SELF-STABILIZING TRANSFORMATIONS BETWEEN MESSAGE PASSING
AND SHARED MEMORY MODELS

by

Kleoni Ioannidou

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2001 by Kleoni Ioannidou
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-62964-3
Abstract

Self-stabilizing transformations between message passing and shared memory models

Kleoni Ioannidou

Master of Science

Graduate Department of Computer Science

University of Toronto

2001

We give a new formal definition of a transformation of self-stabilizing algorithms from one distributed model to another and propose a criterion for proving correctness of such transformations. Our method is more natural than the already existing simulation techniques. We give a new transformation of self-stabilizing algorithms from the synchronous and asynchronous message passing models to the asynchronous shared memory model. To do this, we implement a self-stabilizing algorithm that solves mutual exclusion in the shared memory model using read/write registers. This algorithm has improved time complexity compared with existing mutual exclusion algorithms. We also modify an existing transformation of self-stabilizing algorithms from the shared memory model to a message-driven message passing model with time-outs so that it transforms self-stabilizing algorithms from the shared memory to another message passing model that we define without using time-outs.
Acknowledgements

I want to thank my parents for their support and caring all these years. My office-mates Tsuyoshi Morioka, Periklis Andritsos, Steve Myers, and Eric Joanis for being very good friends and for creating an excellent working environment. Alan Skelley for his support and for making my life beautiful. I am thankful to my second reader Vassos Hadzilacos for his useful comments. Especially I want to thank Faith Fich for being better that what I was hoping for in a supervisor. Her help and contributions to my education have been invaluable. Her advice was always right on target, she helped me expand my knowledge, and motivated me to work in this area. Not only that, but she is a great person as well. I always felt nicely to talk to her about anything that arose and her support was amazing.
# Contents

1 Introduction .......................... 1

2 Definitions ........................................ 4
   2.1 Distributed Systems ........................................ 4
   2.2 Models ........................................ 10
      2.2.1 Shared Memory Models ........................................ 11
      2.2.2 Message Passing Models ........................................ 12
   2.3 Self-Stabilization ........................................ 13
      2.3.1 Complexity Measures for Self-Stabilizing Algorithms ........................................ 16
      2.3.2 A Technique for Constructing Self-Stabilizing Algorithms ........................................ 17
   2.4 Transformations of Self-Stabilizing Algorithms from One Model to Another ........................................ 19
      2.4.1 Correctness of Transformation of Self-Stabilizing Algorithms ........................................ 19
      2.4.2 Simulation of States ........................................ 20
      2.4.3 Simulation of Operations ........................................ 22
      2.4.4 Simulation of Executions ........................................ 24
      2.4.5 Correctness of Transformations Using Simulation ........................................ 27
   2.5 Other Transformation Techniques ........................................ 36
2.6 Message Passing Models for Self-Stabilization ..................................... 41

3 Related Work .................................................................................. 43

3.1 General Transformations between the Message Passing Model and the
Shared Memory Model ........................................................................ 44
3.2 Transformations of General Algorithms to Self-Stabilizing Algorithms .. 51
3.3 Transformations of Self-Stabilizing Algorithms from the Shared Memory
Model to the Message-Driven Model .................................................. 54
3.4 Self-stabilizing Token-Passing Algorithms for Message Passing Systems
with Two Processors .......................................................................... 56
3.5 Self-Stabilizing Token-Passing Algorithms in Shared Memory .......... 59
3.6 Lower Bounds for Self-Stabilization .............................................. 60

4 Transformations ............................................................................. 62

4.1 Round-Robin Algorithm .................................................................. 63
  4.1.1 Binary Tree Structure ................................................................. 63
  4.1.2 Round-Robin Algorithm ............................................................. 64
  4.1.3 Properties of the Round-Robin Algorithm .................................. 65
  4.1.4 Complexity of the Round-Robin Algorithm ............................. 72

4.2 Transforming Self-Stabilizing Algorithms from the Message Passing Model
to the Shared Memory Model ............................................................ 74
  4.2.1 The Composed Algorithm .......................................................... 76
  4.2.2 Simulation of states ................................................................... 78
  4.2.3 Simulation of Operations ............................................................ 79
4.2.4 Correctness of Transformation ........................................ 81
4.2.5 Complexity of the Transformed Algorithm .......................... 83
4.3 Transforming Self-Stabilizing Algorithms from Synchronous Message Passing to Asynchronous Shared Memory ....................... 85
4.4 Transforming Self-Stabilizing Algorithms from the Shared Memory Model to the Message Passing Model ........................................ 86

5 Discussion and Future Work ............................................. 91
5.1 Extensions of our Work ..................................................... 92
5.2 Future Work ................................................................... 94

Bibliography ........................................................................ 95

Index ................................................................................... 99
List of Figures

2.1 Example ....................................................... 24
2.2 Definition of execution $\hat{b}$ .................................. 29
2.3 Definition of superrange and range .......................... 34
2.4 Example ....................................................... 40
4.1 Round-Robin algorithm for processor $P_i$ using Read/Write registers ........................ 88
4.2 State Diagram of Round-Robin algorithm .................. 89
4.3 Simulation Cycle ............................................. 90
5.1 An example of the new transformation from message passing to shared memory ......................... 94
Chapter 1

Introduction

A distributed system is a collection of computing devices that communicate with each other. For example, communication is needed between processors that share resources, such as printers, or data. Depending on the way the processors communicate, we define different distributed models that belong to one of the following two classes of models: message passing models and shared memory models. Because of the different characteristics of a shared memory model and a message passing model, we need to design two different solutions, one for each model. This can be avoided if we can design a general method of transforming algorithms from one model to another.

A transformation of algorithms from one model to another takes as input an algorithm of the first model, and produces as output an algorithm designed to solve the same task in the second model. If we have transformations of algorithms between two models, we can use them to get algorithms for both models, although we only design an algorithm for one of them. Studying ways to transform algorithms between these models helps us to understand the models better and to compare them. If we establish a transformation
from model $\mathcal{M}$ to model $\mathcal{M}'$, then $\mathcal{M}'$ is at least as powerful as $\mathcal{M}$.

There are distributed systems that allow processor faults, for example processor crashes, or faults of the communication medium. In this thesis, we will restrict our attention to the types of failures caused by temporary processor or communication medium malfunctions. We call these failures transient faults. A transient fault may change the information stored in processors and/or the information contained in the communication medium of the system but not the algorithm itself (note that the algorithm is also information stored in the local memory of the processors). We want algorithms that eventually make the system regain normal behavior. Such algorithms are called self-stabilizing algorithms and were first studied by Dijkstra [12], and then by Lamport [18].

In this thesis, we give a new formal definition of transformation of self-stabilizing algorithms between distributed models and propose a technique for proving correctness of such transformations. Our definition of transformation is consistent with the definition proposed by Lynch [20], who was not concerned with self-stabilization. We give a new transformation of self-stabilizing algorithms from the message passing model to the shared memory model. To do this, we implement a self-stabilizing algorithm that solves a special case of mutual exclusion in the shared memory model using only read/write registers. The algorithm ensures that eventually processors enter the critical section in round-robin fashion in the order of their identifiers. Our algorithm has better time complexity than existing algorithms for mutual exclusion for the shared memory model, and hence, the transformed algorithms have smaller overhead. We also modify an existing transformation of self-stabilizing algorithms from the shared memory model to a message
passing model so that it transforms self-stabilizing algorithms from the shared memory to another message passing model.

We give a brief overview of this thesis. In Chapter 2, we present definitions related to distributed systems, distributed models, self-stabilization, and we present formal definitions of transformations of self-stabilizing algorithms from one model to another. Next, in Chapter 3, we present related work, and in Chapter 4, we give a self-stabilizing transformation from the message passing model to the shared memory model. We conclude with a chapter containing some extensions of our transformation and proposed future work.
Chapter 2

Definitions

In this chapter, we define distributed systems, the notion of a scheduler, and the execution of a distributed algorithm. Then, we give a formal definition of self-stabilization, a complexity measure used for self-stabilizing algorithms, and the composition technique used for constructing self-stabilizing algorithms. Finally, we give a formal definition of transformation of self-stabilizing algorithms from one model to another and discuss the differences with existing definitions that are not designed for self-stabilizing algorithms. Some of the definitions we present are from the books of Dolev [14] and Lynch [20]. Other definitions have been modified either because we disagreed with the existing characterization of some notions, or because they did not fit our context.

