is smallest interval.

[3]As explained in Section 2, we analyze the pseudo-code of the algorithms [21]. It is unclear whether such error exists in the informal plaintext description of the algorithm.

current configuration satisfies some conditions. If the conditions are satisfied, it moves the robot and re-establish the previous axis of symmetry, leading to a symmetric configuration with two multiplicities. In this specific case, the configuration of Fig. 5.8(c) is obtained.

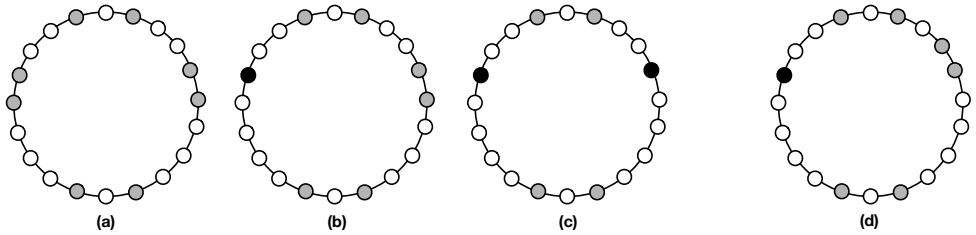


Figure 5.8: Four configurations. The gathering algorithm should follow the sequence (a), (b), (c), but the configuration (d) is actually obtained from (b)

Unfortunately, instead of moving the robot and re-establishes the previous axis of symmetry, the algorithm moves both adjacent robots in the same direction. That means the symmetric configuration with two multiplicities is not obtained. In this specific case, the configuration of Fig. 5.8(d) is obtained. This error is somehow similar to the previous design error, but at a different level; robot instead of configuration. Here, the robot that is supposed to move (from the plaintext description of the algorithm) really moves, but an additional robot also moves.

These design errors prevent the robots from gathering. Said differently, the algorithm fails in gathering all robots in one location. Both errors could certainly be fixed by including additional tests before computing the moves. However adding these tests may lead to other problems and is therefore not straightforward.

Some Minor Errors

We also want to report some other small errors. They could be detected by carefully checking the pseudo-code. However, it may be difficult to find where they are in the code. Fortunately, we can also detect them by model checking.

Minor "typo" error. If a configuration is periodic, it is impossible to gather all robots to one location. Therefore, it is important to detect whether a configuration is periodic or not. To check the periodicity, the authors give the following procedure:

> **Input**: a configuration $C = (q_0, q_1, ..., q_j)$.
>
> **Output**: **true** if C is periodic, **false** otherwise.
>
> 1. *periodic* := **false** .
> 2. **for** $i := 0, 1, ..., j$ **do if** $C = C_i$ **then** *periodic* = **true**.
> 3. **return** *periodic* .

where *periodic* is a boolean variable. It is expected to return *true* if the configuration C is periodic, *false* otherwise. We discover that *periodic* returns *true* also for aperiodic

configurations. This is obviously wrong. The source of the error is that the condition $C = C_0$ is true for any configuration C. Thus, the procedure always returns *true* for any input configuration C. One needs to simply start iterations from $i = 1$ instead of $i = 0$.

Error of inattention. The second error detected is in the main procedure for phase MULTIPLICITY-CREATION whose purpose is to create one multiplicity or two symmetric multiplicities. Since this is the first phase of the algorithm, which deals with initial configurations, all possible initial configurations are partitioned into seven groups. One of them is called *W6*. In this case, the procedure is given as follows:

Input: CT, $C = Q(r) = (q_0, q_1, ..., q_j)$.

Case $CT = W6$

1. $C' := (q_0 + 1, q_1 - 1, ..., q_j)$.
2. **if** $C' = \overline{C'}$ *and* q_0 *is odd* **then** move towards q_0;
3. **else**

   $C'' := (q_0, ..., q_{j-1} - 1, q_j + 1)$.
   if $C''_j = \overline{C''_j}$ *and* q_j *is odd* **then** move towards q_j;

where $C = Q(r) = (q_0, q_1, ..., q_j)$ is the configuration that is perceived by robot r. $\overline{C}$ corresponds to $(q_0, q_j, q_{j-1}, ...q_1)$ in the case where C = $(q_0, q_1, ..., q_{j-1}, q_j)$.

This code is supposed to handle asymmetric configurations that could have been obtained from symmetric configurations with an odd number of nodes and a node-edge symmetric axis. In the above code, C' (or C'') is the configuration of the symmetric configuration. The intention of the authors is to move a robot r in order to reach a new symmetric configuration with the same original axis. One example taken from the Appendix of [21] is given in Fig. 5.9. The configuration of Fig. 5.9(b) can be obtained from the symmetric configuration of Fig. 5.9(a). Thus, the robot r is supposed to move and the configuration of Fig. 5.9(c) is obtained.

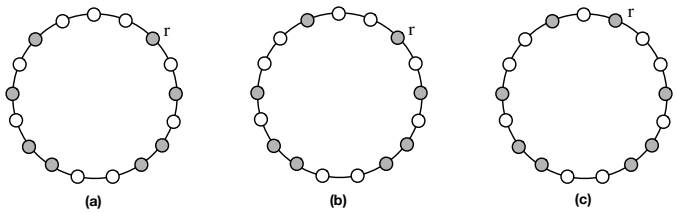


Figure 5.9: Three configurations, where (a) may lead to (b), and (c) presents the expected behavior of the algorithm for the specific asymmetric configuration (b).

However, the algorithm does not make the robot r to move. We found that the error lies in the conditions: q_0 *(or* q_j*) is odd*. Let us take a look at symmetric configurations (e.g as shown in Fig. 5.9(a)) with odd number of nodes and one axis passing through an edge, we can see that the interval crossed by the axis is on an odd interval. That means

that $q_0 + 1$ (or $q_j + 1$) is odd, but not q_0 (or q_j). The error can be fixed by updating the conditions to test if $q_0 + 1$ (or $q_j + 1$) is odd. This error was probably made due to the confusion between the configurations C and C' (or C'').

5.2.4 Summary of Model Checking

It is very common to use case analysis techniques to tackle non-trivial problems, such as robot gathering. The authors in [21] have split the problem into many cases and used different strategies to deal with them. In detail, they have partitioned not only the initial configurations, but also the configurations in each phase. Since the authors need to consider a large number of cases, there is a risk of missing some cases. Moreover, since different strategies are used for each case, there could be some conflicts between these strategies. To prove the algorithm, the authors have to consider all possible cases. However, it is exhausting for a person to figure out all possible executions. Fortunately, a model checker strongly supports to automatically check all possible executions. We have showed how to formally express the desired properties as LTL formulas and conduct the model checking of the algorithm. The model checking has found counterexamples. Analyzing these counterexamples, we found the source of some errors that would be difficult to detect by handmade proof. These errors make the algorithm fail in gathering all robots in one location. We also gave some explanations about why such errors may have occurred. We hope it may be useful to avoid them in the future.

Chapter 6

An Environment and a Domain-Specific Language for Specifying and Model Checking Mobile Ring Robot Algorithms

An environment and a domain-specific language for specifying and model checking mobile robot algorithms on rings (or mobile ring robot algorithms) are proposed. First, we develop Maude Ring Specification Environment (Maude RSE), a ring specification environment that explicitly supports ring-shaped networks. Maude RSE is implemented on top of Maude. Then, we build our domain-specific language, Mobile Ring Robot Maude (MR^2-Maude), on top of Maude RSE. MR^2-Maude makes it possible to specify mobile ring robot algorithms in such a way that the specifications are as close as possible to their mathematical descriptions. One key underlying these tools is pattern matching between ring patterns and ring instances, called "ring pattern matching." Because rings are not commonly available data structures in any existing specification language, we encode ring patterns as sets of sequence patterns and simulate ring pattern matching by pattern matching between sets of sequence patterns and sequence instances, which is proven correct and transparent to both Maude RSE and MR^2-Maude users. MR^2-Maude predefines some LTL formulas as well as atomic propositions to model check that such algorithms enjoy desired properties. The advantages of Maude RSE and MR^2-Maude are demonstrated by case studies analyzing exploration and gathering mobile robot algorithms.

6.1 Overview

This section gives an overview of mobile ring robot algorithms and describes the problems of specifying these algorithms in Maude. It then introduces our solutions.

6.1.1 Mobile Robot Computing on Rings

In the following part, we work under the following assumptions. Robots are *identical*, i.e., they are indistinguishable and all execute the same algorithm. Robots cannot explicitly communicate, but they have the ability to sense their environment and see their relative positions w.r.t. other robots. Robots follow a three-phase behavior: *Look*, *Compute*, and *Move*. During its Look phase a robot takes a snapshot of other robots' positions. The collected information is used in the Compute phase during which the robot decides whether to move or stay idle. A move decided by a robot in a Compute phase may not be immediately conducted by the robot. There may be lag between a Compute phase and the subsequent Move phase and then some other movements by other robots may be done in-between. A move that has been decided by a robot in a Compute phase but has not yet been conducted by the robot in the subsequent Compute phase is called a pending move. In the Move phase, the robot may move to one of the two adjacent according to the decision made by the robot in the preceding Compute phase. Rings may be *anonymous*, that is, there is neither node nor edge labeled. Moreover, robots may be assumed to be *oblivious* and *disoriented*, meaning that they have no memory of past actions and they share no common orientation (no chirality).

6.1.2 Problems

Although mobile ring robot algorithms have been specified in some exiting specification languages, these specifications are usually complicated and lengthy. This is because the *ring* characteristics peculiar to such algorithms are not well supported. Let us illustrate these problems with a simple example. Assume that we specify the ring (the system state on a ring) shown in Fig. 6.1(a), in which robots are *disoriented*. Such a system state can be expressed as a sequence $\{q_0, q_1, \ldots, q_{j-1}, q_j\}$ of intervals, where an interval q_i is the number of consecutive empty nodes between two non-empty nodes, in a view starting from any robot and traversing the ring in one arbitrary direction. We call *configurations* to the particular representation of system states.

The system state shown in Fig. 6.1(a) could be expressed as {0, 3, 1, 2, 1} in the (clockwise) view starting from the one at the bottom. Because it is a ring, the state could be also expressed, starting from other robots, as {3, 1, 2, 1, 0}, {1, 2, 1, 0, 3}, {2, 1, 0, 3, 1}, and {1, 0, 3, 1, 2}. Since robots are disoriented, the state could be expressed as {1, 2, 1, 3, 0}, {0, 1, 2, 1, 3}, {3, 0, 1, 2, 1}, {1, 3, 0, 1, 2}, and {2, 1, 3, 0, 1}

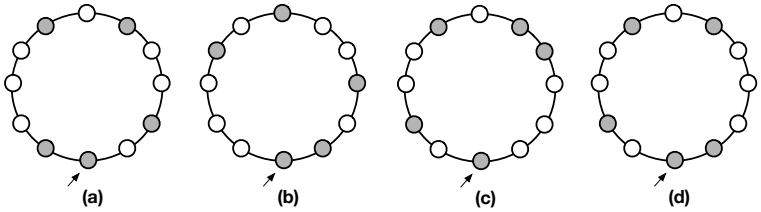


Figure 6.1: The four configurations are considered the same.

by reversing (i.e., considering counterclockwise) these sequences. All these configurations should be considered the same. Generally, given a sequence $\{q_0, q_1, \ldots, q_{j-1}, q_j\}$, the state it expresses is equivalent, in a ring, to all sequences obtained by rotating it — $\{q_0, q_1, \ldots, q_{j-1}, q_j\}$, $\{q_1, \ldots, q_{j-1}, q_j, q_0\}$, $\ldots$, $\{q_j, q_0, q_1, \ldots, q_{j-1}\}$ — and by reversing it — $\{q_j, q_{j-1}, \ldots, q_1, q_0\}$, $\{q_{j-1}, \ldots, q_1, q_0, q_j\}$, $\ldots$, $\{q_0, q_j, q_{j-1}, \ldots, q_1\}$. Moreover, let us assume that robots in Fig. 6.1(a) decide to move and the system reaches the states (b), (c), and (d). Because the ring and the robots are anonymous, all of them are considered the same state. Unfortunately, it is impossible to directly specify this in any existing specification language, such as SPIN [37], DVE [6], and Maude [16]. Actually, the configurations above are considered totally different from any existing specification language point of view, so specifiers are required to implement their own strategies to handle them. Consequently, the specifiers need to specify rings by adapting other defined structures, such as *sets* and *sequences*. For instance, Doan, et al. [29] use binary operators that are associative in Maude.

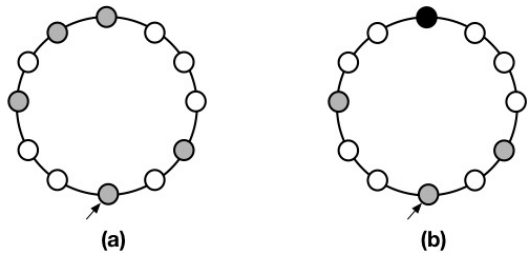


Figure 6.2: (a) A system with two adjacent robots and (b) The obtained system after the movement.

To illustrate the idea used in [29], let us show how to specify a mobile ring robot algorithm in Maude. Given a ring on which there are two robots located at two adjacent nodes, respectively (such two robots may be called adjacent robots), we want to put them together (which is called a *multiplicity*) by moving one to the node at which the other is located, where there is a non-empty node closer to the node at which the former is located than to the other node. For example, in Fig. 6.2(a) we have two adjacent robots on the top, the one on the left is separated from the rest of nodes by one empty node, while the one on the right is separated by three empty nodes. Hence, we would move the one on the left to the node at which the other one is located, as shown Fig. 6.2(b), where the black node indicates that there are two robots. Assuming we use `-1` to denote that two robots are at the same node, we can use a rewrite rule to specify this transition. The source state would use (i) `0` to indicate that two robots are adjacent, (ii) variables `I1` and `I2` to denote the intervals next to the adjacent robots, and (iii) a variable `S` to denote the remaining sequences. Assuming `I2` is larger than `I1`, we will increment the smaller interval (`I1`) and replace `0` (robots are adjacent) by `-1` (robots are in the same node):

```
crl {0, I2, S, I1} => {-1, I2, S, I1 + 1} if I2 > I1 .
```

In the particular case depicted in Fig. 6.2(a) the state could be expressed as {0, 3, 1, 2, 1}. This configuration matches (the left-hand side of) the rule by substituting `I2` with

3, `I1` with 1, and `S` with 1, 2. The state is rewritten to {-1, 3, 1, 2, 2}, which expresses the configuration in Fig. 6.2(b). However, the state shown in Fig. 6.2(a) could be also expressed as another sequence {3, 1, 2, 1, 0}. In this case, there is no substitution such that the sequence can match the rule. For this reason we need another rule to handle it:

```
crl {I2, S, I1, 0} => {-1, I2, S, I1 + 1} if I2 > I1 .
```

The configuration {3, 1, 2, 1, 0} matches this rule by substituting `I2` with 3, `I1` with 1, and `S` with 1, 2. Likewise, we need to have all the rules by rotating and reversing the left-hand side of the first rule to handle all possible sequences.

Splitting problem. The state in Fig. 6.2(a) could be also expressed as {2, 1, 0, 3, 1}, but it is impossible to apply any of the rules above to this configuration. We, thus, need to generate another rule by rotating the first rule to deal with this case. However, the rest of the sequence is split into two sub-sequences at both sides of the previously complete sequence. It, thus, is necessary to split the variable `S` into two variables `S1` and `S2` that denote the remaining sequences on the left- and right-hand sides, respectively.

```
crl {S2, I1, 0, I2, S1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
```

In our theoretical framework we need to formally define and work on splitting and joining (which puts together two sub-sequences that substitute two sequence variables obtained by splitting before) functions that deal with these cases.