2.1 Distributed Systems

In this section, we describe distributed systems and we discuss the main parameters that characterize a model of a distributed system.
Chapter 2. Definitions

A distributed system is a set of processors that communicate with one another. The state of a processor specifies the information contained in its local memory and its program counter. The state of the communication medium specifies the information contained in the communication medium. Examples of information contained in the communication medium are the sequences of messages in each communication link and values of shared objects. The state of a distributed system consists of the states of the processors and the state of the communication medium. When the context is clear, we use "state" as an abbreviation of "state of the distributed system".

A step is the smallest possible action performed by a processor that changes the state of the system. Each distributed system provides a set of steps that the processors are able to perform, enabling them to communicate and change the state of the system. A step can be internal or external. A processor that executes an internal step uses and changes only variables stored in its local memory. External steps are divided into input and output steps. A processor that performs an input step copies information from the communication medium into its local memory and updates its program counter. This may or may not change the state of the communication medium. A processor that performs an output step uses its local information to possibly change the state of the communication medium, and does not change the values of its local variables apart from its program counter.

An operation is a sequence of steps performed (atomically) by a processor. This means that, for operations that consist of more than one step, we do not examine the states created in the middle of operations. An internal operation is a sequence of internal steps performed by the same processor. An external operation will contain at
least one external step and 0 or more internal steps. Special cases of external operations are the input and output operations. An input operation consists of a sequence of at least one input step followed by 0 or more internal steps. An output operation consists of a sequence of 0 or more internal steps followed by at least one output step.

The processors of a distributed system interact with users via some user interface, using operations that we call interface operations. For example, if we design an algorithm that solves the mutual exclusion problem, then a processor that enters the critical section informs the corresponding user about that. The interface operations behave like internal operations, because the user interface is accessed by each processor separately, is never used for purposes of communication between processors, and does not affect the communication medium. There are two different kinds of interface operations: the input interface operations and the output interface operations. An input interface operation is triggered by the users and is applicable to any state of the system. An output interface operation is triggered by the distributed system and provides information to the users.

An operation by a processor is applicable to a state if the processor can perform the operation starting in this state. A sequence of operations is applicable to a state if the following hold. The first operation of the sequence is applicable to this state and, if we remove the first operation, the resulting sequence of operations is applicable to the state reached after the first operation is performed.

The part of the communication medium that is accessed directly from a processor is either the part that may change after the processor performs some operation, or the part that provides information to the processor during an operation. The environment of
a processor consists of the local memory of the processor and the part of the communication medium accessed directly by the processor. The state of the environment of a processor consists of the state of the processor and the state of the part of communication medium which is accessed directly by the processor. We use \textit{env}(s, i)$ to denote the state of the environment of the processor with identifier $i$ when the system is in state $s$.

We say that two processors are \textbf{neighbouring} if their environments have some common part. We assume that each processor knows the identifiers of its neighbors. By $\mathcal{NBR}_i$ we denote the array of the identifiers of the neighbouring processors of $P_i$ arranged in increasing order. We use $d_i$ to denote the degree of $P_i$, which is the number of the elements in $\mathcal{NBR}_i$.

Each processor performs operations in an order specified by a program. There might be restrictions about when processors can perform operations. For example, all the processors of a system may perform operations simultaneously or perhaps only one processor at a time can perform an operation. The order in which the processors are chosen to perform operations is determined by a scheduler. A \textbf{scheduler} repeatedly selects a set of processors and activates the processors in the set to perform operations. Recall that we define operation to be \textbf{atomic} which means that the steps of the operation are performed without interruption by other operations or steps in any execution of the operation.

In a \textbf{synchronous system}, all processors are selected by the scheduler and activated simultaneously to perform an operation, although not necessarily the same operation. We call this a \textbf{synchronous round}. In such a system, the processors perform operations at the same rate. An \textbf{execution of a synchronous system} is an alternating sequence of states and synchronous rounds, $s_0, t_1, s_1, t_2, \ldots$ such that, for all $i \geq 1$, every state $s_i$
is reached from state $s_{i-1}$ after the processors perform the synchronous round $t_i$. The triple $(s_{i-1}, t_i, s_i)$ is called a transition.

In an asynchronous system, there are no bounds on the relative processor speeds. A special case of an asynchronous system is the one with an interleaving scheduler, which repeatedly selects one processor at a time to activate. A special case of an interleaving scheduler is the Round-Robin scheduler which activates the processors in a Round-Robin fashion: If there are $n$ processors with identifiers $1, \ldots, n$, it activates processor with identifier $(i \mod n) + 1$ after activating processor with identifier $i$. We define a Round-Robin cycle to be a period of time during which every processor performs exactly one operation.

In our asynchronous models, we will restrict our attention to interleaving schedulers. In other words, we assume that even if the processors may perform operations concurrently, there is always a linearization of these operations that leads to the same state of the system as the state reached by performing them concurrently. An execution of an asynchronous system is an alternating sequence of states and operations. Starting from a state in the execution, the next operation performed is determined by a scheduler selecting a processor, and the program running on that processor determines the operation to be executed next. This can be described as a transition $(s, t, s')$, by a processor $P$, where $s$ and $s'$ are states of the system and $t$ is an operation performed by $P$ when the system is in state $s$. After $t$ is performed by $P$, the system is in state $s'$.

The internal or interface operations performed by some processor $P$ will be noticed by other processors only after $P$ performs the next external operation. As a result, it does not matter (for the rest of the execution) exactly when a processor $P$ executes
internal or interface operations between two consecutive external operations performed by $P$. Consider an interface operation $op$ performed by some processor $P$ at a particular time in some execution. Then, we define the range of $op$ to be the fragment of the execution with the following properties: This fragment starts with the preceding external operation performed by $P$ (excluding this operation) and ends with the following external operation performed by $P$ (excluding this operation).

The set of possible external and internal operations depends on the algorithm that runs in the distributed system, but the set of possible interface operations does not. The set of possible interface operations is part of the problem specification. The interface operations of an execution describe the progress of the system trying to solve a specific problem. For that reason, it is useful to define the trace of an execution to be the sequence of interface operations performed in the execution. If $a$ is an execution, we use the notation $\text{trace}(a)$ to denote its trace. We also define trace of an execution to be the sequence of interface operations performed by the processor with identifier $i$ in the execution.

Dolev, Israeli and Moran [13], define a task to be the set of safe sequences of states, where a safe sequence of states is a sequence of states in which the desired properties of an algorithm hold. The notion of task should be independent of the system, and the algorithm designed to solve a problem in a system. It should only depend on the problem specification. This is not true in the definition above because the state of a system depends on the system. Furthermore, the set of safe sequences of states may describe, in part, how the algorithm solves the problem, which is not what we want for a specification. Instead, we define the task of a problem to be a set of traces that describes
the problem specifications. For example, a trace for the mutual exclusion problem satisfies the following: if some processor $P$ enters the critical section then no other processor can enter it before $P$ releases it. Similar definitions of the notion of the task are given by Lynch [20, page 216] under the name of "trace property", and by Attiya and Welch [7, page 161] under the name of "problem specification".

Dolev, Israeli, and Moran [15] define the **weak exclusion class** of tasks, that contains a number of problems commonly used in self-stabilization. A task in the weak exclusion class satisfies the following. In any execution of an algorithm that solves this task, there exist two operations performed by two different processors during the execution that cannot be performed concurrently. For example, consider the token-passing problem. In an execution of an algorithm that solves this problem, the operations performed by $P_i$ and $P_j$ that make them get the token cannot happen concurrently. Note that mutual exclusion is not included in this task because a valid execution of an algorithm for this problem may have only one processor ever requesting the resource.

### 2.2 Models

We present our algorithms in a similar way as in [20]. We use $pc_i$ to denote the program counter of a processor with identifier $i$. In our description of algorithms, the program counter takes values that specify the operation that is applicable to a state. Specifically, an operation will be performed by a processor when it is allocated a step by the scheduler, if the program counter of that processor has a value that specifies that operation.

We will now describe in detail the specific models of distributed systems that we
study in this thesis. In the first class of models, processors communicate using shared objects. In the second class of models, the communication medium consists of links to which processors pass messages. In both classes, the models vary also depending on the atomicity conditions assumed in the definitions of operations.

2.2.1 Shared Memory Models

In the shared memory model, processors communicate using shared objects. There are many different types of shared objects. In this thesis, we will use registers that can be read and written by a subset of processors. A processor that performs a read step to a register stores the value of the register to its local memory and returns this value. A processor that performs a write step of a value \( v \) to a register updates the contents of the register to \( v \) and returns ack. We may have a single-writer register that can be written by exactly one processor or a multi-writer register that can be written by more than one processor. Similarly, we define a single-reader register as a register that can be read by exactly one processor and a multi-reader register that can be read by more than one processor. We will use both multi-reader, multi-writer registers, and single-reader, single-writer registers in this thesis.

Another parameter of the models is which sequences of steps we choose to be (atomic) operations. We are interested in read/write registers that provide separate atomic read operations and atomic write operations. An example of a more powerful type of object is a read-modify-write object, which allows read, modify and write steps to be performed as one operation. We will not use this type of object. Another example is
the snapshot object. This object has a component for each of the processors that is writable exclusively by that processor. All the components together can be read by any processor in one atomic operation that is called scan.

A different shared memory model is used by Dijkstra in his seminal work on self-stabilization. We call this the composite atomicity model. In this model, exactly one processor performs an operation at a time. In one atomic operation, a processor reads the states of all its neighbours and changes its state. Note, that in the previously defined shared memory models, in order for the state of some processor to become available to the rest, it has to write it to some shared object. That means that it is possible for a processor to update its local memory but not the shared object where this information is stored. Instead, in the composite atomicity model, a processor is able to read the local memory of its neighbours directly. Note that the neighbours are determined by a graph connecting the processors and that the edges of this graph do not contain any information.

2.2.2 Message Passing Models

In this model, a link that connects processor \( P \) to processor \( P' \) contains messages that are sent from \( P \) and have not yet been received by \( P' \). For every pair of processors that communicate directly, there are two links: one for each direction of communication. Each link is modeled by a queue that may contain an unbounded number of messages.

Every processor can execute two kinds of external steps: send and receive. A processor \( P \) that sends a message stored in its local memory to processor \( P' \) adds the
message to the end of the queue for the directed link from \( P \) to \( P' \). A processor \( P \) that performs a receive step from processor \( P' \) removes the first message from the queue for the directed link from \( P' \) to \( P \) and copies it to its local memory. If the queue is empty, so there is no message to be received, the operation will return a special symbol, \( \perp \). The receive step is an input step and the send step is an output step.

There are many ways to define an atomic operation in this class of models. We consider models where an operation consists of a collection of receive steps, one from each processor in a subset of 0 or more of its neighbors, followed by 0 or more internal steps, followed by a collection of send steps, one to each of a subset of 0 or more of its neighbors, with the restriction that each operation contains at least one step. For example, consider a model whose operations consist of either a send step, or a receive step.

We present an alternative definition for the message passing model, the message-driven model, that is used in many of the results in the literature. The send step is the same as above, but the receive step can only be performed by a processor when there is a message in the receiving queue of this processor. In this model, any operation begins with a single receive step, followed by 0 or more internal and send steps.

### 2.3 Self-Stabilization

In a **fair execution** each processor performs operations infinitely often. A **legal execution** with respect to some task is an execution whose trace is in the task. A state is **safe** with respect to a task, if every fair execution of the algorithm that starts from
that state is a legal execution. An algorithm is **self-stabilizing** for the task if every fair execution of the algorithm eventually reaches a safe state. We note that, after a safe state is reached, all the states reached from it are safe, in any execution of the algorithm if no further transient faults occur.

We consider infinite executions, where some transient faults change the initial state and then no more transient faults happen. Even if transient faults occur infinitely often, which may happen in practice, a self-stabilizing algorithm is still useful, if the states of the execution are safe for large proportions of the execution.

We will discuss the effects of transient faults in the two models that we will use. In the message passing model, it is possible that messages are corrupted over links, and hence the link contents may be arbitrary after a transient fault. The only restriction that we impose on the link failures is to disallow an infinite number of messages to be created as a result of a fault because, otherwise, communication between processors would be prevented. In the shared memory model a transient fault may change the contents of the shared objects. In both models, a transient fault may change the contents of the local memory of the processors but not the algorithm executed by the processors (which, in practice, is also stored in their local memory).

There is another definition of fairness in the literature. In their books, Lynch [20] and Attiya and Welch [7] define an execution of a non-self-stabilizing algorithm to be fair if every internal and external operation that is continuously applicable eventually occurs. In particular, if \( a \) is an infinite fair execution, then it contains either infinitely many occurrences of operation \( op \), or infinitely many occurrences of states in which \( op \) is not applicable. This definition is given for self-stabilizing algorithms in Dolev's book [14].
Both these definitions for fairness of executions appear in the self-stabilizing literature even in articles by the same authors. For example, Dolev, Israeli, and Moran use the first definition in [13], and the second definition in [15].

Next, we discuss the differences of the two definitions. The second definition of fairness is used for distributed systems where processors may crash (i.e. perform a "crash" operation after which no operation is applicable). Specifically, even if a processor crashes during an execution, this execution may still be fair under the second definition, but not under the first. In self-stabilization, we do not need a definition that considers possible processor crashes because we allow only transient faults.

The second definition also applies to non-deterministic systems where more than one operation is applicable to a state of a processor. In particular, if a state of a processor is reached infinitely many times, the fairness condition ensures that each of the operations applicable to that processor state will be executed by the processor infinitely many times during the execution. This definition is useful if we consider algorithms that stabilize to two different tasks in parallel. For example, consider that in some state with two operations applicable by some processor, where one of them contributes to the stabilization of one task and the other to the stabilization of the other task. Then, the second definition of fairness ensures that both operations will be fairly executed, and as a result, the algorithm will stabilize to both tasks. We chose not to use this definition because we construct deterministic self-stabilizing algorithms, where exactly one operation is applicable to every processor state.
2.3.1 Complexity Measures for Self-Stabilizing Algorithms

For time complexity, we define the stabilization time as follows. In synchronous systems, we measure the maximum number of synchronous rounds needed for the algorithm to stabilize starting from any arbitrary state. In asynchronous systems, we use a similar definition. The only difference is that, instead of counting the synchronous rounds, we count asynchronous rounds. These are defined as follows: The first asynchronous round in an execution is the shortest prefix of the execution during which every processor executes at least one external operation. The $k^{th}$ round of an execution is the first round of the suffix of the execution after we remove the first $k - 1$ rounds.

Another complexity measure is work, which is the total number of external operations performed. This would be problematic for self-stabilization in an asynchronous system: Stabilization requires all processors to execute some steps. Work is affected by the individual progress of the processors since we can schedule one processor to perform any number of operations before the other processors perform one operation. In this case work will fail to provide a measure of the progress of the whole system.

There are many alternatives for the space complexity of a self-stabilizing algorithm, depending on the model used. For any distributed model, we need to know the number of bits used locally by the processors. For the shared memory model, we also count the number of different shared objects used by the algorithm and the number of bits used by each of these objects. For the message passing model, we may count the size of the messages sent, and/or have a bound on the number of messages on the links. (Note that this bound may depend on the number of messages in the initial state.)
The message complexity of an algorithm in the message passing model is the maximum number of messages sent in any execution of this algorithm, until it reaches a safe state.