Reversing problem. Let's take look at the state in Fig. 6.2(a) (counter-clockwise), which could be expressed as {1, 2, 1, 3, 0}. We need the following rule for this case:

```
crl {I1, S, I2, 0} => {-1, I2, rev(S), I1 + 1} if I2 > I1 .
```

Note that the result of reversing S_1, I, S_2, for S_1 and S_2 sequences and I a natural number, is not S_2, I, S_1, because we might need to reverse the two sequences that substitute S_1 and S_2. We, thus, need in this case to reverse the sequence `S`. The function `rev` reverses a sequence, e.g rev(2,1) is 1, 2. The configuration {1, 2, 1, 3, 0} matches this rule by substituting `I2` with 3, `I1` with 1, and `S` with 2, 1. The state is rewritten to {-1, 3, 1, 2, 2}, which expresses the configuration in Fig. 6.2(b).

Consequently, we need to use 10 rules to specify the transition above-mentioned.
Rotating

```
crl {0, I2, S, I1} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {I2, S, I1, 0} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S, I1, 0, I2} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {I1, 0, I2, S} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S2, I1, 0, I2, S1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
```

Reversing

```
crl {I1, S, I2, 0} => {-1, I2, sev(S), I1 + 1} if I2 > I1 .
crl {0, I1, S, I2} => {-1, I2, sev(S), I1 + 1} if I2 > I1 .
crl {I2, 0, I1, S} => {-1, I2, sev(S), I1 + 1} if I2 > I1 .
crl {S, I2, 0, I1} => {-1, I2, sev(S), I1 + 1} if I2 > I1 .
crl {S1, I2, 0, I1, S2} => {-1, I2, sev(S1), sev(S2), I1 + 1} if I2 > I1 .
```

This makes the specification complicated and specifiers exhausted. If a ring is not faithfully specified, the formal verification of a mobile ring robot algorithm may overlook some cases.

6.1.3 Our solutions

Maude Ring Specification Environment (Maude RSE). One possible way to solve the problems of specifying mobile ring robot algorithms is to develop a specification environment in which rings are explicitly supported. We extend Maude by creating "ring" attributes that allow users to specify rings. The main idea is that given a user ring specification as a ring pattern, Maude RSE generates all corresponding sequence patterns to deal with the "ring" characteristic. For example, in the problem above-mentioned, users only need to specify the first rule while all other rules are automatically generated by Maude RSE. Users, therefore, do not need to deal with the "ring" characteristic, which is handled transparently by Maude RSE.

Mobile Ring Robot Maude (MR^2-Maude). For the problem above-mentioned, using of a ring operator attribute does not require users to generate those rules. However, using such ring attributes requires users to come up with how to handle pending moves or the moves at a multiplicity (see Section 6.3). Analyzing several published algorithms, we conclude that there are three common ways to express the states of mobile ring robot systems as follows: a sequence of intervals, a sequence of all nodes, and a sequence of nodes on which robots are located. For example the system as shown Fig. 6.2(a) could be expressed in these three ways, respectively: {0, 3, 1, 2, 1}, {1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0} in which 0 denotes an empty node and 1 denotes a node occupied by 1 robot, and $\{< r1, 0 >, < r2, 1 >, < r3, 5 >, < r4, 7 >, < r5, 10 >\}$ in which robots are named from r1 to r5 and nodes are labeled from 0 to 10. MR^2-Maude provides three ways to specify mobile ring robot algorithms, covering most of such algorithms proposed in the literature by which user do not need to handle pending moves or the moves at a multiplicity. Moreover, even in Maude RSE, users need to specify some propositions and formulas to model check that such algorithms enjoy desired properties. MR^2-Maude, therefore, predefines some useful LTL formulas as well as atomic propositions specific to such algorithms.

6.2 Maude RSE

Maude RSE is a specification environment that solves the problem of specifying mobile ring robot algorithms. Maude RSE explicitly supports ring-shaped networks. It is reasonable, and saves time and effort, if the environment is built on an available specification

system. For this reason Maude RSE is implemented on top of Maude, a rewriting logic-based programming and specification language, taking advantage of its meta-programming features. This section gives a theory of pattern matching on rings (or "ring-patten matching") that guarantees that our way of dealing with ring-pattern matching makes sense and briefly describes how Maude RSE is built, its architecture, and how to define a ring topology in it.

6.2.1 Ring Patterns

The main question to design and built the environment is how to construct a "ring" data structure, which has "rotative" and "reversible" properties. We define "ring patterns" by using "sequence patterns." Definition 6.2.1 presents sequence patterns and Definition 6.2.8 introduces our definition of ring patterns based on sequence patterns.

Sequences

Let **Elt** be the set of (concrete) elements, **EV** be the set of element variables and **SV** be the set of sequence variables.

Definition 6.2.1 (Sequence Patterns)**.** The set **SP** of sequence patterns are inductively defined as follows:

1. $\varepsilon \in \mathbf{SP}$ (the empty sequence);
2. For each element $e \in \mathbf{Elt}$, $e \in \mathbf{SP}$;
3. For each element variable $E \in \mathbf{EV}$, $E \in \mathbf{SP}$;
4. For each sequence variable $S \in \mathbf{SV}$, $S \in \mathbf{SP}$;
5. For any sequence patterns $SP_1, SP_2 \in \mathbf{SP}$, $SP_1\ SP_2 \in \mathbf{SP}$.

The binary juxtaposition operator used in $SP_1\ SP_2$ is associative, namely that $(SP_1\ SP_2)\ SP_3 = SP_1\ (SP_2\ SP_3)$ for any sequence patterns $SP_1, SP_2, SP_3 \in \mathbf{SP}$. ε is an identity of the binary juxtaposition operator, namely that $\varepsilon\ SP = SP$ and $SP\ \varepsilon = SP$ for any sequence patterns $SP \in \mathbf{SP}$.

Sequence patterns that do not have any variables at all are called sequences. Let $\mathbf{Seq} \subseteq \mathbf{SP}$ be the set of all sequences.

A substitution σ is a function from the disjoint union $\mathbf{EV} \uplus \mathbf{SV}$ of **EV** and **SV** to the disjoint union $\mathbf{Seq} \uplus \mathbf{EV} \uplus \mathbf{SV}$ of **Seq**, **EV** and **SV**. For $E \in \mathbf{EV}$, $\sigma(E)$ is an element $e \in \mathbf{Elt}$ or E and for $S \in \mathbf{SV}$, $\sigma(S)$ is a sequence $seq \in \mathbf{Seq}$ or S. The domain of a substitution σ can be naturally extended to **SP** such that $\sigma(\varepsilon)$ is ε, for an element $e \in \mathbf{Elt}$, $\sigma(e)$ is e and for a sequence pattern $SP_1, SP_2 \in \mathbf{SP}$, $\sigma(SP_1\ SP_2)$ is $\sigma(SP_1)\ \sigma(SP_2)$.

Definition 6.2.2 (Sequence pattern match)**.** Pattern match between $sp \in \mathbf{SP}$ & $seq \in \mathbf{Seq}$ is to find all substitutions σ such that $\sigma(sp) = seq$. Let $sp =?= seq$ be the set of all such substitutions.

Elements, element variables and sequence variables used in sequence patterns are called components in the sequence patterns. For $sp \in \mathbf{SP}$, let $|sp|$ be the number of components in it. $sp \in \mathbf{SP}$ can be in the form $ES_1\ ES_2\ \ldots\ ES_{|sp|}$, where each ES_i is an element, an element variable or a sequence variable. For $sp \in \mathbf{SP}$, let $sp(i)$, where $i \in \{1, 2, \ldots, |sp|\}$, be the ith element ES_i in sp. Let $e, e_1, e_2, \ldots \in \mathbf{Elt}$, $E, E_1, E_2, \ldots \in \mathbf{EV}$, $S, S_1, S_2, \ldots \in \mathbf{SV}$ and $ES, ES_1, ES_2, \ldots \in \mathbf{Elt} \uplus \mathbf{EV} \uplus \mathbf{SV}$. A binary construct $\mathrm{sv}(S, I)$ that is not in $\mathbf{SV}$, where S is a sequence variable and I is either 0 or 1, is used as an extra sequence variable. Let $\mathbf{SSV}$ be $\{\mathrm{sv}(S, I) \mid S \in \mathbf{SV}, I \in \{0, 1\}\}$. $\mathbf{SV} \cup \mathbf{SSV}$ may be used as the set of sequence variables instead of $\mathbf{SV}$.

Definition 6.2.3 (Split sequence patterns)**.** For $sp \in \mathbf{SP}$, $\mathrm{split}(sp)$ is a sequence pattern such that each sequence variable S in sp is replaced with $\mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)$. $\mathrm{split}(\varepsilon) = \varepsilon$, $\mathrm{split}(e) = e$ for $e \in \mathbf{Elt}$, $\mathrm{split}(E) = E$ for $E \in \mathbf{EV}$, $\mathrm{split}(S) = \mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)$ for $S \in \mathbf{SV}$ and $\mathrm{split}(SP_1\ SP_2) = \mathrm{split}(SP_1)\ \mathrm{split}(SP_2)$ for $SP_1, SP_2 \in \mathbf{SP}$.

Definition 6.2.4 (Joining split sequence variables)**.** For $sp \in \mathbf{SP}$ and $seq \in \mathbf{Seq}$, let σ be in $(\mathrm{split}(sp) =?= seq)$. $\mathrm{join}(\sigma)$ is the substitution σ' such that for each sequence variable S in sp $\sigma'(S) = \sigma(\mathrm{sv}(S, 0))\ \sigma(\mathrm{sv}(S, 1))$ and for any other variables X $\sigma'(X) = \sigma(X)$. The domain of join can be naturally extended to the set of substitutions such that $\mathrm{join}(\mathrm{split}(sp) =?= seq)$ is $\{\mathrm{join}(\sigma) \mid \sigma \in (\mathrm{split}(sp) =?= seq)\}$.

rtt (that stands for rotate) takes a sequence pattern sp and returns the sequence pattern obtained by rotating sp clockwise. rev (that stands for reverse) takes a sequence pattern sp and returns the sequence pattern obtained by reversing sp. Let sp be $ES_1\ ES_2 \ldots ES_{|sp|-1}\ ES_{|sp|}$. $\mathrm{rtt}(sp) = ES_{|sp|}\ ES_1\ ES_2 \ldots ES_{|sp|-1}$ and $\mathrm{rev}(sp) = ES_{|sp|}\ ES_{|sp|-1} \ldots ES_2\ ES_1$. Let us suppose that a subscript exp of ES_{exp} used as an element in sp is interpreted as $(exp \bmod |sp|) + 1$.

Definition 6.2.5 (Reversing substitutions)**.** σ_{rev} is defined as follows: for an element $e \in \mathbf{Elt}$ $\sigma_{\mathrm{rev}}(e) = \sigma(e) = e$, for an element variable $E \in (E)$ $\sigma_{\mathrm{rev}}(E) = \sigma(E)$ and for a sequence variable $S \in (SV)$ $\sigma_{\mathrm{rev}}(S) = \mathrm{rev}(\sigma(S))$ and $\sigma_{\mathrm{rev}}(\epsilon) = \sigma(\epsilon)\quad \epsilon\ \sigma_{\mathrm{rev}}(SP_1SP_2) = \sigma_{\mathrm{rev}}(SP_1)\sigma_{\mathrm{rev}}(SP_2)$.

Given two sequence patterns $sp, sp' \in \mathbf{SP}$, $sp \circlearrowright sp'$ holds if there exists a natural number n such that $sp = \mathrm{rtt}^n(sp')$, namely that sp is obtained by rotating sp' finitely many times; $sp \circlearrowleft sp'$ holds if there exists a natural number n such that $sp = \mathrm{rtt}^n(\mathrm{rev}(sp'))$, namely that sp is obtained by reversing sp' once and rotating it finitely many times. For example, let sp and sp' be $e_1\ S_1\ e_2$ and $e_1\ e_2\ S_1$ $(= \mathrm{rtt}(sp))$ and then $(sp \circlearrowright sp')$ holds, while $(sp \circlearrowleft sp')$ does not; let sp and sp' be $e_1\ S_1\ e_2$ and $e_2\ e_1\ S_1$ $(= \mathrm{rtt}(\mathrm{rev}(sp)))$ and then $(sp \circlearrowright sp')$ does not, while $(sp \circlearrowleft sp')$ holds; let sp and sp' be $e_1\ S_1\ e_2$ and $e_2\ e_1\ S_1$ $(= \mathrm{rtt}(\mathrm{rev}(sp)))$ and then both $(sp \circlearrowright sp')$ and $(sp \circlearrowleft sp')$ hold.

Definition 6.2.6 (Sequence pattern contexts)**.** A sequence pattern context is a sequence pattern $sp \in \mathbf{SP}$ in which one component (say, ith component, where $1 \leq i \leq |sp|$) is replaced with a special symbol $\Box$ called a hole, denoted $sp_{(i)}\{\Box\}$. A hole $\Box$ is treated as an element. Let sp be $ES_1 \ldots ES_i \ldots ES_{|sp|}$ and then $sp_{(i)}\{\Box\}$ is $ES_1 \ldots \Box \ldots ES_{|sp|}$.

For a sequence pattern or a sequence pattern context spc and a sequence pattern sp, $spc_{(i)}\{sp\}$ is spc in which the ith component in spc is replaced with sp. $(sp_{(i)}\{\Box\})_{(i)}\{sp(i)\} = sp_{(i)}\{sp(i)\} = sp$.

Definition 6.2.7 (Correspondent components)**.** Let $sp \in \mathbf{SP}$ be $ES_1 \ldots ES_{i-1}\ ES_i\ ES_{i+1} \ldots ES_{|sp|}$, where $1 \leq i \leq |sp|$ and $sp' \in \mathbf{SP}$ be $ES'_1 \ldots ES'_{j-1}\ ES'_j\ ES_{j+1}\ \ldots ES'_{|sp'|}$, where $1 \leq j \leq |sp'|$. If $ES_i\ ES_{i+1}\ \ldots ES_{|sp|}\ ES_1 \ldots ES_{i-1} = ES'_j\ ES'_{j+1}\ \ldots ES'_{|sp'|}\ ES'_1 \ldots ES'_{j-1}$ or $ES_i\ ES_{i+1}\ \ldots ES_{|sp|}\ ES_1 \ldots ES_{i-1} = ES'_j\ ES'_{j-1}\ \ldots ES'_1\ ES'_{|sp'|} \ldots ES'_{j+1}$, then ES'_j is the corresponding component in sp' to ES_i in sp.

Proposition 6.2.1. For any sequence patterns $sp, sp' \in \mathbf{SP}$ and any natural numbers $i \in \{1, \ldots, |sp|\}$ and $j \in \{1, \ldots, |sp'|\}$ such that the jth component $sp'(j)$ in sp' is the corresponding component in sp' to $sp(i)$ in sp, (1) $(sp \circlearrowright sp') \Leftrightarrow sp_{(i)}\{\Box\} \circlearrowright sp'_{(j)}\{\Box\}$ and (2) $(sp \circlearrowleft sp') \Leftrightarrow sp_{(i)}\{\Box\} \circlearrowleft sp'_{(j)}\{\Box\}$.