2.3.2 A Technique for Constructing Self-Stabilizing Algorithms

We describe the fair algorithm composition technique based on the descriptions presented by Dolev, Israeli and Moran [14, 13]. This is one of the most widely used techniques for constructing self-stabilizing algorithms in shared memory models. We will use this technique in Chapter 4 for the transformation of self-stabilizing algorithms from the message passing model to the shared memory model. Although we present it for two algorithms, it can easily be generalized to more than two.

Suppose that we want to construct an algorithm that stabilizes to some task $T_2$. We will demonstrate how the composition technique is used to do this. Let $A_1$ be an algorithm that stabilizes to some other task $T_1$. Assume that we can construct an algorithm $A_2$ that stabilizes to $T_2$ under some conditions that are satisfied if the states reached in an execution of $A_2$ are safe for $T_1$. Now, consider an execution where each processor executes operations of $A_1$ and operations of $A_2$ infinitely often. In this execution, after $A_1$ stabilizes to task $T_1$, the conditions that enable $A_2$ to stabilize to $T_2$ will be satisfied. Then, $A_2$ will stabilize to $T_2$. Since the processors in this execution execute both algorithms, the system will eventually stabilize to both tasks $T_1$ and $T_2$. The algorithm $A$ whose executions are created by interleaving the executions of $A_1$ and $A_2$ is called the fair composition of the algorithms $A_1$ and $A_2$. 
We give an application of the fair composition technique. Consider an algorithm that is designed for systems where every processor knows the set of its neighbours and the graph that describes the neighborhood relation does not change. If dynamic changes of the system are allowed, then we could compose our algorithms with a self-stabilizing algorithm where each processor calculates its neighbourhood set.

Now, we describe the composition technique in more detail. We denote by $C_1$, $C_2$, and $C$ the sets that contain the values of the program counters of the algorithms $A_1$, $A_2$, and $A$, respectively. We can define $C = C_1 \times C_2$, and use the program counter of the composed algorithm to show which operation of $A_1$ and which operation of $A_2$ are applicable next.

Let $S_1$ be the set of states of the communication medium and the processors when they execute $A_1$, excluding the values of their program counters. Then, $S_1 \times C_1^n$ is the set of system states of algorithm $A_1$. Let $S_2$ be the set of values of the part of the state of the communication medium and the processors that is modified by the processors when they execute $A_2$, excluding the values of their program counters. ($C_2^n$). When the processors execute $A_2$, they can use (but not modify) the information of the states produced by $A_1$. Therefore, the set of system states of algorithm $A_2$ is $S_1 \times S_2 \times C^n$. The set of system states of the composed algorithm $A$ is also $S_1 \times S_2 \times C^n$. The $A_1$-projection of a state in $S_1 \times S_2 \times C^n$ consists of the components in $S_1 \times C_1^n$. Similarly, an $A_1$-projection of an execution is the sequence of $A_1$-projections of the system states of the execution.

We say that algorithm $A_2$ is self-stabilizing for task $T_2$, given task $T_1$, if every fair execution of $A_2$, whose $A_1$-projection is a sequence of safe states for $T_1$, has a suffix of safe states for $T_2$. We present a theorem, proved in [13], that gives sufficient conditions
for the composition of two self-stabilizing algorithms to be self-stabilizing.

**Theorem 2.3.1.** Assume that $A_2$ is self-stabilizing for a task $T_2$ given task $T_1$. If $A_1$ is self-stabilizing for $T_1$, then a fair composition of $A_1$ and $A_2$ is self-stabilizing for $T_2$.

### 2.4 Transformations of Self-Stabilizing Algorithms from One Model to Another

In this section we define the correctness of a transformation of self-stabilizing algorithms from one model to another. Then we present some techniques to prove correctness.

#### 2.4.1 Correctness of Transformation of Self-Stabilizing Algorithms

We are interested in transforming self-stabilizing algorithms that are designed to work in model $\mathcal{M}$ so that they will stabilize for the same task in model $\mathcal{M}'$. A transformation of self-stabilizing algorithms from model $\mathcal{M}$ to model $\mathcal{M}'$ takes as input an algorithm $A$ that works in model $\mathcal{M}$ and produces as output an algorithm $A'$ that works in model $\mathcal{M}'$. We only consider transformations such that for every processor $P_i$ with identifier $i$ in model $\mathcal{M}$, there is a corresponding processor $P'_i$ with identifier $i$ in model $\mathcal{M}'$.

Intuitively, if the transformation is correct, algorithm $A'$ eventually simulates algorithms $A$ (and then eventually stabilizes to the task of $A$). This means that there is a point in every execution of $A'$ after which the trace of the suffix of the execution is equal to the trace of some execution of $A$. Note that this execution of $A$ does not necessarily
start from a safe state for the task of $A$.

More formally, a transformation is correct if, for any algorithm $A$ of model $\mathcal{M}$, the resulting algorithm $A'$ of model $\mathcal{M}'$ has a subset $Q''$ of its set of states $Q'$, such that:

1. In every execution of the transformed algorithm $A'$, eventually all the states will be in $Q''$.

2. For any execution $a'$ of the transformed algorithm $A'$, starting from a state in $Q''$, there is an execution $a$ of the original algorithm $A$ such that $\text{trace}(a') = \text{trace}(a)$.

The rest of the section presents a technique that can be used in the proof of correctness of a transformation. We define simulation of the states of $A$ and simulation of the operations of $A$ in Sections 2.4.2 and 2.4.3, respectively. Next, in Section 2.4.4, we define simulation of executions, using the definitions of simulation of states and simulation of operations. Finally, we prove that simulation of executions can be used to prove that a transformation satisfies the second correctness condition.

### 2.4.2 Simulation of States

First, we consider the states of the processors of the two systems. Suppose $Q_i$ is the set of states of processor $P_i$ and $Q'_i$ is the set of states of processor $P'_i$ that occur in some state in $Q''$. A simulation of the states of processor $P_i$ by $P'_i$ is a function $\nu_i : Q'_i \rightarrow Q_i$. Informally, $P'_i$ must contain at least as much information as $P_i$ in order to simulate its behavior. The reason why $\nu_i$ is defined to be a function instead of a projection is because the value of the program counter in some state of $P_i$ is not necessarily the same as the value of the program counter in the corresponding state of $P'_i$. This may happen
because processor \( P_i \) may need to perform a sequence of operations in order to simulate an operation of \( P_1 \). In that case, the program counter will not increment in the same way in the two executions of the models \( \mathcal{M}' \) and \( \mathcal{M} \). Let \( C_i \) and \( C'_i \) be the sets of possible values of the program counters of \( P_i \) and \( P'_i \), respectively, and \( S_i \) and \( S'_i \) be the sets of possible states of \( P_i \) and \( P'_i \), respectively, excluding their program counters. Then \( Q_i \subseteq S_i \times C_i \) and \( Q'_i \subseteq S'_i \times C'_i \). and the function \( \psi_i \) can be viewed as the product of a projection from \( S'_i \) to \( S_i \) and a function from \( C'_i \) to \( C_i \).

Next, we consider the communication media of the models \( \mathcal{M} \) and \( \mathcal{M}' \). Let \( W \) be the set of states of the communication medium in \( \mathcal{M} \) and let \( W' \) be the set of states of the communication medium in \( \mathcal{M}' \) that occur in some state in \( Q'' \). A simulation of states of the communication medium of the model \( \mathcal{M} \) by the communication medium of \( \mathcal{M}' \) is a function \( \varphi : Q'' \rightarrow W \). Intuitively, we would like \( \varphi \) to be a projection from \( W' \) to \( W \), because we want \( W' \) to contain at least as much information as \( W \). However, this cannot happen in some cases. For example, suppose that \( \mathcal{M} \) is the shared memory model and \( \mathcal{M}' \) is the message passing model. Say \( P_1 \) reads from the shared register \( \mathcal{M}_{1,2} \) and \( P_2 \) writes to it. Likewise, \( P'_1 \) receives messages from the queue \( L'_{1,2} \) to which \( P'_2 \) sends messages. Now, assume a situation where \( L'_{1,2} \) contains exactly one message. Then, after \( P'_1 \) receives this message, the queue will be empty. In contrast, in the shared memory model, after \( P_1 \) reads \( \mathcal{M}_{1,2} \), the shared register remains unchanged. The problem is that the empty queue contains less information than the shared register. What we can do for this case is to store the received message in the local memory of processor \( P'_1 \) and map the pair that consists of the state of the processor \( P'_1 \) and the state of the queue \( L'_{1,2} \) to the state of the shared memory. In this way, the information in the shared register \( \mathcal{M}_{1,2} \) can
be stored in the local memory of some processor of the system $\mathcal{M}'$, if the transformation is properly designed.

Now, we are able to give similar definitions for the system states. Note that the set of states of the system $\mathcal{M}$ that has $n$ processors is $Q = Q_1 \times \ldots \times Q_n \times W$ and the set of states of $\mathcal{M}'$ that we consider is $Q'' \subseteq Q'_1 \times \ldots \times Q'_n \times W'$. A simulation of system states of model $\mathcal{M}$ by system states of model $\mathcal{M}'$ is a function $\chi : Q'' \rightarrow Q$. This function $\chi$ is the combination of the functions $\psi_1, \ldots, \psi_n$, and $\phi$.

A similar definition can be made for the environment of processors. Let $W_i$ be the set of states of the portion of the communication medium directly accessed by $P_i$ and let $W'_i$ be the set of states of the portion of the communication medium directly accessed by $P'_i$ that occur in some state in $Q''$. Then, $E_i = Q_i \times W_i$ is the set of states of the environment of $P_i$, and $E'_i = Q'_i \times W'_i$ is the set of states of the environment of $P'_i$. A simulation of the environment of $P_i$ by the environment of $P'_i$ is a function $\xi_i : E'_i \rightarrow E_i$. This function is defined in terms of $\psi_i$ and $\phi$.

In the following table we summarize our notation for the simulation of the states in two systems:

<table>
<thead>
<tr>
<th>States of processor $P_i$</th>
<th>Model $\mathcal{M}$</th>
<th>Model $\mathcal{M}'$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_i$</td>
<td>$Q'_i$</td>
<td>$\psi_i : Q'_i \rightarrow Q_i$</td>
<td></td>
</tr>
<tr>
<td>States of the communication medium</td>
<td>$W$</td>
<td>$W'$</td>
<td>$\phi : Q'' \rightarrow W$</td>
</tr>
<tr>
<td>States of the system</td>
<td>$Q$</td>
<td>$Q'' \subseteq Q'$</td>
<td>$\chi : Q'' \rightarrow Q$</td>
</tr>
<tr>
<td>States of the environment</td>
<td>$E_i$</td>
<td>$E'_i$</td>
<td>$\xi_i : E'_i \rightarrow E_i$</td>
</tr>
</tbody>
</table>

### 2.4.3 Simulation of Operations

Now, we define a simulation of the operations provided by model $\mathcal{M}$ by operations (or sequences of operations) provided by model $\mathcal{M}'$ if it satisfies the following property:
For every operation $\tau$ of model $M$, there is a finite, nonempty sequence of operations $\tau'_1, \ldots, \tau'_k$ of model $M'$ such that for every transition $(q_1, \tau, q_2)$ by $P_j$ in $M$ there is a nonempty finite sequence of transitions $(q'_1, \tau'_1, q'_2), (q'_2, \tau'_2, q'_3), \ldots, (q'_k, \tau'_k, q'_{k+1})$ by $P'_j$ in $M'$ such that $\chi(q'_1) = q_1$ and $\chi(q'_{k+1}) = q_2$. We say that the sequence of operations $\sigma(\tau) = \tau'_1, \ldots, \tau'_k$ simulates $\tau$. Since the interface operations are the same in both models, an interface operation simulates itself.

We give some useful observations about the applicability of operations.

**Observation 2.4.1.** Consider a state $s$ of model $M$ and a state $s'$ of model $M'$ such that $\xi_j(\text{env}(s'.j)) = \text{env}(s, j)$. If the first operation in the sequence $\sigma(\tau)$ can be performed by $P'_j$ in state $s'$, then operation $\tau$ can be performed by $P_j$ in state $s$.

The converse is not true. Consider the state $s$ of model $M$ and the state $s'$ of model $M'$ such that $\xi_j(\text{env}(s'.j)) = \text{env}(s, j)$, as above. Processor $P'_j$ in $s'$ contains the information needed to simulate the variables used by $P_j$ in $s$, but it also contains some additional control information used by the simulating algorithm. Suppose that an operation $t$ by $P_j$ is applicable to state $s$ in model $M$. Because of the control information in $s'$ that does not exist in $s$, the sequence of operations $\sigma(\tau)$ may not be applicable to $s'$. For example, suppose that $\sigma(\tau) = \text{write}(0)$ by $P'_j$ to some register. Also, assume that according to the simulating algorithm, $A'$, processor $P'_j$ will be able to perform write(0) after it increments a shared counter (used for controlling the simulation). Let $s'$ be the state immediately before $P'_j$ increments the counter and let $s''$ be the state reached from $s'$ if $P'_j$ increments the counter. Although $\xi_j(\text{env}(s'.j)) = \xi_j(\text{env}(s''.j)) = \text{env}(s, j)$, write(0) by $P_j$ is not applicable to $s'$ because the only applicable operation by $P'_j$ to $s'$ is
the increment of the counter. Note that write(0) by $P_j$ is applicable to state $s''$.

![Figure 2.1: Example](image)

**2.4.4 Simulation of Executions**

We will give some preliminary definitions, and then present the formal definition of a simulation of executions. If $\sigma(\tau) = \tau'_1, \ldots, \tau'_k$ is executed by $P'_i$ in some execution $a'$, then any state in $a'$ between the operations in $\sigma(\tau)$ is called an **intermediate state** for $P'_i$.

**Execution** $a' = s'_1, t'_1, s'_2, t'_2, \ldots$ **simulates execution** $a = s_1, t_1, s_2, t_2, \ldots$ if for all $1 \leq i \leq n$, the following hold:

1. The first state of $a'$ simulates the first state of $a$: i.e. $\xi_i(\text{env}(s'_i, i)) = \text{env}(s_1, i)$.

2. If $\tau_1, \tau_2, \ldots$ is the sequence of operations performed by $P_i$ in $a$, and $\tau'_1, \tau'_2, \ldots$ is the sequence of operations performed by $P'_i$ in $a'$, then there exist $0 \leq k_0 < j_1 \leq k_1 < j_2 \leq k_2 < \ldots$ such that:

   (a) All the operations performed by $P_i$ in $a$ are simulated by a sequence of operations in $a'$ performed consecutively by $P'_i$. Formally, for all $l \geq 1$, if $\tau_l$ is the $g^{th}$ operation of $a$, (i.e. $t_g = \tau_l$), and $\tau'_{j_l}$ is the $g''^{th}$ operation of $a'$.
(i.e. $t'_{q'} = \tau'_{j_{l}}$), then \( \xi_{i}(env(s'_{q'}, i)) = env(s_{q}, i) \), and the sequence of operations \( \tau'_{j_{l}}. \tau'_{j_{l}+1}. . . . \tau'_{k_{l}} \) simulates \( \tau_{l} \).

(b) Processor \( P'_{i} \) does not simulate any operation in \( a' \) that is not executed by \( P_{i} \) in \( a \). In particular, this means that in between any two sequences of operations by \( P'_{i} \) that simulate two consecutive operations by \( P_{i} \), processor \( P'_{i} \) does not change its simulating environment. Formally, for all \( h \geq 0 \) and all \( r \) such that \( k_{h} < r < j_{h+1} - 1 \), if \( \tau'_{r} \) is the \( q^{th} \) operation of \( a' \). (i.e. \( t'_{q'} = \tau'_{r} \)), then

\[
\xi_{i}(env(s'_{q'}, i)) = \xi_{i}(env(s'_{q'+1}, i))
\]

3. **Atomicity Property**: Atomicity of the operations in \( a \) is preserved in \( a' \). In particular, if \( s'_{r} \) is an intermediate state for \( P'_{i} \) and \( (s'_{r}, t'_{r}, s'_{r+1}) \) is a transition performed by another processor \( P'_{m} \), then the following hold:

(a) Operation \( t'_{r} \) does not change the environment of processor \( P'_{i} \).