Proof. (1) ($\Rightarrow$) There exists a natural number m such that $\mathrm{rtt}^m(sp)$ is $sp(i)\ sp(i+1) \ldots sp(i-1)$ and there exits a natural number n such that $\mathrm{rtt}^n(sp')$ is $sp'(j)\ sp'(j+1) \ldots sp'(j-1)$. Because of $sp \circlearrowright sp'$, $\mathrm{rtt}^m(sp) = \mathrm{rtt}^n(sp')$ and then $(\mathrm{rtt}^m(sp))_{(1)}\ \{\Box\} = (\mathrm{rtt}^n(sp'))_{(1)}\{\Box\}$. $\mathrm{rtt}^{-m}((\mathrm{rtt}^m(sp))_{(1)}\{\Box\}) = sp_{(i)}\{\Box\}$ and $\mathrm{rtt}^{-n}((\mathrm{rtt}^n(sp'))_{(1)}\{\Box\}) = sp'_{(j)}\{\Box\}$. Therefore, $sp_{(i)}\{\Box\} \circlearrowright sp'_{(j)}\{\Box\}$. ($\Leftarrow$) There exists a natural number m such that $\mathrm{rtt}^m(sp_{(i)}\{\Box\})$ is $\Box\ sp(i+1) \ldots sp(i-1)$ and there exits a natural number n such that $\mathrm{rtt}^n(sp'_{(j)}\{\Box\})$ is $\Box\ sp'(j+1) \ldots sp'(j-1)$. Because of $sp_{(i)}\{\Box\} \circlearrowright sp'_{(j)}\{\Box\}$, $\mathrm{rtt}^m(sp_{(i)}\{\Box\}) = \mathrm{rtt}^n(sp'_{(j)}\ \{\Box\})$ and then $(\mathrm{rtt}^m(sp_{(i)}\{\Box\}))_{(1)}\{sp(i)\} = (\mathrm{rtt}^n(sp'_{(j)}\{\Box\}))_{(1)}\ \{sp(i)\}$. Because $sp'(j)$ in sp' is the corresponding component to $sp(i)$ in sp, $sp'(j) = sp(i)$. Therefore, $(\mathrm{rtt}^m(sp_{(i)}\{\Box\}))_{(1)}\ \{sp(i)\} = (\mathrm{rtt}^n(sp'_{(j)}\{\Box\}))_{(1)}\{sp'(j)\}$ and then $\mathrm{rtt}^{-m}((\mathrm{rtt}^m(sp_{(i)}\ \{\Box\}))_{(1)}\{sp(i)\}) = sp$ and $\mathrm{rtt}^{-n}((\mathrm{rtt}^n(sp'_{(j)}\{\Box\}))_{(1)}\{sp'(j)\}) = sp'$. Thus, $sp \circlearrowright sp'$.

(2) ($\Rightarrow$) There exists a natural number m such that $\mathrm{rtt}^m(sp)$ is $sp(i)\ sp(i+1) \ldots sp(i-1)$ and there exits a natural number n such that $\mathrm{rtt}^n(\mathrm{rev}(sp'))$ is $sp'(j)\ sp'(j-1) \ldots sp'(j+1)$. Because of $sp \circlearrowleft sp'$, $\mathrm{rtt}^m(sp) = \mathrm{rtt}^n(\mathrm{rev}(sp'))$ and then $(\mathrm{rtt}^m(sp))_{(1)}\ \{\Box\} = (\mathrm{rtt}^n\ (\mathrm{rev}\ (sp')))_{(1)}\{\Box\}$. $\mathrm{rtt}^{-m}\ ((\mathrm{rtt}^m(sp))_{(1)}\ \{\Box\}) = sp_{(i)}\{\Box\}$ and $\mathrm{rtt}^{-n}\ ((\mathrm{rtt}^n(\mathrm{rev}(sp')))_{(1)}\{\Box\}) = (\mathrm{rev}(sp'))_{(j)}\{\Box\}$. Therefore, $sp_{(i)}\{\Box\} \circlearrowleft sp'_{(j)}\{\Box\}$. ($\Leftarrow$) There exists a natural number m such that $\mathrm{rtt}^m(sp_{(i)}\{\Box\})$ is $\Box\ sp(i+1) \ldots sp(i-1)$ and there exits a natural number n such that $\mathrm{rtt}^n(\mathrm{rev}(sp'_{(j)}\{\Box\}))$ is $\Box\ sp'(j-1) \ldots sp'(j+1)$. Because of $sp_{(i)}\{\Box\} \circlearrowleft sp'_{(j)}\{\Box\}$, $\mathrm{rtt}^m(sp_{(i)}\{\Box\}) = \mathrm{rtt}^n(\mathrm{rev}(sp'_{(j)}\ \{\Box\}))$ and then $(\mathrm{rtt}^m(sp_{(i)}\ \{\Box\}))_{(1)}\{sp(i)\} = (\mathrm{rtt}^n(\mathrm{rev}(sp'_{(j)}\{\Box\})))_{(1)}\ \{sp(i)\}$. Because $sp'(j)$ in sp' is the corresponding component to $sp(i)$ in sp, $sp'(j) = sp(i)$. Therefore, $(\mathrm{rtt}^m(sp_{(i)}\{\Box\}))_{(1)}\ \{sp(i)\} = (\mathrm{rtt}^n(\mathrm{rev}(sp'_{(j)}\ \{\Box\})))_{(1)}\{sp'(j)\}$ and then $\mathrm{rtt}^{-m}((\mathrm{rtt}^m(sp_{(i)}\{\Box\}))_{(1)}\{sp(i)\}) = sp$ and $\mathrm{rtt}^{-n}((\mathrm{rtt}^n(\mathrm{rev}\ (sp'_{(j)}\{\Box\})))_{(1)}\{sp'(j)\}) = \mathrm{rev}(sp')$. Thus, $sp \circlearrowleft sp'$. □

Rings

Definition 6.2.8 (Rings)**.** For $sp \in \mathbf{SP}$, $[sp]$ is called a ring pattern and satisfies (1) the rotative law ($[sp] = [\mathrm{rtt}(sp)]$) and (2) the reversible law ($[sp] = [\mathrm{rev}(sp)]$). When sp is a sequence $seq \in \mathbf{Seq}$, $[seq]$ is called a ring.

Proposition 6.2.2. For any sequence patterns $sp, sp' \in \mathbf{SP}$ and natural numbers m, n, if $[sp] = [sp']$, then (1) $[sp] = [\mathrm{rtt}^m(sp')]$ and (2) $[sp] = [\mathrm{rev}^n(sp')]$.

Proof. Let us suppose $[sp] = [sp']$. (1) By induction on m. (1.1) Base case ($m = 0$) can be discharged from the assumption $[sp] = [sp']$. (1.2) Induction case ($m = k + 1$). From Definition 6.2.8, $[\mathrm{rtt}^k(sp')] = [\mathrm{rtt}^{k+1}(sp')]$. From this and the induction hypothesis $[sp] = [\mathrm{rtt}^k(sp')]$, $[sp] = [\mathrm{rtt}^{k+1}(sp')]$. (2) By induction on n. (2.1) Base case ($n = 0$) can be discharged from the assumption $[sp] = [sp']$. (2.2) Induction case ($n = k + 1$). From Definition 6.2.8, $[\mathrm{rev}^k(sp')] = [\mathrm{rev}^{k+1}(sp')]$. From this and the induction hypothesis $[sp] = [\mathrm{rev}^k(sp')]$, $[sp] = [\mathrm{rev}^{k+1}(sp')]$. □

For any sequence patterns $sp, sp' \in \mathbf{SP}$, if $([sp] = [sp']) \Rightarrow [sp] = [\mathrm{rtt}(sp')] \wedge [sp] = [\mathrm{rev}(sp')]$, then $[sp] = [\mathrm{rtt}(sp)]$ and $[sp] = [\mathrm{rev}(sp)]$ because the equivalence relation is reflexive, namely $[sp] = [sp]$. Therefore, Definition 6.2.8 can be rephrased as follows:

Definition 6.2.9 (Another definition of rings)**.** For $sp, sp' \in \mathbf{SP}$, $[sp] = [sp']$ is inductively defined as follows: (1) $[sp] = [sp]$ and (2) if $[sp] = [sp']$, then $[sp] = [\mathrm{rtt}(sp')]$ and $[sp] = [\mathrm{rev}(sp')]$.

Let sp be $ES_1\ ES_2 \ldots\ ES_{|sp|-1}\ ES_{|sp|}$. $\mathrm{rtt}^{-1}(sp)$ is $ES_2\ \ldots\ ES_{|sp|-1}\ ES_{|sp|}\ ES_1$ and $\mathrm{rev}^{-1}(sp)$ is $ES_{|sp|}\ ES_{|sp|-1} \ldots ES_1\ ES_2$. Therefore, $\mathrm{rtt}^{-1} = \mathrm{rev} \circ \mathrm{rtt} \circ \mathrm{rev}$ and $\mathrm{rev}^{-1} = \mathrm{rev}$.

Proposition 6.2.3. For any sequence patterns $sp, sp' \in \mathbf{SP}$, if $[sp] = [sp']$, then $[sp] = [\mathrm{rtt}^{-1}(sp')]$ and $[sp] = [\mathrm{rev}^{-1}(sp')]$.

Proof. This is derived from $\mathrm{rtt}^{-1} = \mathrm{rev} \circ \mathrm{rtt} \circ \mathrm{rev}$, $\mathrm{rev}^{-1} = \mathrm{rev}$ and Proposition 6.2.2, □

Definition 6.2.10 (Ring pattern match)**.** For $sp \in \mathbf{SP}$ and $seq \in \mathbf{Seq}$, pattern match between $[sp]$ and $[seq]$ is to find all substitutions σ such that $[\sigma(sp)] = [seq]$. Let $[sp] = ?= [seq]$ be the set of all such substitutions.

Definition 6.2.11 (Sequences rotated and/or reversed)**.** For $sp \in \mathbf{SP}$, $[[sp]]$ is the set of sequence patterns inductively defined as follows: (1) $sp \in [[sp]]$ and (2) if $sp' \in [[sp]]$, then $\mathrm{rtt}(sp') \in [[sp]]$ and $\mathrm{rev}(sp') \in [[sp]]$.

Proposition 6.2.4. For any sequence patterns $sp, sp' \in \mathbf{SP}$, if $sp' \in [[sp]]$, then $\mathrm{rtt}^{-1}(sp') \in [[sp]]$ and $\mathrm{rev}^{-1}(sp') \in [[sp]]$.

Proof. This is derived from $\mathrm{rtt}^{-1} = \mathrm{rev} \circ \mathrm{rtt} \circ \mathrm{rev}$, $\mathrm{rev}^{-1} = \mathrm{rev}$ and Definition 6.2.11. □

Proposition 6.2.5. For any sequences $seq, seq', seq'' \in \mathbf{Seq}$ and any natural number $i \in \{1, \ldots, |seq|\}$ and $j \in \{1, \ldots, |seq'|\}$ such that $seq'(j)$ is the correspond component to $seq(i)$, seq is $e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_{|sp|}$ and seq' is $e'_1 \ldots e'_{j-1}\ e'_j\ e'_{i+1} \ldots\ e'_{|sp'|}$, (1) if $seq_{(i)}\{\Box\} \circlearrowright seq'_{(j)}\{\Box\}$, then $[seq_{(i)}\{seq''\}] = [seq'_{(j)}\ \{seq''\}]$, and (2) if $seq_{(i)}\{\Box\} \circlearrowleft seq'_{(j)}\{\Box\}$, then $[seq_{(i)}\ \{seq''\}] = [seq'_{(j)}\{\mathrm{rev}(seq'')\}]$.

Proof. (1) Let seq be $e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_{|seq|}$ and seq' be $e'_1 \ldots e'_{j-1}\ e'_j\ e'_{j+1} \ldots e'_{|seq'|}$. (1) Because $seq_{(i)}\{\Box\} \circlearrowright seq'_{(j)}\{\Box\}$, $\Box\ e_{i+1} \ldots e_{|seq|}\ e_1 \ldots e_{i-1} = \Box\ e'_{j+1} \ldots e'_{|seq'|}\ e'_1 \ldots e'_{j-1}$ and then $seq''\ e_{i+1} \ldots e_{|seq|}\ e_1 \ldots e_{i-1} = seq''\ e'_{j+1} \ldots e'_{|seq'|}\ e'_1 \ldots e'_{j-1}$. Therefore, $[seq_{(i)}\{seq''\}] = [seq'_{(j)}\ \{seq''\}]$. (2) Because $seq_{(i)}\{\Box\} \circlearrowleft seq'_{(j)}\{\Box\}$, $\Box\ e_{i+1} \ldots e_{|seq|}\ e_1 \ldots e_{i-1} = \Box\ e'_{j-1} \ldots e'_1\ e'_{|seq'|} \ldots e'_{j+1}$ and then $seq''\ e_{i+1} \ldots e_{|seq|}\ e_1 \ldots e_{i-1} = seq''\ e'_{j-1} \ldots e'_1\ e'_{|seq'|} \ldots e'_{j+1}$. $\mathrm{rev}(seq''\ e'_{j-1} \ldots e'_1\ e'_{|seq'|} \ldots e'_{j+1})$ is $e'_{j+1} \ldots e'_{|seq'|}\ e'_1 \ldots e'_{j-1}\ \mathrm{rev}(seq'')$. Thus, $[seq_{(i)}\{seq''\}] = [seq'_{(j)}\{\mathrm{rev}(seq'')\}]$. □

Lemma 6.2.1. For sequence patterns $sp, sp' \in \mathbf{SP}$, $(sp' \in [[sp]]) \Leftrightarrow ([sp] = [sp'])$.

Proof. $(sp' \in [[sp]]) \Rightarrow ([sp] = [sp'])$ is proved by induction on Definition 6.2.11. (1) Base case in which $sp \in [[sp]]$ holds. $[sp] = [sp]$ holds because of Definition 6.2.4. (2) Induction case in which $\mathrm{rtt}(sp') \in [[sp]]$ and $\mathrm{rev}(sp') \in [[sp]]$ hold. $sp' \in [[sp]]$ holds from Proposition 6.2.4. From the induction hypothesis $([sp] = [sp'])$ and Definition 6.2.9, therefore, $[sp] = [\mathrm{rtt}(sp')]$ and $[sp] = [\mathrm{rev}(sp')]$ hold.

$(sp' \in [[sp]]) \Leftarrow ([sp] = [sp'])$ is proved by induction on Definition 6.2.9. (1) Base case in which $[sp] = [sp]$ holds. $sp \in [[sp]]$ holds because of Definition 6.2.11. (2) Induction case in which $[sp] = [\mathrm{rtt}(sp')]$ and $[sp] = [\mathrm{rev}(sp')]$ hold. $[sp] = [sp']$ holds from Proposition 6.2.3. From the induction hypothesis $(sp' \in [[sp]])$ and Definition 6.2.11, therefore, $\mathrm{rtt}(sp') \in [[sp]]$ and $\mathrm{rev}(sp') \in [[sp]]$ hold. □

Let sp be $e_1\ S_1\ e_4\ S_2$ and sp' be $\mathrm{rev}(sp)$, namely $S_2\ e_4\ S_1\ e_1$. Clearly, $sp' \in [[sp]]$ and $[sp] = [sp']$. Let us consider a substitution σ such that $\sigma(S_1) = e_2\ e_3$, $\sigma(S_2) = e_5\ e_6$ and $\sigma(X) = X$ for any other variable X. $\sigma(sp)$ is $e_1\ e_2\ e_3\ e_4\ e_5\ e_6$ and $\sigma(sp')$ is $e_5\ e_6\ e_4\ e_2\ e_3\ e_1$. Clearly, $\sigma(sp') \notin [[\sigma(sp)]]$ and $[\sigma(sp)] \neq [\sigma(sp')]$. If $sp \circlearrowright sp'$ does not hold but $sp \circlearrowleft sp'$ holds, we need to reverse the sequence that replaces each sequence variable. $\sigma_{\mathrm{rev}}(sp')$ is $e_6\ e_5\ e_4\ e_3\ e_2\ e_1$. Therefore, $\sigma_{\mathrm{rev}}(sp') \in [[\sigma(sp)]]$ and $[\sigma(sp)] = [\sigma_{\mathrm{rev}}(sp')]$.

Lemma 6.2.2. For any sequence pattern $sp \in \mathbf{SP}$ and any substitution σ, for each $sp' \in [[sp]]$ if $sp \circlearrowright sp'$, then $[\sigma(sp)] = [\sigma(sp')]$; if $sp \circlearrowleft sp'$, then $[\sigma(sp)] = [\sigma_{\mathrm{rev}}(sp')]$.

Proof. By induction on the number of element and sequence variable occurrences in sp.

(1) Base case in which the number is 0. Because sp does not have any variables, $\sigma(sp) = sp$, $\sigma(sp') = sp'$ and $\sigma_{\mathrm{rev}}(sp') = sp'$. From Lemma 6.2.1, $[sp] = [sp']$.