(b) If operation \( t'_{r} \) belongs to a sequence of operations \( \sigma(\lambda) \) for some operation \( \lambda \) of model \( \mathcal{M} \), then \( P'_{m} \) is not a neighbour of \( P'_{i} \) (i.e. \( m \notin \mathcal{NBR}_{i} \)).

By the atomicity property, we know that any operation in \( a' \) between \( \tau'_{j_{l}} \) and \( \tau'_{k_{l}} \) performed by a processor other than \( P'_{i} \) will not affect the environment of \( P'_{i} \). Therefore, we can ignore all the operations in \( a' \) between \( \tau'_{j_{l}} \) and \( \tau'_{k_{l}} \) performed by other processors and apply the definition of simulation of operations. As a result, we get that the state of the environment of \( P'_{i} \) in the state immediately after \( \tau'_{k_{l}} \) in \( a' \) simulates the state of the environment of \( P_{i} \) in the state immediately after \( \tau_{l} \) in \( a \).

**Observation 2.4.2.** Consider the sequence of operations \( \tau'_{j_{1}} . . . \tau'_{k_{l}} = \sigma(t_{l}) \). performed
by $P_i'$ in $a'$. Suppose that $\tau_{k_i}'$ is the $\ell^{th}$ operation of $a'$ (i.e. $t_{\ell}' = \tau_{k_i}'$) and $\tau_{i}$ is the $\ell^{th}$ operation of $a$ (i.e. $t_{\ell} = \tau_{i}$). Then $\xi_i(\text{env}(s_{\ell'+1}, i)) = \text{env}(s_{\ell+1}, i)$.

We will also allow operations in $\mathcal{M}'$ that do not simulate any operation in $\mathcal{M}$. We call these operations **auxiliary operations**. Note that auxiliary operations cannot be interface operations.

Since $P_i'$ performs the same interface operations as $P_i$ and in the same order (by the definition of simulation of execution), we get the following observation.

**Observation 2.4.3.** If execution $a'$ simulates execution $a$, then the sequence of interface operations performed by some processor in $a'$ is the same as the sequence of interface operation performed by the corresponding processor in $a$: i.e. for all $1 \leq i \leq n$. $\text{trace}_i(a') = \text{trace}_i(a)$.

We present an example to demonstrate that the fact that an execution $a'$ simulates an execution $a$ does not ensure that $a$ and $a'$ have the same traces. Let $a = s_1. op_1. s_2. op_2. s_3. . . . \text{ and } a' = s'_1. op_1'. s'_2. op_2'. s'_3. . . . \text{ be such that } a'$ simulates $a$. and $\text{trace}(a') = \text{trace}(a)$. Suppose the task of the algorithms that produce these executions is mutual exclusion. $op_1$ is the "enter trying section" interface operation by $P_1$ in $a$ (and by $P_i'$ in $a'$) and $op_2$ is the "enter trying section" interface operation by $P_2$ in $a$ (and by $P_2'$ in $a'$). Each of these interface operations will only change the program counter of the processor that executes it. Since they are performed by different processors and the execution of one does not affect the execution of the other, we can reverse them and get a new execution $\hat{a} = s_1. op_2. q_2. op_1. s_3. . . .$. Because $a'$ simulates $a$, $a'$ also simulates $\hat{a}$, but $\text{trace}(a') \neq \text{trace}(\hat{a})$. 
The definition of simulation also applies when \( \mathcal{M} \) (and/or \( \mathcal{M}' \)) is a synchronous system. Note that in this case, \( \tau_1, \tau_2, \ldots \) are the operations performed by \( P_i \) in the synchronous rounds 1, 2, \ldots, respectively. (Similarly, \( \tau'_1, \tau'_2, \ldots \) are the operations performed by \( P'_i \) in the synchronous rounds 1, 2, \ldots, respectively.) In the model \( \mathcal{M}' \), two simulated synchronous rounds of model \( \mathcal{M} \) may interleave. This may happen when non-neighbouring processors of some processor \( P' \) are still executing operations to simulate a synchronous round while \( P' \) has started executing the next simulated synchronous round. This is allowed because the operations performed by the non-neighbouring processors of \( P' \) will not affect the computation by \( P' \) in the new synchronous round.

### 2.4.5 Correctness of Transformations Using Simulation

We will use the above definition of simulation of executions to prove correctness of transformations. In particular, we will show that if execution \( a' \) simulates an execution \( a \) of some algorithm, then it simulates some other execution \( b \) of the same algorithm, such that \( \text{trace}(a') = \text{trace}(b) \). From Observation 2.4.3, all the interface operations in \( a' \) must appear in \( a \), but they are not necessarily in the same order. In this section, we will show how to create the execution \( b \) by reordering operations in \( a \) so that the order of interface operations in \( b \) will be the same as in \( a' \).

Let \( op_1 \) and \( op_2 \) be two interface operations in \( a' \) that occur in the reverse order in \( a' \). There are two cases that may happen: the ranges of \( op_1 \) and \( op_2 \) overlap, or their ranges are in reverse order. The first case is easy to deal with because an interface operation can happen at any time in its range, leaving the other operations in the execution unchanged.
In this case, reordering of the interface operations is possible without affecting the rest of the execution. To deal with the second case, we reorder the external, internal operations of a to get another valid execution b in which the ranges of the interface operations $op_1$ and $op_2$ are possibly reversed or extended so that the interface operations can be arranged and have the same order as in $a'$.

Moving an interface operation $op$ by some processor $P_i$ in its range will not affect the remaining operations of the execution, because $op$ will not be noticed by the other processors unless $P_i$ executes an external operation following $op$. Furthermore, moving an interface operation by $P_i$ in its range will not affect $P_i$'s computation because it will not provide different amount of information to this processor.

Next, we present some definitions that are used by the following lemmata and theorems in this section. Let $a' = s'_1, t'_1, s'_2, t'_2, \ldots$ be an execution in model $\mathcal{M}'$ and $a = s_1, t_1, s_2, t_2, \ldots$ be an execution of an algorithm $A$ in model $\mathcal{M}$ such that $a'$ simulates $a$. Each operation $t_i$ of $a$ is performed by processor $P_{s_i}$. We define the permutation $\tau = \tau_1, \tau_2, \ldots$ of the operations in $a$, so that the first operation of $\sigma(t_i)$ occurs before the first operation of $\sigma(t_j)$ in $a'$. then $t_i$ occurs before $t_j$ in $\tau$. We denote by $s'_n$ the state in $a'$ immediately before the first operation of $\sigma(\tau_i)$ is performed and by $s'_{n+1}$ the state in $a'$ immediately after the last operation of $\sigma(\tau_i)$ is performed. Let $\hat{b} = q_1, \tau_1, q_2, \tau_2, \ldots$ be the execution obtained by applying the operations in $\tau$ to $s_1 = q_1$. We illustrate these definitions in Figure 2.2.

In order to prove that $\hat{b}$ is a valid execution of algorithm $A$, we need to prove that $\tau$ is applicable to $q_1$. We prove this by showing that, for all $j \geq 1$, the environment of $P_{s_j}$ in state $q_j$ (i.e. immediately before $\tau_j$) is simulated by the environment of $P'_{s_j}$ in state


\[
\begin{array}{c}
\text{EXECUTION } \hat{b} \\
\begin{array}{c}
\mathcal{S}_1 \\
\tau_1 \\
\mathcal{S}_2 \\
\tau_2 \\
\mathcal{S}_3 \\
\tau_3 \\
\mathcal{S}_{i+2} \\
\tau_{i+2} \\
\mathcal{S}_{j-2} \\
\tau_{j-2} \\
\mathcal{S}_j \\
\tau_j \\
\mathcal{S}_{j+1} \\
\tau_{j+1} \\
\mathcal{S}_{j+2} \\
\tau_{j+2}
\end{array}
\end{array}
\begin{array}{c}
\mathcal{S}_1' \\
\tau_1' \\
\mathcal{S}_2' \\
\tau_2' \\
\mathcal{S}_3' \\
\tau_3' \\
\mathcal{S}_{i+2}' \\
\tau_{i+2}' \\
\mathcal{S}_{j-2}' \\
\tau_{j-2}' \\
\mathcal{S}_j' \\
\tau_j' \\
\mathcal{S}_{j+1}' \\
\tau_{j+1}' \\
\mathcal{S}_{j+2}' \\
\tau_{j+2}'
\end{array}
\xrightarrow{\mathcal{S}} \hat{b}
\]

\[
\begin{array}{c}
\text{EXECUTION } a'
\end{array}
\]

\[
\begin{array}{c}
\mathcal{S}_1 \\
\tau_1 \\
\mathcal{S}_2 \\
\tau_2 \\
\mathcal{S}_3 \\
\tau_3 \\
\mathcal{S}_{i+2} \\
\tau_{i+2} \\
\mathcal{S}_{j-2} \\
\tau_{j-2} \\
\mathcal{S}_j \\
\tau_j \\
\mathcal{S}_{j+1} \\
\tau_{j+1} \\
\mathcal{S}_{j+2} \\
\tau_{j+2}
\end{array}
\begin{array}{c}
\mathcal{S}_1' \\
\tau_1' \\
\mathcal{S}_2' \\
\tau_2' \\
\mathcal{S}_3' \\
\tau_3' \\
\mathcal{S}_{i+2}' \\
\tau_{i+2}' \\
\mathcal{S}_{j-2}' \\
\tau_{j-2}' \\
\mathcal{S}_j' \\
\tau_j' \\
\mathcal{S}_{j+1}' \\
\tau_{j+1}' \\
\mathcal{S}_{j+2}' \\
\tau_{j+2}'
\end{array}
\xrightarrow{\mathcal{S}} a'
\]

Figure 2.2: Definition of execution \( \hat{b} \)

\( s'_{y_j} \) (i.e. immediately before the first operation in \( \sigma(\tau_j) \)). Then, the validity of \( \hat{b} \) follows from Observation 2.4.1.

**Lemma 2.4.4.** Execution \( \hat{b} \) is valid for algorithm A.

**Proof.** We prove by induction that, for all \( j \geq 1 \), \( \xi_{\mathcal{G}_j}(env(s'_{y_j}, g_j)) = env(q_j, g_j) \) and \( \tau_j \) is applicable to \( q_j \). Because \( a' \) simulates \( a \), \( \xi_l(env(s'_l, l)) = env(s_1, l) \) for all \( 1 \leq l \leq n \). In particular, for \( l = g_1 \) we get that \( \xi_{g_1}(env(s'_{1}, g_1)) = env(s_1, g_1) \). Since \( q_1 = s_1 \), \( \xi_{g_1}(env(s'_1, g_1)) = env(q_1, g_1) \). By definition of \( \tau \), we know that between \( s'_1 \) and \( s'_{y_1} \) no operation belongs to a sequence of operations that simulate an operation of \( a \). This means
that only auxiliary operations happen between $s'_1$ and $s'_{y_1}$. As a result, $\xi_{q_1}(\text{env}(s'_{y_1}, g_1)) = \xi_{q_1}(\text{env}(s'_1, g_1))$ and hence $\xi_{q_1}(\text{env}(s'_{y_1}, g_1)) = \text{env}(q_1, g_1)$. Because $\tau'_1$ is applicable to $s'_{y_1}$, it follows by Observation 2.4.1 that $\tau_1$ is applicable to $q_1$.

Now, let $j \geq 1$ and suppose $\xi_{q_k}(\text{env}(s'_{y_k}, g_k)) = \text{env}(q_k, g_k)$ for $1 \leq k \leq j - 1$. We will prove that $\xi_{q_j}(\text{env}(s'_{y_j}, g_j)) = \text{env}(q_j, g_j)$. We consider two possible cases.

1. First, assume that none of the operations $\tau_1, \ldots, \tau_{j-1}$ are performed by $P_{q_j}$. That means that the state of $P_{q_j}$ will be the same in states $q_1, \ldots, q_j$, and the part of the state of $P'_{q_j}$ that simulates the state of $P_{q_j}$ will be the same in states $s'_1, \ldots, s'_{y_j}$. Since $\xi_1(\text{env}(s'_1, l)) = \text{env}(s_1, l)$, for all $1 \leq l \leq n$, the state of $P'_{q_j}$ in $s'_{y_j}$ simulates the state of $P_{q_j}$ in $q_j$. It remains to prove the same for the part of the communication medium directly accessed by processors $P'_{q_j}$ and $P_{q_j}$. Let $\tau_1, \ldots, \tau_r$ be the operations performed by neighbours of $P_{q_j}$ between $q_1$ and $q_j$ that change the environment of $P_{q_j}$. Then, by the induction hypothesis, $\xi_{q_m}(\text{env}(s'_{y_m}, g_{i_1})) = \text{env}(q_{i_1}, g_{i_1})$ and by Observation 2.4.2 we get that $\xi_{q_m}(\text{env}(s'_{y_m}, g_{i_1})) = \text{env}(q_{i_m+1}, g_{i_m})$, for all $1 \leq m \leq r$.

Because $a'$ simulates $a$, the atomicity property holds for execution $a'$. Let $m \in \{i_1, \ldots, i_r\}$. Since $\tau_m$ happens before $\tau_j$, the first operation of $\sigma(\tau_m)$ occurs before $s'_{y_j}$. If the last operation of $\sigma(\tau_m)$ occurs after $s'_{y_j}$, then $s'_{y_j}$ is an intermediate state for $P'_{q_m}$. Then, from part (b) of the atomicity property, $g_j \notin \text{VBR}(g_m)$. But this is impossible, since $P'_{q_m}$ is a neighbour of $P'_{q_j}$. Hence, $\sigma(\tau_m)$ is finished by state $s'_{y_j}$.

Note that $\tau_1, \ldots, \tau_r$ are the only operations that may change the environment of $P_{q_j}$ between the states $q_1$ and $q_j$. Similarly, the operations in the sequence
\(\sigma(\tau_1), \ldots, \sigma(\tau_n)\) are the only operations that may change the simulating environment of \(P'_{g_i}\) between the states \(s'_1\) and \(s'_{y_i}\).

2. Now, assume that \(P_{g_j}\) performs some operations before \(\tau_j\). Let \(\tau_i\) be the last operation performed by \(P_{g_j}\) in \(\hat{b}\) before \(\tau_j\). Then, by the induction hypothesis, \(\xi_{g_j}(env(s'_y, g_i)) = env(q_{y}, g_i)\). Since \(g_i = g_j\), it follows by Observation 2.4.2 that \(\xi_{g_j}(env(s'_x, g_j)) = env(q_{i+1}, g_j)\). As in the previous case, let \(\tau_i, \ldots, \tau_r\) be the operations performed by neighbours of \(P_{g_j}\) between \(\tau_i\) and \(\tau_j\) that change the environment of \(P_{g_j}\). As above, \(\xi_{g_j}(env(s'_x, g_i)) = env(q_{i+1}, g_i)\) and, by Observation 2.4.2 we get that \(\xi_{g_j}(env(s'_x, g_i)) = env(q_{i+1}, g_i)\), for all \(1 \leq f \leq r\).

Because \(a'\) simulates \(a\), the atomicity property holds for execution \(a'\). Let \(m \in \{i_1, \ldots, i_r\}\). Since \(\tau_m\) happens after \(\tau_i\), but before \(\tau_j\), the first operation of \(\sigma(\tau_m)\) occurs after \(s'_y\), but before \(s'_{y_j}\). If the first operation of \(\sigma(\tau_m)\) occurs before \(s'_x\), then \(s'_{ym}\) is an intermediate state for \(P'_{g_j}\). Then, from part (b) of the atomicity property \(g_m \notin NBR_{g_j}\). Similarly, if the last operation of \(\sigma(\tau_m)\) occurs after \(s'_y\), then \(s'_{ym}\) is an intermediate state for \(P'_{g_m}\). Then, from part (b) of the atomicity property \(g_j \notin NBR_{g_m}\). But \(P'_{g_m}\) is a neighbour of \(P'_{g_j}\). Hence, \(\sigma(\tau_m)\) starts after state \(s'_{z_i}\) and is finished by state \(s'_{y_j}\).