(2) Induction case in which the number is $k+1$. Let us arbitrarily choose a component that is a variable in sp and the component be the ith component $sp(i)$ in sp. Let sp be $sp_1\ sp(i)\ sp_2$. sp' can be obtained by rotating and/or reversing sp and then must have the correspondent component in sp' to $sp(i)$ in sp. Then, sp' can be $sp'_1\ sp(i)\ sp'_2$.

(2.1) Let us suppose that $sp \circlearrowright sp'$ holds. From Proposition 6.2.1, $(sp_1 \Box sp_2) \circlearrowright (sp'_1 \Box sp'_2)$. By induction hypothesis, $[\sigma(sp_1 \Box sp_2)] = [\sigma(sp'_1 \Box sp'_2)]$ and then $[\sigma(sp_1) \Box \sigma(sp_2)] = [\sigma(sp'_1) \Box \sigma(sp'_2)]$. From Lemma 6.2.5, $[\sigma(sp_1)\ \sigma(sp(i))\ \sigma(sp_2)] = [\sigma(sp'_1)\ \sigma(sp(i))\ \sigma(sp'_2)]$. Hence, $[\sigma(sp_1\ sp(i)\ sp_2)] = [\sigma(sp'_1\ sp(i)\ sp'_2)]$.

(2.2) Let us suppose that $sp \circlearrowleft sp'$ holds. From Proposition 6.2.1, $(sp_1 \Box sp_2) \circlearrowleft (sp'_1 \Box sp'_2)$. By induction hypothesis, $[\sigma(sp_1 \Box sp_2)] = [\sigma_{\mathrm{rev}}(sp'_1 \Box sp'_2)]$ and then $[\sigma(sp_1) \Box \sigma(sp_2)] = [\sigma_{\mathrm{rev}}(sp'_1) \Box \sigma_{\mathrm{rev}}(sp'_2)]$. From Lemma 6.2.5, $[\sigma(sp_1)\ \sigma(sp(i))\ \sigma(sp_2)] =$

$[\sigma_{\mathrm{rev}}(sp'_1)\ \mathrm{rev}(\sigma(sp(i)))\ \sigma_{\mathrm{rev}}(sp'_2)]$. Because $\mathrm{rev}(\sigma(sp(i))) = \sigma_{\mathrm{rev}}(sp(i))$, $[\sigma(sp_1\ sp(i))\ sp_2)] = [\sigma_{\mathrm{rev}}(sp'_1\ sp(i)\ sp'_2)]$. □

Definition 6.2.12 (Ring pattern match simulated (1))**.** For $sp \in \mathbf{SP}$ and $seq \in \mathbf{Seq}$, pattern match between sp and $[[seq]]$ is to find all substitutions σ such that $\sigma(sp) = seq'$ for some $seq' \in \mathbf{Seq}$. Let $sp =?= [[seq]]$ be the set of all such substitutions.

Lemma 6.2.3. For any sequence pattern $sp \in \mathbf{SP}$ and any sequence $seq \in \mathbf{Seq}$, $([sp] =?= [seq]) = (sp =?= [[seq]])$.

Proof. Let $\sigma \in ([sp] =?= [seq])$. $[\sigma(sp)] = [seq]$ by Definition 6.2.10. $\sigma(sp) \in [[seq]]$ due to Lemma 6.2.1. Thus, $\sigma \in (sp =?= [[seq]])$.

Let $\sigma \in (sp =?= [[seq]])$. Let $seq' \in [[seq]]$ such that $\sigma(sp) = seq'$. $[seq'] = [seq]$ due to Lemma 6.2.1 and then $[\sigma(sp)] = [seq]$. Hence, $\sigma \in ([sp] =?= [seq])$ □

Definition 6.2.13 (Ring pattern match simulated (2))**.** For $sp \in \mathbf{SP}$ and $seq \in \mathbf{Seq}$, pattern match between $[[sp]]$ and seq is to find all substitutions σ such that $\sigma'(sp') = seq$ for some substitution σ' and some $sp' \in [[sp]]$ and if $sp \circlearrowright sp'$, then $\sigma = \sigma'$ and if $sp \circlearrowleft sp'$, then $\sigma = \sigma'_{\mathrm{rev}}$. Let $[[sp]] =?= seq$ be the set of all such substitutions.

Note that $([[sp]] =?= seq) \subset ([sp] =?= [seq])$ but $([sp] =?= [seq]) \not\subset ([[sp]] =?= seq)$.

Lemma 6.2.4. For any sequence pattern $sp \in \mathbf{SP}$, any sequence $seq \in \mathbf{Seq}$ and any substitution $\sigma \in (sp =?= [[seq]])$, there exist σ' and a sequence $seq' \in [[seq]]$ such that $\sigma = \mathrm{join}(\sigma')$, $\sigma'(\mathrm{split}(sp)) = seq'$ and there exists $sp' \in [[\mathrm{split}(sp)]]$ such that $\sigma'(sp') = seq$. Besides, $\sigma \in \mathrm{join}([[\mathrm{split}(sp)]] =?= seq)$.

Proof. Let sp be $ES_1\ ES_2 \ldots ES_m$ and seq be $e_1\ e_2 \ldots e_n$.

If there exists $i \in \{1, \ldots, m\}$ such that $\sigma(ES_i)$ is $\ldots e_n\ e_1\ \ldots$ or $\ldots e_1\ e_n \ldots$, ES_i is a sequence variable S that is replaced with $\mathrm{sv}(S,0)\ \mathrm{sv}(S,1)$ in $\mathrm{split}(sp)$.

If $\sigma(S)$ is $\ldots e_n\ e_1 \ldots$, then $\sigma'(\mathrm{sv}(S,0))$ is $\ldots e_n$, $\sigma'(\mathrm{sv}(S,1))$ is $e_1 \ldots$ and $\sigma'(\mathrm{sv}(S',0))$ is $\sigma(S')$ and $\sigma'(\mathrm{sv}(S',1))$ is ε for any other sequence variable S' in sp, and $\sigma'(E) = \sigma(E)$ for any element variable E in sp. By the construction of σ', $\sigma = \mathrm{join}(\sigma')$ and $\sigma'(\mathrm{split}(sp)) = \sigma(sp)$, where $\sigma(sp) \in [[seq]]$. Let sp' be $\mathrm{sv}(S,1)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})\ \mathrm{sv}(S,0)$. Then, $sp' \in [[\mathrm{split}(sp)]]$ and $\sigma'(sp') = seq$. Therefore, $\sigma' \in ([[\mathrm{split}(sp)]] =?= seq)$ because of $sp \circlearrowright sp'$ from Definition 6.2.13. Hence $\sigma \in \mathrm{join}([[\mathrm{split}(sp)]] =?= seq)$ from Definition 6.2.4.

If $\sigma(S)$ is $\ldots e_1\ e_n\ \ldots$, then $\sigma'(\mathrm{sv}(S,0))$ is $\mathrm{rev}(\ldots e_1)$, $\sigma'(\mathrm{sv}(S,1))$ is $\mathrm{rev}(e_n \ldots)$ and $\sigma'(\mathrm{sv}(S',0))$ is $\mathrm{rev}(\sigma(S'))$ and $\sigma'(\mathrm{sv}(S',1))$ is ε for any other sequence variable S' in sp, and $\sigma'(E) = \sigma(E)$ for any element variable E in sp. By the construction of σ', $\sigma = \mathrm{join}(\sigma'_{\mathrm{rev}})$ and $\sigma'_{\mathrm{rev}}(\mathrm{split}(sp)) = \sigma(sp)$, where $\sigma(sp) \in [[seq]]$. Let sp' be $\mathrm{rev}(\mathrm{sv}(S,1)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})\ \mathrm{sv}(S,0))$. Then, $sp' \in [[\mathrm{split}(sp)]]$ and $\sigma'_{\mathrm{rev}}(sp') = seq$. Therefore, $\sigma'_{\mathrm{rev}} \in ([[\mathrm{split}(sp)]] =?= seq)$ because of $sp \circlearrowleft sp'$ from Definition 6.2.13. Hence $\sigma \in \mathrm{join}\ ([[\mathrm{split}(sp)]] =?= seq)$ from Definition 6.2.4.

If there exists no $i \in \{1, \ldots, m\}$ such that $\sigma(ES_i)$ is $\ldots e_n\ e_1\ \ldots$ or $\ldots e_1\ e_n \ldots$, there must be $i \in \{1, \ldots, m\}$ such that $\sigma(ES_i)$ is e_1, $e_1 \ldots$ or $\ldots e_1$. If $\sigma(ES_i)$ is e_1, there are two possible cases: (1) $\sigma(ES_i\ ES_{i+1} \ldots ES_{i-1}) = seq$ and (2) $\sigma(ES_i\ ES_{i-1} \ldots ES_{i+1}) = seq$.

Case (1) can be treated in the same way as the case in which $\sigma(ES_i)$ is $e_1 \ldots$. In either case, $\sigma'(\mathrm{sv}(S', 0))$ is $\sigma(S')$ and $\sigma'(\mathrm{sv}(S', 1))$ is ε for any sequence variable S' in sp, and $\sigma'(E) = \sigma(E)$ for any element variable E in sp. By the construction of σ', $\sigma = \mathrm{join}(\sigma')$ and $\sigma'(\mathrm{split}(sp)) = \sigma(sp)$, where $\sigma(sp) \in [[seq]]$. Let sp' be $\mathrm{split}(ES_i)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})$. Then, $sp' \in [[\mathrm{split}(sp)]]$ and $\sigma'(sp') = seq$. Therefore, $\sigma' \in ([[\mathrm{split}(sp)]] = ?= seq)$ because of $sp \circlearrowright sp'$ from Definition 6.2.13. Hence $\sigma \in \mathrm{join}([[\mathrm{split}(sp)]] =?= seq)$ from Definition 6.2.4.

Case (2) can be treated in the same way as the case in which $\sigma(ES_i)$ is $\ldots e_1$. In either case, $\sigma'(\mathrm{sv}(S', 0))$ is $\mathrm{rev}(\sigma(S'))$ and $\sigma'(\mathrm{sv}(S', 1))$ is ε for any sequence variable S' in sp, and $\sigma'(E) = \sigma(E)$ for any element variable E in sp. By the construction of σ', $\sigma = \mathrm{join}(\sigma'_{\mathrm{rev}})$ and $\sigma'_{\mathrm{rev}}(\mathrm{split}(sp)) = \sigma(sp)$, where $\sigma(sp) \in [[seq]]$. Let sp' be $\mathrm{rev}(\mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})\ \mathrm{split}(ES_i))$. Then, $sp' \in [[\mathrm{split}(sp)]]$ and $\sigma'_{\mathrm{rev}}(sp') = seq$. Therefore, $\sigma'_{\mathrm{rev}} \in ([[\mathrm{split}(sp)]] =?= seq)$ because of $sp \circlearrowleft sp'$ from Definition 6.2.13. □

Lemma 6.2.5. For any sequence pattern $sp \in \mathbf{SP}$, any sequence $seq \in \mathbf{Seq}$ and any substitution $\sigma \in \mathrm{join}([[\mathrm{split}(sp)]] =?= seq)$, $\sigma \in ([sp] =?= [seq])$.

Proof. Let sp be $ES_1\ ES_2 \ldots ES_m$ and seq be $e_1\ e_2 \ldots e_n$. Let σ' be an arbitrary substitution in $([[\mathrm{split}(sp)]] =?= seq)$ from which σ is constructed, namely that $\sigma = \mathrm{join}(\sigma')$. Let $sp' \in [[\mathrm{split}(sp)]]$ such that $\sigma''(sp') = seq$, if $\mathrm{split}(sp) \circlearrowright sp'$, then $\sigma' = \sigma''$ and if $\mathrm{split}(sp) \circlearrowleft sp'$, then $\sigma' = \sigma''_{\mathrm{rev}}$. There are four possible cases: (1) sp' is $\mathrm{split}(E_i)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})$, (2) sp' is $\mathrm{rev}(\mathrm{split}(E_i))\ \mathrm{rev}(\mathrm{split}(ES_{i-1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i+1}))$. (3) sp' is $\mathrm{sv}(S, 1)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})\ \mathrm{sv}(S, 0)$ and (4) sp' is $\mathrm{sv}(S, 0)\ \mathrm{rev}(\mathrm{split}(ES_{i-1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i+1}))\ \mathrm{sv}(S, 1)$.

(1) For each ES_j for $j = 1, 2, \ldots, m$, we calculate $\sigma''(\mathrm{split}(\ ES_j))$ and $\sigma(ES_j)$. There are three possible cases: (1.1) ES_j is an element e, (1.2) ES_j is an element variable E and (1.3) ES_j is a sequence variable S. (1.1) $\sigma''(\mathrm{split}(e)) = e$ and $\sigma(e) = e = \sigma''(\mathrm{split}(e))$. (1.2) $\sigma''(\mathrm{split}(E)) = \sigma''(E)$ and $\sigma(E) = (\mathrm{join}(\sigma''))(E) = \sigma''(E) = \sigma''(\mathrm{split}(E))$. (1.3) $\sigma''(\mathrm{split}(S))$ and $\sigma(S)$ are calculated as follows:

$$\sigma''(\mathrm{split}(S)) = \sigma''(\mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1))$$

$$\sigma(S) = (\mathrm{join}(\sigma''))(S) = \sigma''(\mathrm{sv}(S, 0))\ \sigma''(\mathrm{sv}(S, 1)) = \sigma''(\mathrm{split}(S))$$

Therefore, $\sigma(ES_j) = \sigma''((\mathrm{split}(ES_j)))$ and then $\sigma(ES_i\ ES_{i+1}\ \ldots ES_{i-1})$ is calculated as follows:

$$\begin{aligned}
&\sigma(ES_i\ ES_{i+1} \ldots ES_{i-1}) = \sigma(ES_i)\ \sigma(ES_{i+1}) \ldots \sigma(ES_{i-1}) \\
&= \sigma''((\mathrm{split}(ES_i)))\ \sigma''((\mathrm{split}(ES_{i+1}))) \ldots \sigma''((\mathrm{split}(ES_{i-1}))) \\
&= \sigma''(\mathrm{split}(ES_i)\ \mathrm{split}(ES_{i+1}) \ldots \mathrm{split}(ES_{i-1})) \\
&= \sigma''(sp')
\end{aligned}$$

Because $\sigma''(sp') = seq$ from the assumption, $\sigma(ES_i\ ES_{i+1} \ldots\ ES_{i-1}) = seq$. Because $sp \circlearrowright ES_i\ ES_{i+1} \ldots ES_{i-1}$, $[\sigma(sp)] = [\sigma(ES_i\ ES_{i+1} \ldots ES_{i-1})]$ from Lemma 6.2.2. Thus, $[\sigma(sp)] = [seq]$ and then $\sigma \in ([sp] =?= [seq])$.

(2) For each ES_j for $j = 1, 2, \ldots, m$, we calculate $\sigma''(\mathrm{rev}(\ \mathrm{split}(ES_j)))$ and $\sigma(ES_j)$. There are three possible cases: (2.1) ES_j is an element e, (2.2) ES_j is an element variable E and (2.3) ES_j is a sequence variable S. (2.1) $\sigma''(\mathrm{rev}(\mathrm{split}(e))) = e$ and $\sigma(e) = e = \sigma''(\mathrm{rev}(\mathrm{split}(e))) = \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(e))))$. (2.2) $\sigma''(\mathrm{rev}(\mathrm{split}(E))) = \sigma''(E)$ and $\sigma(E) = (\mathrm{join}(\sigma''_{\mathrm{rev}}))(E) = \sigma''(E) = \sigma''(\mathrm{rev}(\mathrm{split}(E))) = \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(E))))$. (2.3) $\sigma''(\mathrm{rev}(\mathrm{split}(S)))$ and $\sigma(S)$ are calculated as follows:

$\sigma''(\mathrm{rev}(\mathrm{split}(S))) = \sigma''(\mathrm{rev}(\mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)))$
$= \sigma''(\mathrm{sv}(S, 1)\ \mathrm{sv}(S, 1))$

$\sigma(S) = (\mathrm{join}(\sigma''_{\mathrm{rev}}))(S) = \sigma''_{\mathrm{rev}}(\mathrm{sv}(S, 0))\ \sigma''_{\mathrm{rev}}(\mathrm{sv}(S, 1))$
$= \mathrm{rev}(\sigma''(\mathrm{sv}(S, 0)))\ \mathrm{rev}(\sigma''(\mathrm{sv}(S, 1)))$
$= \mathrm{rev}(\sigma''(\mathrm{sv}(S, 1))\ \sigma''(\mathrm{sv}(S, 0))) = \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(S))))$

Therefore, $\sigma(ES_j) = \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(ES_j))))$ and then $\sigma(ES_{i+1} \ldots ES_{i-1}\ ES_i)$ is calculated as follows:

$\sigma(ES_{i+1} \ldots ES_{i-1}\ ES_i) = \sigma(ES_{i+1}) \ldots \sigma(ES_{i-1})\ \sigma(ES_i)$
$= \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(ES_{i+1})))) \ldots$
$\quad \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(ES_{i-1}))))\ \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(ES_i))))$
$= \mathrm{rev}(\sigma''(\mathrm{rev}(\mathrm{split}(ES_i))))$
$\qquad \sigma''(\mathrm{rev}(\mathrm{split}(ES_{i-1}))) \ldots \sigma''(\mathrm{rev}(\mathrm{split}(ES_{i+1}))))$
$= \mathrm{rev}(\sigma''(sp'))$

Because $\sigma''(sp') = seq$ from the assumption, $\sigma(ES_{i+1} \ldots\ ES_{i-1}\ ES_i) = \mathrm{rev}(seq)$. Because $sp \circlearrowleft ES_{i+1} \ldots ES_{i-1}\ ES_i$, $[\sigma(sp)] = [\sigma(ES_{i+1} \ldots ES_{i-1}\ ES_i)]$ from Lemma 6.2.2. Moreover, $[seq] = [\mathrm{rev}(seq)]$ from Proposition 6.2.2. Thus, $[\sigma(sp)] = [seq]$ and then $\sigma \in ([sp] =?= [seq])$.

(3) $\mathrm{rtt}(sp')$ is calculated as follows:

$\mathrm{rtt}(sp')$
$= \mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)\ \mathrm{rev}(\mathrm{split}(ES_{i+1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i-1}))$
$= \mathrm{split}(ES_i)\ \mathrm{rev}(\mathrm{split}(ES_{i+1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i-1}))$

Because $\sigma''(sp') = seq$, there exists a natural number k such that $\sigma''(\mathrm{rtt}(sp')) = \mathrm{rtt}^k(seq)$. As what has been done for case (1), we have $\sigma(ES_i\ ES_{i+1} \ldots ES_{i-1}) = \mathrm{rtt}^k(seq)$. Because $sp \circlearrowleft ES_i\ ES_{i+1} \ldots ES_{i-1}$, $[\sigma(sp)] = [\sigma(ES_i\ ES_{i+1} \ldots ES_{i-1}]$ from Lemma 6.2.2. Moreover, $[seq] = [\mathrm{rtt}^k(seq)]$ from Proposition 6.2.2. Thus, $[\sigma(sp)] = [seq]$ and then $\sigma \in ([sp] =?= [seq])$.

(4) $\mathrm{rtt}(sp')$ is calculated as follows:

$\mathrm{rtt}(sp')$
$= \mathrm{sv}(S, 1)\ \mathrm{sv}(S, 0)\ \mathrm{rev}(\mathrm{split}(ES_{i-1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i+1}))$
$= \mathrm{rev}(\mathrm{split}(ES_i))\ \mathrm{rev}(\mathrm{split}(ES_{i-1})) \ldots \mathrm{rev}(\mathrm{split}(ES_{i+1}))$

Because $\sigma''(sp') = seq$, there exists a natural number k such that $\sigma''(\text{rtt}(sp')) = \text{rtt}^k(seq)$. As what has been done for case (2), we have $\sigma(ES_{i+1} \dots ES_{i-1}\ ES_i) = \text{rev}(\text{rtt}^k(seq))$. Because $sp \circlearrowleft ES_{i+1} \dots ES_{i-1}\ ES_i$, $[\sigma(sp)] = [\sigma(ES_{i+1} \dots ES_{i-1}\ ES_i)]$ from Lemma 6.2.2. Moreover, $[seq] = [\text{rev}(\text{rtt}^k(seq))]$ from Proposition 6.2.2. Thus, $[\sigma(sp)] = [seq]$ and then $\sigma \in ([sp] =?= [seq])$. □

Theorem 1. *For any sequence pattern* $sp \in \mathbf{SP}$ *and any sequence* $seq \in \mathbf{Seq}$, join ([[split (sp)]] $=?=$ seq) = ($[sp] =?= [seq]$).

Proof. It is derived from Lemmas 6.2.3, 6.2.4, and 6.2.5. □

Theorem 1 implies and guarantees that a ring pattern is represented by a set of sequence patterns by rotating and reversing elements including splitting and joining sequence variables. The splitting and reversing problems are solved by splitting sequence patterns, joining split sequence variables, and reversing substitutions (Definitions 6.2.3, 6.2.4, and 6.2.5).

6.2.2 Extending Maude with Ring Attributes

Rewriting logic is a natural model for specifying concurrent and communicating systems. Several specification languages based on rewriting logic, such as ASF+SDF, Maude, and ELAN have been designed and implemented. It has been demonstrated in [29, 30, 35, 36, 61, 62] that Maude allows programmers to specify distributed algorithms/systems more succinctly than the others. Moreover, Maude supports meta-programming. A meta-program is a program that takes a program (or specification) as input and performs some useful computations, such as analyzing the program and transforming it into another. Thanks to these meta-programming features, we can build Maude RSE on top of Maude. In particular, we extend Full-Maude [16], which is an extension of Maude written in Maude itself and provides extra features to extend Maude.

We extend Maude by creating two `ring` attributes that indicate that a given constructor for sequences behaves as a ring. The specification environment is built as depicted in Fig. 6.3. A specification in the environment is considered as a user specification, which may contain specifications of a ring topology that would not be supported by the standard Maude engine. The main player in the system is Transformer that takes a user specification and transforms it into an ordinary Maude specification, which can be handled by Maude. Transformer is a meta-program that handles specifications. The theory under the transformer is given in Section 6.2.1. The main idea is that Transformer analyzes a user specification to find out all functions related with rings and then modify them by adding extra equations/rules that handle rings. Given a user specification as a ring pattern, the transformer generates all corresponding sequence patterns following the rotative law and the reversible law (see Definition 6.2.8). Intuitively, given a ring pattern $[ES_1 \dots ES_i \dots ES_n]$, where $ES_1, \dots, ES_i, \dots, ES_n \in \mathbf{C}$ are components, Transformer generates as the left-hand side of a rule: n rotative patterns $[ES_1 \dots ES_i \dots ES_n], \dots, [ES_i \dots ES_n\ ES_1], \dots, [ES_n\ ES_1 \dots ES_i]$ and n reversible patterns $[ES_1\ ES_n \dots ES_i], \dots, [ES_i \dots ES_1 \dots ES_n], \dots, [ES_n \dots ES_i \dots ES_1]$. When ES_i (i = 1, 2, . . . , n) is a

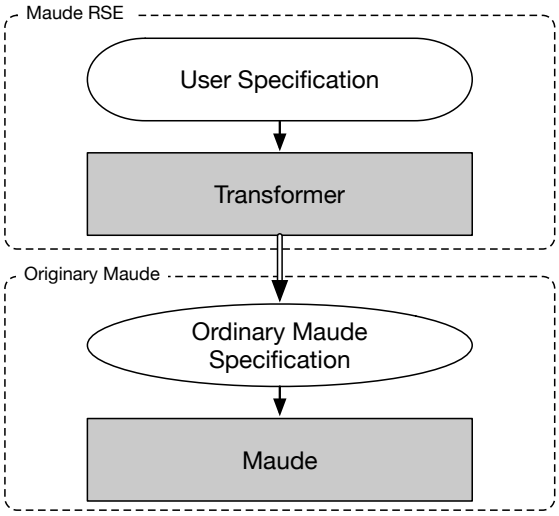


Figure 6.3: Architecture of Maude RSE.

variable, it is split following Definition 6.2.3. The correctness of Transformer is proven in Theorem 1. We can basically use the right-hand side of the given rule as the right-hand side for the other $2n$-1 patterns generated as the left-hand side. We, however, need to reverse sequences that substitute sequence variables occurring in the right-hand side for the n reversible patterns (see Definition 6.2.5). For example, to specify the transition as shown Fig. 6.2, users only need to specify one rule assumed as follows:

```
crl {0, I2, S, I1} => {-1, I2, S, I1 + 1} if I2 > I1 .
```

Intuitively, Transformer generates all possible rules by

1. Splitting: All sequence variables are splitted.

```
crl {0, I2, S1, S2, I1} => {-1, I2, S1, S2, I1 + 1}  if I2 > I1 .
```

2. Rotating: All components in the sequence of the left-hand side are rotated.

```
crl {0, I2, S1, S2, I1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {I2, S1, S2, I1, 0} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {S1, S2, I1, 0, I2} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {S2, I1, 0, I2, S1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {I1, 0, I2, S1, S2} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
```

3. Joining: Some pairs of sequence variables that are splitted from one sequence variable and appeared in the splitted order are joined.

```
crl {0, I2, S, I1} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {I2, S, I1, 0} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S, I1, 0, I2} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S2, I1, 0, I2, S1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {I1, 0, I2, S} => {-1, I2, S, I1 + 1} if I2 > I1 .
```

4. Reversing: All sequences on the left-hand sides are reversed. `rev` is an operator that reverses a sequence, e.g `rev(1, 2)` = `2, 1`.

```
crl {0, I2, S, I1} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {I2, S, I1, 0} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S, I1, 0, I2} => {-1, I2, S, I1 + 1} if I2 > I1 .
crl {S2, I1, 0, I2, S1} => {-1, I2, S1, S2, I1 + 1} if I2 > I1 .
crl {I1, 0, I2, S} => {-1, I2, S, I1 + 1} if I2 > I1 .

crl {I1, S, I2, 0} => {-1, I2, rev(S), I1 + 1} if I2 > I1 .
crl {0, I1, S, I2} => {-1, I2, rev(S), I1 + 1} if I2 > I1 .
crl {I2, 0, I1, S} => {-1, I2, rev(S), I1 + 1} if I2 > I1 .
crl {S1, I2, 0, I1, S2} => {-1, I2, rev(S1), rev(S2), I1 + 1} if I2 > I1 .
crl {S, I2, 0, I1} => {-1, I2, rev(S), I1 + 1} if I2 > I1 .
```

Because the result of the transformation is a standard Maude specification, we can guarantee that all Maude facilities, such as the LTL model checker, can be directly used for user specifications.

6.2.3 Syntax declaration

This section shows how to define rings in the new environment. We consider two kinds of rings: *oriented* rings in which the orientation of a ring such as clockwise order and anti-clockwise order is taken into account, and *disoriented* rings in which there is no orientation of a ring. In Maude, types are called *sorts*. A sort denotes the set of elements of the same type. For example, the sort `Nat` denotes the set of natural numbers. A sort is a *subsort* of another sort if and only if the set denoted by the former is a subset of the one by the latter, and the latter is called a supersort of the former. Keywords `sort` and `subsort` are used to declare a sort and a subsort relation, respectively. Elements of a given sort are built by constructors, with keyword `op`, together with keyword `ctor`, given the arity and the coarity. Moreover, operators can have equational axioms, such as being associative for which keyword `assoc` is used.

We first consider *disoriented* rings, implemented by the `ring` attribute. In particular, rings are constructed as a sequence of elements with this attribute. Let us assume that the sort for elements is `Elem`. The sort for sequences of elements is `Seq`. The configurations of a system as rings could be defined as follows:

```
subsort Elem < Seq .
op emp : -> Seq [ctor] .
op __ : Seq Seq -> Seq [ctor assoc id: emp] .
op {_} : Seq -> Config [ring ctor].
```

where the keyword `ctor` indicates that the operator is a constructor. An operator without any argument is called a *constant*, such as `emp`, which stands for the empty sequence. Underscores are placeholders where arguments are placed. Similarly, `id: emp` indicates that operator `emp` is the identity element of the juxtaposition (empty syntax) operator

`__`. `Seq` is a supersort of `Elem`, which means that each `Elem` is treated as the singleton sequence only consisting of this element. The operator `__` is used to construct sequences of elements: for $\mathtt{s}_1$ and $\mathtt{s}_2$ of sort `Seq`, $\mathtt{s}_1\mathtt{s}_2$ has sort `Seq`. The structure `__` is presented just as an example; it could be replaced by any other structure that depends on the user's preferences, such as _,_ and _|_. Likewise, the structure `{_}` is an optional preference. A configuration is defined as a ring structure that is specified as a bounded sequence of elements. Because a ring is disoriented, the mirror image of a ring represents the same state represented by the ring. Therefore, for $s_0, s_1, \ldots, s_{j-1}, s_j \in \mathit{Elem}$, $\{s_j\ s_{j-1}\ \ldots\ s_1\ s_0\}$ represents the same state represented by $\{s_0\ s_1\ \ldots\ s_{j-1}\ s_j\}$. All the configurations $\{s_0\ s_1\ \ldots\ s_{j-1}\ s_j\}$, $\{s_1\ \ldots\ s_{j-1}\ s_j\ s_0\}$, $\ldots$, $\{s_j\ s_0\ s_1\ \ldots\ s_{j-1}\}$, and $\{\ s_j\ s_{j-1}\ \ldots\ s_1\ s_0\}$, $\{s_{j-1}\ \ldots\ s_1\ s_0\ s_j\}$, $\ldots$, $\{\ s_0\ s_j\ s_{j-1}\ \ldots\ s_1\}$ are the same; they represent the same state.

For *oriented* rings, the environment provides the `r-ring` attribute that could be considered as a sub-class attribute of the `ring` attribute. For the `r-ring` attribute, the ring and its mirror image do not necessarily represent the same state. Given a configuration expressed as the ring $\{s_0\ s_1\ ...\ s_{j-1}\ s_j\}$, all the following configurations are as the same: $\{s_0\ s_1\ ...\ s_{j-1}\ s_j\}$ $\{s_1\ \ldots\ s_{j-1}\ s_j\ s_0\}$, $\ldots$, and $\{s_j\ s_0\ s_1\ \ldots\ s_{j-1}\}$.

Let us show how to specify the system as shown in Fig. 6.1. An interval can be specified as an element of the sort `Int` that is used for denoting integers. The structure `_ _` is used to construct sequences of intervals. The configurations could be defined as follows:

```
subsort Int < Seq .
op emp : -> Seq [ctor] .
op __ : Seq Seq -> Seq [ctor assoc id: emp] .
op {_} : Seq ->  Config [ring ctor].
```

The ring as shown in the configurations Fig. 6.1(a) is expressed as {0, 3, 1, 2, 1} as well as {3, 1, 2, 1, 0} , {1, 2, 1, 0, 3} , {2, 1, 0, 3, 1}, and {1, 0, 3, 1, 2} and so on because of `ring`.

6.2.4 Applications

This section presents how to formalize and specify a mobile robot algorithm on a ring in Maude RSE. We specify and model check an exploration algorithm with stop (Section 2.1.2).

Robots Exploration on Ring under ASYNC

Exploration Problem. We consider the problem of exploring with stop a disoriented ring populated by a group of identical, oblivious mobile robots. About timing assumption, the ASYNC (or asynchronous) model is considered. In addition, there may be more than one robot located at one. Each robot can distinguish whether a node is empty, occupied by one robot, or more than one robot. When there are more than one robot, the node is called a multiplicity (or a tower). The problem of exploring with stop requires that regardless of the initial placement of the robots, each node must be visited by at least one robot and the robots must be in a configuration in which they all remain idle.

Exploration Algorithm [33]. The exploration algorithm checks whether, starting from any initial configuration without towers, all robots explore the entire ring and reach a configuration in which they all remain idle (no rules can be applied). The algorithm works following a sequence of three distinct phases: Set-Up, Tower-Creation, and Exploration. The Set-Up phase transforms any initial configuration into one that is in a predetermined set of configurations (called *no-towers-final*) with special properties. After that, the Tower-Creation phase and then the Exploration phase are executed. Finally, all nodes are visited and no robot will make any further moves.

Formal Specification of Exploration Algorithm

Let us suppose that there are n nodes denoted u_0 u_1, ..., u_{n-1} and each node may be occupied by more than one robot. The multiplicity of robots located at node u_i is denoted d_i: $d_i = 0$ indicates that there are no robots, $d_i = 1$ indicates that there is exactly one robot, and $d_i = 2$ indicates that there are more than one robot. We consider how to express a configuration and how to describe an action as a state transition.

State Expressions. So far the state of an algorithm on a ring shape network has been represented as a sequence of elements. For this system, each element is a node of the ring. To deal with pending moves, we denote a node as a tuple $\langle n_i, p_i, ps_i \rangle$, where n_i denotes the multiplicity of robots located at the node, p_i denotes a pending move, and ps_i denotes the list of pending moves from other robots that moves to this one after they execute their pending moves. n_i can take the values: 0, 1, and 2, which correspond to the value of d_i. p_i is either a pending move moving to one of its adjacent nodes or `nil` meaning that there are no pending moves (or the pending move is staying idle). When a robot at node u_i takes the snapshot of a configuration at some moments, the sequence $n_i, n_{i+1}, \ldots, n_{i-1}$ of the multiplicities taken from the snapshot is called the robot's view of the snapshot. Such views are used to represent pending moves. ps_i is either a set of pending moves from other robots that move to this node after they execute their pending moves or `emp` (the empty set). Because the ring is disoriented, a robot needs to find its pending move in the pending move set of one of its adjacent nodes, so it can distinguish the direction of the movement. The sort `Pending` denotes pending moves, `PendingSet` pending move sets, and `Node` nodes. A configuration is expressed as a ring of nodes. The sort for configurations is `Config`, which is declared with the `ring` attribute:

```
subsort Node < Seq .
op <_,_,_> : Nat Pending PendingSet -> Node [ctor] .
op emp : -> Seq [ctor] .
op __ : Seq Seq -> Seq [ctor assoc id: emp] .
op {_} : Seq -> Config [ctor ring].
```

The structure `<_,_,_>` is used to construct nodes. For $n \in Nat$, $p \in Pending$, $ps \in PendingSet$, we have $\langle n, p, ps \rangle \in Node$. A configuration is defined as `{_}`, which takes as argument a sequence of nodes. A term of sort S is either a variable or $f(t_1, \ldots, t_n)$ if f is

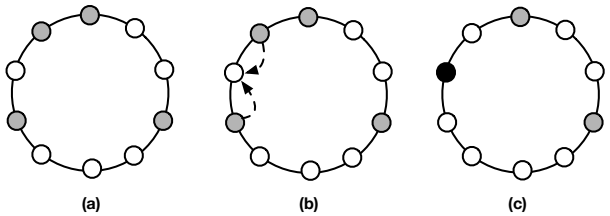


Figure 6.4: Some configurations. A dashed arrow represents a pending move. Black nodes represent multiplicities.

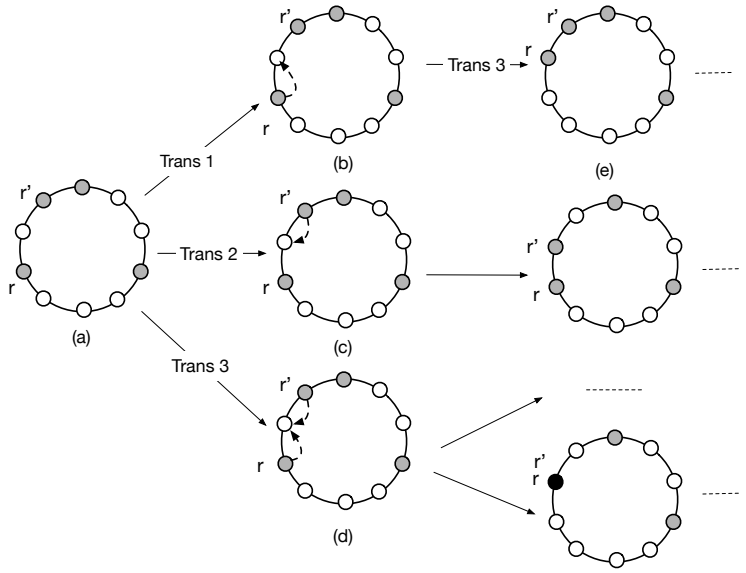


Figure 6.5: Transition graph from the initial configuration (a).

an operator declared as $f : S_1 \ldots\ S_n \rightarrow S$ $(n \geq 0)$ and $t_1, \ldots, t_n$ are terms of $S_1, \ldots, S_n$. Constructor terms are those terms consisting of only constructor operators and variables. Ground constructor terms hence are those composed of constructors only and no variables. Ground constructor terms of sort `Config` express concrete states of the system. For example, the initial configuration of the system as shown in Fig. 6.4(a) could be expressed as the view of the robot at the top as $\{\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, nil,\ emp\ \rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\}$. Let v and v' be the views of the two robots holding pending moves: 1, 0, 1, 1, 0, 0, 1, 0, 0, 0 and $1, 1, 0, 0, 1, 0, 0, 0, 1, 0$. The configuration of the system with two pending moves as shown in Fig. 6.4(b) could be expressed as $\{\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 0, nil,\ emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, v, emp\rangle\ \langle 0, nil, (v; v')\rangle\ \langle 1, v',\ emp\rangle\}$. The configuration of the system in Fig. 6.4(c) could be expressed as $\{\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil,\ emp\rangle\ \langle 2, nil, emp\rangle\ \langle 0, nil, emp\rangle\}$. Note that a configuration is defined as a ring.

State Transitions. Because the *Compute* phase uses the snapshot of the system taken in the *Look* phase as input and a robot does not perform any movements during two consecutive phases, to model the system we combine the two phases into one called *Look-Compute* phase in which a robot takes the snapshot of the system and computes a move. When either (1) a robot takes the snapshot of the system and then computes a move, or (2) a robot executes its pending move, the current configuration of the system changes to another. Such changes are called a state transition (a transition for short). A transition is expressed as a pair (l, r), where l and r are configurations. Let us examine the following scenario. Assume that both robots r and r' in Fig. 6.5(a) are allowed to move and the robot r looks at the system and computes a move. The configuration of the system might be transferred to the one in Fig. 6.5(b). The transition is named *trans1* and expressed by the pair ($\{\langle\ 1, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, nil, emp\ \rangle\}$, $\{\langle\ 1, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 0, nil, emp\ \rangle\ \langle\ 1, v, emp\ \rangle\ \langle\ 0, nil, (v; emp)\ \rangle\ \langle\ 1, nil, emp\ \rangle\}$). The graph in Fig. 6.5 shows the possible transitions from the initial configuration. We formalize state transitions by means of rewrite rules. Each rewrite rule is defined only over `Config` in the form $L \Rightarrow R$ such that L is a constructor term. For example, the following rewrite rule describes the action when a robot performs its pending move.

```
crl [Pending]: {S1 < 1, P, PS > < N, P', (P ; PS') > S2}
=>  {S1 < 0, nil, PS > <  N + 1, P', PS' > S2}
if nonMul({S1 < 1, P, PS > < N,  P', (P ; PS') > S2}).
```

where `S1` and `S2` are variables of sort `Seq`, `P`, and `P'` are variables of sort `Pending`, `PS` and `PS'` are variables of sort `PendingSet`, and `N` is a variable of sort `Nat`. The function `nonMul` returns `true` when the configuration has no multiplicity and `false` otherwise.
The rule above is a conditional rule named `Pending` with the condition specified in the `if` part. Since it is applied to a state in which there are no multiplicities, `N` is either 0 or 1. The configuration {S1 ⟨ 1, P, PS ⟩ ⟨ N, P', (P ; PS') ⟩ S2} stands for any state such that the robot ⟨ 1, P, PL ⟩ has a pending move P and the next node is ⟨ N, P', (P ; PL') ⟩ in which P is in the set of pending moves. In addition to these two nodes, such a state may have some more nodes that are expressed as $S1$ and $S2$. The ground constructor term $\{\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, v, emp\rangle\ \langle 0, nil, (v; emp)\rangle\ \langle 1, nil, emp\rangle\}$ expresses the state of Fig. 6.5(b). The left-hand side of the above rewrite rule matches this ground term by substituting $S1, P, PS, N, P', PS'$ and $S2$ with $\langle 1, nil, emp\rangle\langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 0, nil, emp\rangle$, v, emp, nil, v, emp and $\langle 1, nil, emp\rangle$, and the rewrite rule can be applied to the term, changing it to $\{\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\ \langle 1, nil, emp\rangle\ \langle 1, nil, emp\rangle\}$ expressing the state in Fig. 6.5(e). In this way, a single rewrite rule stands for a set of state transitions.

Let us analyze the following example to show how the new environment helps to specify the algorithm. Assume that the algorithm is specified in standard Maude. This means

that a configuration is defined as a sequence of nodes enclosed by { and }. Considering the (counter-clockwise) system in Fig. 6.5(b), given as $\{\langle 1, nil, emp\rangle\langle 0, nil, (v; emp)\rangle\langle 1, v, emp\rangle$ $\langle 1, nil, emp\rangle$ $\langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\langle 0, nil, emp\rangle$ $\langle 0, nil, emp\rangle$ $\langle 0, nil, emp\rangle\}$. However, this term does not match the left-hand side of the rule above. To handle this case, we need to specify another rule as follows:

```
crl [Pending]: {S1 < N,  P', (P ; PS') > < 1, P, PS > S2}
=> {S1 <  N + 1, P', PS' > < 0, nil, PS >  S2}
if nonMul({S1 < N,  P', (P ; PS') > < 1, P, PS >  S2}).
```

Consequently, we need to use more rules to specify the algorithm. Fortunately, in Maude RSE, the first rule `Pending` is enough to cover all possible cases.

Model Checking

We use the Maude LTL model checker to verify the algorithm in the previous section. By definition in the algorithm, at the end of each phase the system should be in some configurations with special properties. The authors in [33] give some important theorems, such as *Theorem 3.1* and *Theorem 3.2*, that must hold to guarantee the correctness of the algorithm. These theorems are used to model check the algorithm. We have formally expressed these theorems as LTL formulas [38]. We take here the formalization of *Theorem 3.1* as an example. *Theorem 3.1* states a property that must be satisfied at the end of the Set-Up phase: any initial configuration is transformed into a *no-towers-final* configuration. We define the atomic propositions `endOf` and `SetUp` as follows:

- The proposition `endOf` holds if and only if the Set-Up phase has finished.
- The proposition `SetUp` holds if and only if the current state does not have towers and robots are located adjacent to others in one or two groups.

The theorem then is expressed as the LTL formula:

```
theorem3-1 = [] (endOf -> SetUp) /\ <> endOf .
```

where `[]` is the always operator and `<>` is the eventually operator.
Intuitively, the formula states that it is always true that the phase Set-Up will finally terminate and whenever the phase has been just over, then `SetUp` is true. This means that the *Theorem 3.1* is satisfied at the end of the Set-Up phase. As the result of the model checking, no counterexamples are found. This makes us more confident on the correctness of *Theorem 3.1*.

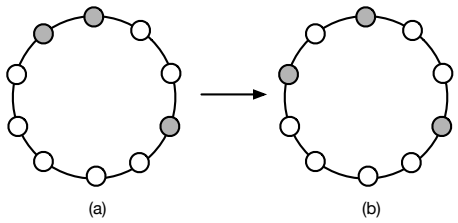


Figure 6.6: (a) A system with 10 nodes and 3 robots and (b) The system obtained after a robot moved.

6.3 MR^2-Maude

In the previous section, we presented how to specify and model check mobile ring robot algorithms in Maude RSE, in which two ring operator attributes are available. The two ring operator attributes make it possible to substantially reduce the number of rules required to specify mobile ring robot algorithms because the attributes take into account the rotative and reversible properties of rings. It would, however, be preferable that mobile ring robot algorithms can be specified such that their specifications are closer to their mathematical descriptions. More specifically, dealing with (i) pending moves, which have been calculated but not performed yet, and (ii) moves of robots in a multiplicity adds extra complexities to specifications.

For example, let us show how to specify the following action: in this case (see Fig. 6.6(a)), the isolated robot should not move, and among the two other robots, only the furthest one (w.r.t. the isolated robot) should compute a move to approach to the isolated node (Fig. 6.6(b)). To represent system states, we could use sequences of natural numbers (more precisely, 0, 1, and 2) such that each element corresponds to one node: 0, 1, and 2 stand for no robot, one robot, and more than one robot at the node, respectively. The configuration in Fig. 6.6(a), assuming the node that is the neighbor on the left of the two adjacent robots is the initial one, is expressed as {0, 1, 1, 0, 0, 1, 0, 0, 0, 0}. In the sense of distributed computing, the action is simply given as the following rule:

```
rl {0, 1, 1, 0, 0, 1, 0, 0, 0, 0} => {1, 0, 1, 0, 0, 1, 0, 0, 0, 0} .
```

However, robots must follow the Look, Compute, and Move cycle. A robot looks at the system and then calculates a movement, but it does not perform it instantly; the movement is performed an arbitrary time after being computed. When either (i) a robot takes the snapshot of the system and then computes a move, or (ii) a robot executes its pending move, the current configuration changes to another (see Fig. 6.7). Thus, it is necessary to store the information about a pending move in each node. One possible solution as presented in Section 6.2.4 is that we could add more information to each node to handle pending moves. A node could be expressed as a tuple $\langle n, p, ps \rangle$, where n denotes the multiplicity of robots presented at node, p denotes a pending move, and ps denotes the set of pending moves from other robots that move to this note after they execute their pending moves. Let v be 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, which is the view of the robot that is supposed to move. The above rule is revised as:

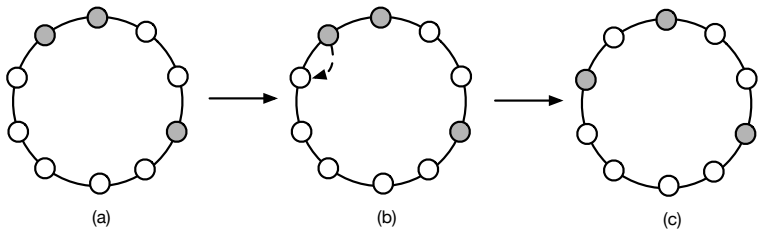


Figure 6.7: (a) A system with 10 nodes and 3 robots, (b) a robot takes the snapshot of the system and computes a move, and (c) the robot executes its pending move.

```
rl {< 0, nil, emp >, < 1, nil, emp >, < 1, nil, emp >, < 0, nil, emp >,
    < 0, nil, emp >, < 1, nil, emp >, < 0, nil, emp >, < 0, nil, emp >,
    < 0, nil, emp > , < 0, nil, emp >}
=>
   {< 0, nil, (v ; emp) >, < 1, v, emp >, < 1, nil, emp >, < 0, nil, emp >,
    < 0, nil, emp >, < 1, nil, emp >, < 0, nil, emp >, < 0, nil, emp >,
    < 0, nil, emp >, < 0, nil, emp >} .
```

The question is how to perform the move since the configuration may be changed after a robot has computed the move. We, thus, need at least one more rule to perform a pending move (see the pending rule mentioned in Section 6.2.4). Moreover, to model check a property for an algorithm, it is required to define several LTL atomic propositions. MR^2-Maude, therefore, provides a syntax to specify and analyze ring robot algorithms such that MR^2-Maude users do not need to deal with how to specify pending moves as well as the moves at a multiplicity. All these tasks are taken by MR^2-Maude and transparent to users. For example, users are able to specify the action above-mentioned in a way that it looks closer to its mathematical description without use of any extra rules. MR^2-Maude is built on top of Maude RSE. MR^2-Maude then inherits two ring operator attributes. In the next sections we present MR^2-Maude syntax and commands.

6.3.1 Syntax

We present in this section the syntax of MR^2-Maude for specifying states and transitions.

State Expression

MR^2-Maude provides three ways to specify mobile ring robot algorithms, covering most of such algorithms proposed in the literature: *ring-interval*, *ring-fix*, and *ring-location*.

Ring-interval: a system is described as a sequence of intervals, where an interval is the number of consecutive empty nodes between two non-empty nodes. In this case a ring is expressed as a sequence of numbers enclosed by { and }. Each number in the sequence denotes the size of the interval between one robot and the next one. For example, the

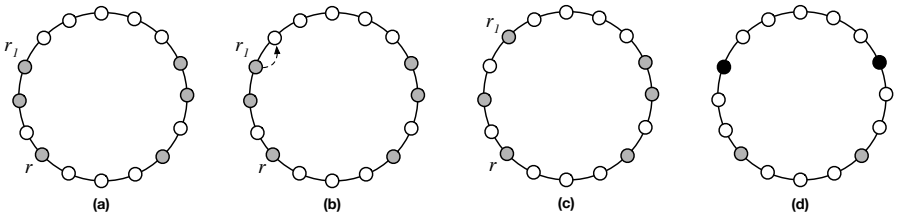


Figure 6.8: Some configurations. A dashed arrow represents a pending move. Black nodes represent multiplicities.

configuration in Fig. 6.8(a) could be expressed as {1, 0, 5, 0, 1, 3} in the view from the robot r in clockwise order. Due to the multiplicity assumption, it is possible that a robot moves to a node that is occupied by other robots. As written, nodes occupied by two or more are called multiplicities (or towers). Since robots are anonymous, we can denote all of them by I in which the value of I is set to the negative of the additional number of robots located on the multiplicity (-3 indicates 3 additional robots, which means a multiplicity of 4 robots). Note that this notation allows us to represent the exact number of robots in multiplicities (called strong multiplicity detection). Robots may (or may not) have access to this information; they may only know if there is a multiplicity (i.e. a negative number). For instance, the configuration in Fig. 6.8(d) assuming that there are two robots in each multiplicity, is expressed as {2, -1, 5, -1, 2, 3}.

Ring-fix: a ring of n nodes is expressed as $\{u_0\ u_1\ .\ .\ .\ u_{n-1}\}$, with a natural number u_i. In this case each u_i stands for the number of robots located at node i. For instance, the configuration in Fig. 6.8(d) is expressed as {0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0 , 0, 1}.

Ring-location: a system of n nodes and k robots is represented in this case as a sequence of pairs of the form $\{\langle\ r_1, l_1\rangle, \ldots, \langle\ r_k, l_k\ \rangle\}$ with $r_i \in \{1, ..., k\}$ standing for each robot and $l_i \in \{1, \ldots, n\}$ standing for the node at which r_i is located. We want to note that this does not affect the fact that rings (namely nodes and edges) and robots are anonymous because the implementation of the rules for the algorithm considers that both robots and nodes are identical. The configuration in Fig. 6.8(d) could be expressed as $\{< r1, 0 >, < r2, 0 >, < r3, 6 >, < r4, 6 >, < r5, 9 >, < r6, 13 >\}$.

State Transition

MR^2-Maude predefines two actions for a robot to move to its neighbors: `->` and `<-`, which stand for moving forward and moving backward based on its current view of a configuration. We do not need to consider how to deal with pending moves or how to move in a multiplicity because they are internally handled by MR^2-Maude in a transparent way. For example, the action taken by r1 in Fig. 6.8(b) is specified as the following rule: For ring-interval:

```
rl {1, 0, 5, 0, 1, 3} => {1, 0, (5)->, 0, 1, 3} .
```

For ring-fix:

```
rl {1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0} =>
   {1, (1)->, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0} .
```

For ring-location:

```
rl {< r1, 0 >, < r2, 1 >, < r3, 7 >, < r4, 8 >, < r5, 10 >, < r6, 14 >} =>
   {< r1, 0 >, (< r2, 1 >)->, < r3, 7 >, < r4, 8 >, < r5, 10 >, < r6, 14 >} .
```

Let us emphasize the point that the action can be specified by one rule, while we need to use multiple rules when it is specified in original Maude and even in Maude RSE in which the two ring attributes are available.

6.3.2 Model Checking Facilities

MR^2-Maude predefines some useful LTL formulas as well as atomic propositions to conduct model checking experiments for the two main problems on rings: exploration and gathering problems. All what users need to do is just use the predefined formulas and/or propositions. For example, to model check the gathering property that states that once an algorithm terminates, all robots are gathered in one location, all you need to do is to conduct the following command:

```
(red modelCheck(initial, gathering) .)
```

where `gathering` is the predefined formula that defines gathering property (at the end of the algorithm all robot are gathered at one node) and `initial` is the term expressing an initial configuration of the system. Likewise, we can use the `exploration` formula to analyze the exploration property.

Although users need to learn some about LTL to understand it, the predefined formulas and propositions specific to mobile ring robot algorithms could reduce time and efforts taken and made by users.

6.4 Evaluation

Maude RSE provides a ring specification environment that allows us to declare ring data structures and MR^2-Maude, built on top of Maude RSE, allows us to specify mobile ring robot algorithms such that their specifications are closer to their mathematical descriptions and predefines some useful LTL formulas as well as atomic propositions specific to such algorithms. To demonstrate the advantages of Maude RSE, we first compare the sizes of mobile ring robot algorithm specifications in plain Maude [28, 29] and in Maude RSE and report on the overheads (which is almost nothing) introduced by Maude RSE for model checking. We then show how MR^2-Maude allows us to specify mobile ring robot algorithms.

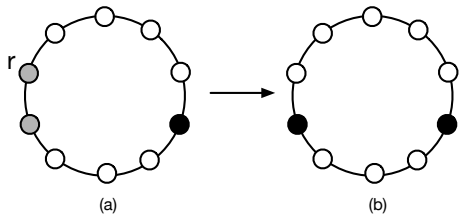


Figure 6.9: (a) A configuration with a multiplicity obtained from a symmetric configuration and (b) The configuration with a multiplicities obtained from (a).

6.4.1 Specifications with "Ring" Attributes in Maude RSE

We consider two algorithms solving two main problems on ring: the perpetual exploration algorithm, which was defined in [10] while we follow the specification in [28], and the gathering algorithm, designed in [21] and specified in [29].

A Perpetual Exploration Algorithm

In [10], its authors use the classical encoding as the sequence of occupied/free nodes in the ring. For example the configuration of Fig. 6.6(a) depicting 3 robots on a 10-node ring is encoded as (R2, F2, R1, F5) because there are 2 adjacent robots followed by 2 free nodes, followed by 1 robot, followed finally by 5 free nodes. In [28], the ring is represented as the set of all non-empty nodes.

The ring features, namely rotation and reversibility, are dealt with by using associative and commutative sets of elements. However, the commutative attribute makes it impossible to keep the order of elements in the ring. Namely, the commutative attribute forces specifiers to use complex constraints to specify the algorithms because the order of elements might change. For this reason, several functions are defined to handle these constrains.

By using Maude RSE, we do not need to handle ring characteristics. The updated specification in Maude RSE, therefore, gets rid of all these extra functions. In total, we reduce over 50% of the code.

A Gathering Algorithm

We discuss now the gathering algorithm as presented in [29], where a configuration is described as a ring-interval as described in the previous section. In [29], the authors use 44 rules to specify the behavior of the system and 53 equations to handle some constraints about configurations. Many of these rules and equations are defined to handle rings. In Maude RSE, the specification requires 17 rules and 18 equations, that is, we obtain a code reduction of more than 60%.

Performance analysis. We also conducted model checking experiments to compare the performance. 8 different systems in terms of the number of robots and the size of the ring are taken. Experiments were conducted on a 4GHz Intel Core i7 processor with 32GB of RAM. The results are shown in Table 6.1. Based on these experiments, we can conclude

Table 6.1: Performance comparison between ordinary Maude and Maude RSE.

Systems	**The number of Processes**	**The number of channels**	**The number of tokens**	**Time taken by the ordinary Maude (Second)**	**Time taken by Maude RSE (Second)**
Sys 1	3	5	1	0.171	0.163
Sys 2	3	11	3	0.270	0.240
Sys 3	5	8	2	0.546	0.723
Sys 4	5	12	2	0.927	1.247
Sys 5	8	19	1	1.999	2.181
Sys 6	8	39	2	4.116	3.929
Sys 7	9	16	2	4.823	4.912
Sys 8	9	21	1	7.130	6.812

that Maude RSE preserves the performance of the ordinary Maude environment; no extra time consuming (Fig. 6.10).

6.4.2 Specifying Mobile Ring Robot Algorithms in MR^2-Maude

We present in this section how mobile ring robot algorithms are specified in MR^2-Maude. We specify and model check the perpetual exploration algorithm [10]. By the ordinary Maude, to express a configuration, we need to store more information to control "pending move". In the other side, with MR^2-Maude, a configuration could be described as a ring of intervals. It, therefore, is suitable to use the ring-interval syntax to specify a configuration. We do not need to define any more notations to describe the system because they are already predefined. Let us consider two ways to specify the first rule RL1 as the following mathematical rule by using Maude original syntax and using MR^2-Maude syntax.

Rule RL1: $(R_2, F_2, R_1, F_z) \rightarrow (R_1, F_1, R_1, F_2, R_1, F_z)$.

By Maude original syntax, we need to handle "pending moves." We, thus, add more information to an interval in order to control pending moves as follows:

```
rl[RL1]: {< 0, nil, PSX > < 2, PY, PSY > < Z, PZ, PSZ >} =>
{< 0, [0, Z, 1], PSX > < 2, PY, PSY > < Z, PZ, add([0, Z, 1], PSZ) >} .
```

Moreover, the following rules need to be added for perform these moves.

```
crl[Pending]: {C < I1, P1, PS1 > < I2, P2, P1 ; PS2 > C'} =>
{C < I1 + 1, nil, PS1 > < I2 - 1, P2, PS2) > C'}
if  (move(P1) == 1) .
```

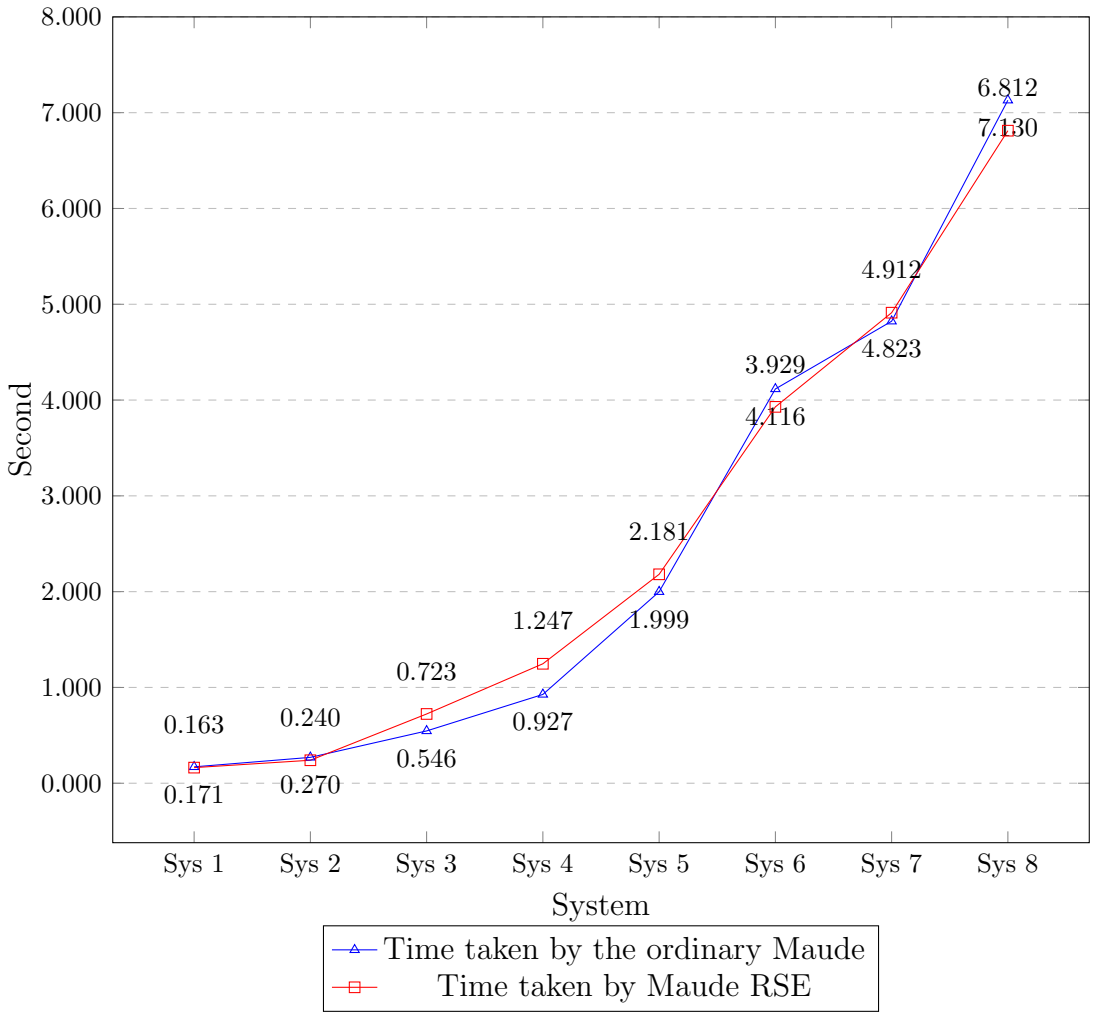


Figure 6.10: Maude RSE preserves the performance of the ordinary Maude environment.

```
crl[Pending]: {C < I1, P1, PS1 > < I2, P2, P1 ; PS2 > C'} =>
{C < I1 - 1, nil, PS1 > < I2 + 1, P2, PS2 > C'}
if (move(P1) == -1) .
```

This specification needs users to understand the Maude language to encode it. However, following the purely Mathematical definition the rule is specified in MR^2-Maude as:

```
rl [RL1] : {0, 2, Z} => {(0)<-, 2, Z} .
```

where `<-` means that the robot located at a node whose next node is occupied by one robot (denoted 0 as the interval), followed by two consecutive empty nodes (denoted `2` as the interval) and one node occupied by one robot or multiple robots (denoted `Z` as the interval), will move to the other next node (denoted `(0)<-`). All tasks to handle a pending move are done by MR^2-Maude, which is transparent to users.

Chapter 7

Conclusion and Future Work

7.1 Conclusion and Main Findings

This thesis has given an effort to demonstrate that rewriting logic and its meta-programming facility is strong enough to formalize distributed systems. We have tackled two important families of distributed systems, namely CAs and mobile robot algorithms, with rewriting logic meta-programming facilities

We have proposed (1) a technique that specifies a distributed control algorithm as a meta-program that takes a specification of an underlying distributed system and automatically generates the specification of the underlying distributed system on which the control algorithm is superimposed (UDS-CA). We have also proposed (2) a technique that makes it possible to model check if the UDS-CA generated as just mentioned enjoys desired properties that may involve not only the UDS-CA but also the underlying distributed system at the meta level. Moreover, we have proposed (3) a technique that takes the number of each kind of entities used, generate all possible initial states for the given number of each kind of entities, and conduct model checking experiments for all the initial states. Even fixing the number of each kind of entities used by an underlying distributed system, there may be more than one initial state because of, for example, the topology of the network used by the underlying distributed system. Technique (3) makes it more likely to detect a subtle flaw lurking in a control algorithm. The Chandy-Lamport Distributed Snapshot algorithm and the Cao-Singhal Mobile Checkpointing algorithm have been used to demonstrate the usefulness of technique (1). One desired property Chandy-Lamport Distributed Snapshot algorithm should enjoy is what is called the distributed snapshot reachability property that involves both an underlying distributed system and the underlying distributed system on which the algorithm is superimposed, which has been used to demonstrate the usefulness of technique (2). For one desired property the Cao-Singhal Mobile Checkpointing algorithm should enjoy, a counterexample is found for some initial states but not for the others even by fixing the numbers of mobile hosts and mobile support stations, which has been used to demonstrate the usefulness of technique (3). Liu, et al. [47] propose a technique similar to (3) but their technique does not take into account network topologies, while (3) takes into account different network topologies.

How to formalize and model check a new software system is always challenging. We show in this thesis how to formalize and model check a new form of distributed systems. We have described how to specify and model check mobile robot algorithms in Maude. In detail, we have conducted a case study in which we specify a mobile robot exploration algorithm on ring in which there are three robots and model check that the system enjoys some desired properties. Furthermore, we have proposed a formal model for mobile robot algorithms on anonymous ring shaped network under multiplicity and asynchrony assumptions - our model is general enough and could be applied to other problems. We then use an LTL model checker to model check an algorithm for robot gathering problem on ring enjoys some desired properties. We refute by model checking that the algorithm enjoys the desired properties. We detect the sources of some unforeseen design errors. We have demonstrated the usefulness of model checking techniques to formally verify mobile robot algorithms. The model checker found counterexamples showing that these analyzed algorithms are not correct. Although informal proofs have been given in [10] and [21] to guarantee the correctness of these algorithms, there remain some mistakes that are subtle and not easy to find by carefully checking the algorithm, even by experts. We want to emphasize that our goal is not to blame the authors or reviewers of the paper. When reading the paper, we also missed most of these errors. But it means that it is indeed difficult and therefore we should really consider using formal methods to check such algorithms.

Because mobile robot systems are a new form of distributed systems, the existing specification methods (and tools) do not support these systems well. In this case, a new language or an extension of an existing language is needed. This paper introduces two extensions of Maude as a specification language specific to mobile ring robot algorithms: Maude RSE and MR^2-Maude. Maude RSE makes it possible to specify ring structures, which need to be used to specify mobile ring robot algorithms and MR^2-Maude makes mobile ring robot algorithm specifications closer to their mathematical descriptions. The reason why we provide Maude RSE as well as MR^2-Maude is that the former permits Maude experts to come up with their way to formalize configurations and actions of mobile ring robot algorithms, while the latter allows even non-Maude experts to specify such algorithms in a closer way used in papers in which such algorithms are proposed.

This thesis will benefit researchers in both formal methods community and distributed computing community:

- To distributed computing community:
  - A new approach to specifying and model checking control algorithms, and
  - The tools that allow users to concisely and naturally specify mobile robot algorithms on ring shaped networks.
- To formal methods community:
  - A demonstration that meta-programs can be used as formal specifications of distributed algorithms, and

– How to build a new environment and a domain-specific language on top of an existing system (namely Maude) by using meta-programming facilities.

7.2 Future Work

Although technique (3) is useful as demonstrated, it causes another challenge because a huge number of initial states may be generated even though we use some constraints, for example, on the network topology. There are at least two possible remedies for this challenge. Initial states may be classified into a small number of groups. One representative initial state can be taken from each group, and then we can conduct model checking for each representative initial state but not for all initial states. The other possible remedy is to use a parallel (or distributed) computing environment. Model checking experiments for multiple initial states are totally independent from each other. Therefore, multiple model checking experiments can be conducted simultaneously. It is our future work to investigate the two possible remedies for a huge number of initial states. Moreover, specifying an algorithm as a meta-program is not straightforward. It requires professional skills on meta-programming. We, therefore, want to build a tool that supports specifiers to write the specification at the object level and then the specification is transformed to the meta-program at the meta level.

As future work, we consider extending Maude RSE and MR^2-Maude in the following directions: (1) to support other features on ring, such that robots located in a multiplicity may have different calculated moves or rings are dynamic and (2) to support other topologies, such as grid. For (1), the algorithm performed by robots should be randomized. Moreover, we consider to apply similar techniques to verify other algorithms that have been proposed in the literature. It would be interesting to specify and model check algorithms designed for other topologies (*e.g.* grid, torus, or arbitrary graphs) or working under other assumptions (such as (semi-)synchronous scheduler, or different notion of fairness). Finally, one of the biggest challenge would be to investigate continuous topologies.

Although our model checking approach successfully detects counterexamples, it cannot guarantee that distributed algorithms surely enjoy desired properties because all possible cases may not be covered. This is also challenging for existing current model checking techniques. Some equational abstraction techniques have been introduced to deal with this challenge. We would want to consider how to tackle control algorithms and mobile robot algorithms with those techniques. However, the specifications of control algorithms (or mobile robot algorithms) contain huge and very complex sets of equations, it is not straightforward to achieve the set of equations E' such that an equational abstraction satisfies many requirements, such as finite, terminating and coherent. Furthermore, the current equational abstraction techniques require human being sense to add these equations. We therefore would like to apply meta-programming techniques to automatically generate the abstraction of an input a rewrite theory describing a distributed system and specified as a module in Maude.

Bibliography

[1] B. Aminof, A. Murano, S. Rubin, and F. Zuleger. Verification of asynchronous mobile robots in partially known environments. In Q. Chen, P. Torroni, S. Villata, J. Hsu, and A. Omicini, editors, *Principles and Practice of Multi-Agent Systems. PRIMA*, volume 9387 of *Lecture Notes in Computer Science*. Springer, Cham, 2015.

[2] K. Bae and J. Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 99:193–234, 2015.

[3] Kyungmin Bae and José Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 99:193–234, 2014.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] T. Balabonski, A. Delga, L. Rieg, S. Tixeuil, and X. Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 10083 of *LNCS*, pages 7–19. Springer, 2016.

[6] Jiří Barnat, Luboš Brim, Milan Češka, and Petr Ročkai. Divine: Parallel distributed model checker. In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology*. IEEE, 2010.

[7] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[8] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu A.J. and Vardi M.Y, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, CAV 1998.

[9] B. Bérard, P. Lafourcade, L. Millet, M. Potop-Butucaru, Y. Thierry-Mieg, and S. Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.

[10] L. Blin, A. Milani, M. Potop-Butucaru, and S. Tixeuil. Exclusive perpetual ring exploration without chirality. In *Proceedings of the 24th International Symposium on Distributed Computing*, volume 6343 of *LNCS*, pages 312–327. Springer, 2010.

[11] F. Bonnet, M. Potop-Butucaru, and S. Tixeuil. Asynchronous gathering in rings with 4 robots. In *Proceedings of the 15th International Conference on Ad-hoc, Mobile, and Wireless Networks*, volume 9724 of *LNCS*, pages 311–324. Springer, 2016.

[12] M. Bruno Andriamiarina, D. Méry, and N. Kumar Singh. Revisiting snapshot algorithms by refinement-based techniques. *Computer Science and Information Systems*, 11(1):251 – 270, 2014.

[13] Guohong Cao and Mukesh Singhal. Mutable checkpoints: A new checkpointing approach for mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):157–172, February 2001.

[14] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Brinksma E. and Larsen K.G., editors, *Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, Berlin, Heidelberg, 2002.

[15] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, (16):1512–1542, 1994.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[17] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[18] et al. Corbett, J.C. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264. USENIX Association, Berkeley, 2012.

[19] P Courtieu, L. Rieg, S Tixeuil, and X. Urbain. Certified universal gathering in r^2 for oblivious mobile robots. In *Proceedings of the 30th International Symposium on Distributed Computing*, volume 9888 of *LNCS*, pages 187–200. Springer, 2016.

[20] Joseph K. Cross and Douglas C. Schmidt. Meta-programming techniques for distributed real-time and embedded systems. In *Proceedings of 7th IEEE international workshop on object-oriented real-time dependable systens*, pages 3–10, 2002.

[21] G. D'Angelo, G. Di Stefano, and A. Navarra. Gathering on rings under the look–compute–move model. *Distributed Computing*, 27:255–285, March 2014.

[22] G. D'Angelo, G. Di Stefano, A. Navarra, N. Nisse, and K. Suchan. Computing on rings by oblivious robots: A unified approach for different tasks. *Algorithmica*, 72(4):1055–1096, 2015.

[23] G. D'Angelo, A. Navarra, and N. Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 28(1):17–48, 2017.

[24] A. K. Datta, A. Lamani, L. L. Larmore, and F. Petit. Enabling ring exploration with myopic oblivious robots. In Hyderabad, editor, *IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 490–499, 2015.

[25] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.

[26] Steṕhane Devismes. Optimal exploration of small rings. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security, WRAS 10*, pages 91–96, 2010.

[27] Steṕhane Devismes, Franck Petit, and Seb́astien Tixeuil. Optimal probabilistic ring exploration by semi-synchronous oblivious robots. In *16th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2009)*, pages 203–217. LNCS, Springer, 2009.

[28] H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of a mobile robots perpetual exploration algorithm. In *Proceedings of the 6th International Workshop on Structured Object-Oriented Formal Language and Method (SOFL+MSVL 2016)*, volume 10189 of *LNCS*, pages 201–219. Springer, 2016.

[29] H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of robot gathering. In *Proceedings of The 21th Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017.

[30] H. T. T. Doan, F. Bonnet, and K. Ogata. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In *Proceedings of The 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, pages 1586–1596, 2017.

[31] H. T. T. Doan, W. Zhang, M. Zhang, and K. Ogata. Model checking chandy-lamport distributed snapshot algorithm revisited. In *Proceedings of the 2nd International Symposium on Dependable Computing and Internet of Things*, pages 7–19, 2015.

[32] Orna Grumberg E. M. Clarke and Doron Peled. *Model Checking*. MIT Press, 1999.

[33] P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

[34] P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool Publishers, 2012.

[35] J. Grov and P.C. Olveczky. Increasing consistency in multi-site data stores: Megastore-cgc and its formal analysis. In *SEFM*, volume 8702 of *LNCS*. Springer, 2014.

[36] Olveczky P.C. Grov J. Formal modeling and analysis of google's megastore in real-time maude. In Ogata K. Iida S., Meseguer J., editor, *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 494–519. Springer, Berlin, Heidelberg, 2014.

[37] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[38] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[39] Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Mobile robots gathering algorithm with local weak multiplicity in rings. In *SIROCCO*, pages 101–113, 2010.

[40] A. Kawamura and Y. Kobayashi. Fence patrolling by mobile agents with distinct speeds. *Distributed Computing*, 28(2):147–154, 2015.

[41] Ralf Klasing, Euripides Markou, and Andrzej Pelc. Gathering asynchronous oblivious mobile robots in a ring. *Theor. Comput. Sci.*, 390(1):27–39, 2008.

[42] Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In Baldan P. and Gorla D., editors, *Concurrency Theory (CONCUR 2014)*, volume 8704 of *Lecture Notes in Computer Science*, pages 125–140. Springer, Berlin, Heidelberg, 2014.

[43] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[44] Anissa Lamani, M. Potop-Butucaru, and S. Tixeuil. Optimal deterministic ring exploration with oblivious asynchronous robots. In *17th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 183–196, 2010.

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

[46] Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering*, 40(5):483 – 501, 2014.

[47] S. Liu, P. Olveczky, Q. Wang, and J. Meseguer. Formal modeling and analysis of the walter transactional data store. report available at https://sites.google.com/site/siliunobi/walter.

[48] S. Liu, P. Olveczky, Q. Wang, and J. Meseguer. Formal modeling and analysis of the walter transactional data store. In *WRLA*. To appear, 2018.

[49] S. Liu, P.C. Olveczky, K. Santhanam, Q. Wang, I. Gupta, and J. Meseguer. Rola: A new distributed transaction protocol and its formal analysis. In To appear, editor, *FASE*, 2018.

[50] K. Mani Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed system. *ACM TOCS 3*, 3/1:63–75, 1985.

[51] K. Mani Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed system. *ACM TOCS 3*, 3/1:63–75, 1985.

[52] Narciso Martí-Oliet, Francisco Durán, and Alberto Verdejo. Equational abstractions in rewriting logic and maude. In *Formal Methods: Foundations and Applications, 17th Brazilian Symposium, SBMF*, number 8941 in LNCS, pages 17–31, 2015.

[53] José Meseguer. Taming distributed system complexity through formal patterns. *Science of Computer Programming*, 83:3–34, 2014.

[54] José Meseguera, MiguelPalominob, and Narciso Martí-Olietb. Equational abstractions. *Theoretical Computer Science Theoretical Computer Science*, 403(2-3):239–264, 2008.

[55] Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *In the proceeding of 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014)*, volume 8756, pages 237–251. LNCS, 2014.

[56] J. Oetsch, J. Puhrer, and H. Tompits. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10:513–529, 2010.

[57] K. Ogata and Thi Thanh P. Huyen. Specification and model checking of the chandy and lamport distributed snapshot algorithm in rewriting logic. In *Lecture Notes in Computer Science*, volume 7635 of *LNCS*, pages 87–102. 14th ICFEM, 2012.

[58] P.C. Olveczky and J. Meseguer. Semantics and pragmatics of real-time maude. In *Higher-Order and Symbolic Computation*, volume 20, pages 161–196, 2007.

[59] Evangelos Kranakis Paola Flocchini, Danny Krizanc, Nicola Santoro, and Cindy Sawchuk. Multiple mobile agent rendezvous in a ring. In *LATIN*, pages 599–608, 2004.

[60] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[61] Liu S., Olveczky P.C., Ganhotra J., Gupta I., and Meseguer J. Exploring design alternatives for ramp transactions through statistical model checking. In *Formal Methods and Software Engineering. ICFEM*, volume 10610 of *Lecture Notes in Computer Science*, pages 298–314. Springer, Cham, 2017.

[62] Liu S., Olveczky P.C., Ganhotra J., and Gupta I.and Meseguer J. Formal modeling and analysis of ramp transaction systems. In *The 31st Annual ACM Symposium on Applied Computing*, pages 1700–1707, 2016.

[63] A. Sangnier, N. Sznajder, M. Potop-Butucaru, and Tixeuil S. Parameterized verification of algorithms for oblivious robots on a ring. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 212–219, 2017.

[64] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

[65] Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23:341–358, 2010.

[66] Wen Zeng, Maciej Koutny, Paul Watson, and Vasileios Germanos. Formal verification of secure information flow in cloud computing. *Journal of Information Security and Applications*, 27-28:103–116, 2016.

Publications

[1] H. T. T. Doan, W. Zhang, M. Zhang, and K. Ogata. **Model checking Chandy-Lamport distributed snapshot algorithm revisited**, In Proceedings of the 2^{nd} IEEE International Symposium on Dependable Computing and Internet of Things, pages 7-19, 2015.

[2] H. T. T. Doan, F. Bonnet, and K. Ogata. **Model checking of a mobile robots perpetual exploration algorithm**. In Proceedings of the 6^{th} International Workshop on Structured Object- Oriented Formal Language and Method (SOFL+MSVL 2016), volume 10189 of LNCS, pages 201-219. Springer, 2017.

[3] H. T. T. Doan, F. Bonnet, and K. Ogata. **Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level**. In Proceedings of The 37^{th} IEEE International Conference on Distributed Computing Systems (ICDCS 2017), pages 1586-1596, 2017.

[4] H. T. T. Doan, F. Bonnet, and K. Ogata. **Model Checking of Robot Gathering**, In Proceedings of The 21th Conference on Principles of Distributed Systems (OPODIS 2017), volume 95 of LIPIcs, pages 12:1-12:16, 2018.