Now, suppose that there is a sequence of operations \(\sigma(\tau_l)\), where \(l \notin \{i_1, \ldots, i_r\}\) that contains an operation that changes the simulating environment of \(P'_{g_j}\) between \(s'_{z_i}\) and \(s'_{y_i}\). Then \(g_l \in NBR_{g_j}\). The first operation of \(\sigma(\tau_l)\) must be before \(s'_{y_i}\). Then, by definition of \(\tau\), \(\tau_l\) must be before \(\tau_j\) in \(\hat{b}\). In fact, the first operation of \(\sigma(\tau_l)\) must be before \(s'_{y_i}\). This is because otherwise, \(\tau_l\) would occur after \(\tau_i\) and since the
only operations between \( \tau_i \) and \( \tau_j \) performed by neighbours of \( P_j \) are \( \tau_{1, \ldots, \tau_r} \). \( \tau_i \) would be one of them. As a result, state \( s_i' \) is an intermediate state for \( P_j \). Then, from part (b) of the atomicity property that holds for \( a' \), we get that \( g_j \notin NBR_j \), which is a contradiction.

From the discussion above, the operations of the sequences \( \sigma(\tau_{i_1}), \ldots, \sigma(\tau_{i_r}) \) are the only operations that may change the simulating environment of \( P_j' \) between the states \( s_i' \) and \( s_j' \). Note that \( \tau_{1, \ldots, \tau_r} \) are the only operations that may change the environment of \( P_j \) between the states \( q_{i+1} \) and \( q_j \).

Let \( H = \max\{z_{i_1}, \ldots, z_{i_r}\} \). Then, \( s'_H \) is the state reached after all operations in \( \sigma(\tau_{i_1}), \ldots, \sigma(\tau_{i_r}) \) are executed. From the analysis of both cases above, the environment of \( P_j' \) in state \( s'_H \) will simulate the environment of \( P_j \) in the state reached after \( \tau_{i_r} \): i.e. \( \xi_{j_i}(env(s'_H, g_j)) = env(q_{i+1}, g_j) \).

Between \( q_{i_r+1} \) and \( q_j \) the environment of \( P_j \) will not change, because neither \( P_j \), nor any of its neighbours will perform any operation during this part of the execution. Similarly, between \( s'_H \) and \( s'_j \) the part of the environment of \( P_j' \) that simulates the environment of \( P_j \), will not change. As a result, we get that \( env(q_{i_r+1}, g_i) = env(q_j, g_j) \) and \( \xi_{j_i}(env(s'_H, g_i)) = \xi_{j_i}(env(s'_j, g_j)) \) and hence, \( env(q_i, g_i) = \xi_{j_i}(env(s'_H, g_i)) \). Now, we apply Observation 2.4.1 and get that \( \tau_i \) is applicable to \( q_i \).

Now, we will prove that the interface operations in \( b \) can be rearranged to get a new execution \( b \) of algorithm \( A \) that has the same trace as \( a' \).

First, we need to introduce some notation and definitions. Consider an interface operation \( op \) performed by some processor \( P' \) at a particular time in some execution \( a' \) in
model \( \mathcal{M}' \). Then, we define the \textit{superrange}\(_{op} \) of \( op \) to be the fragment of the execution \( a' \) with the following properties: This fragment starts with the first operation \( t'_u \) of the sequence of operations \( \sigma(t_i) \) performed by \( P' \) that simulates the preceding operation \( \tau_i \) of execution \( \hat{b} \) (or the first operation of the execution \( a' \), if \( P' \) simulates no operation of \( a \) before \( op \)), and ends with the first operation \( t'_x \) of the sequence of operations \( \sigma(t_j) \) performed by \( P \) that simulates the following operation \( \tau_j \) of execution \( \hat{b} \). Let \( t'_c \) and \( t'_w \) denote the operations where \( \text{range}_{op} \) starts and ends in \( a' \).

Consider two sequences of operations performed by the same processor that each of them simulates one operation in \( a \). These sequences do not overlap. Then, by construction of \( \tau \), the operations performed by \( P \) in \( \hat{b} \) have the same order as the corresponding sequences of operations in \( a' \) that simulate them. Now, if we look at any interface operation \( op \) in \( a' \), from the above observations in \( a' \), the preceding and the following sequences of operations that simulate an operation in \( \hat{b} \) correspond to the upper and lower boundaries of \( \text{range}_{op} \) in \( \hat{b} \). Note that in between the last operation of \( \sigma(t_i) \) and the first operation of \( \sigma(t_j) \) there might be other external auxiliary operations that will be the boundaries of the range of \( op \) in \( a' \).

\textbf{Observation 2.4.5.} The execution fragment \textit{superrange}\(_{op} \) contains the execution fragment \textit{range}\(_{op} \) for any interface operation \( op \) in \( a' \) (i.e. \( t'_u \leq t'_c < t'_w \leq t'_x \)).

We illustrate the definition of superrange and range in Figure 2.3.

\textbf{Lemma 2.4.6.} If \( \text{range}_{op_1} \) and \( \text{range}_{op_2} \) overlap in \( a' \), then they overlap in \( \hat{b} \) as well. If \( \text{range}_{op_1} \) occurs before \( \text{range}_{op_2} \) in \( a' \), then the first operation of \( \text{range}_{op_1} \) occurs before the last operation of \( \text{range}_{op_2} \) in \( \hat{b} \).
Proof. First, we consider the case where $\text{range}_{op_1}$ and $\text{range}_{op_2}$ overlap in $a'$. Without loss of generality assume that $t'_{i_1} < t'_{i_2} < t'_{w_1} < t'_{w_2}$. Then, by Observation 2.4.5, we get $t'_{u_1} < t'_{u_1} < t'_{w_1} < t'_{z_1}$, and $t'_{w_1} < t'_{z_2}$. Now, by definition of $\tau$, we get that $\tau_{i_1} < \tau_{z_2}$ and $\tau_{i_2} < \tau_{j_1}$. Therefore the range of operation $op_1$ and the range of operation $op_2$ in $\hat{b}$ overlap.

Now, suppose that $\text{range}_{op_1}$ and $\text{range}_{op_2}$ do not overlap in $a'$. Without loss of
generality, assume that \( t'_{v_1} < t'_{w_1} < t'_{w_2} < t'_2 \). As above, from Observation 2.4.5, we get 
\( t'_{u_1} \leq t'_{v_1} \) and \( t'_{w_2} \leq t'_{x_2} \), so \( t'_{u_1} < t'_{x_2} \). Then, by definition of \( \tau \), we get that \( \tau_{i_1} < \tau_{j_2} \). \( \square \)

Now, we will prove the main theorem of this section.

**Theorem 2.4.7.** If there is an execution \( a' \) in \( M' \) that simulates an execution \( a \) of an algorithm \( A \) in \( M \), then there is an execution \( b \) of the algorithm \( A \) such that \( \text{trace}(a') = \text{trace}(b) \).

**Proof.** From Lemma 2.4.6 we can reorder the interface operations of \( \hat{b} \) by moving them in their range and get another execution \( b \) such that \( \text{trace}(a') = \text{trace}(b) \). By Lemma 2.4.4, \( \hat{b} \) is valid for algorithm \( A \). As we explained in the introduction of this section by moving an interface operation in its range will not affect the rest of the execution. Therefore, \( b \) is valid as well. \( \square \)

The notion of simulation can be used to prove correctness of a transformation. Recall that the second part of the definition of correctness of a transformation requires that, for any execution \( a' \) of the transformed algorithm, starting from a state in \( Q'' \), there is an execution \( a \) of the original algorithm, such that \( \text{trace}(a') = \text{trace}(a) \). Now, by applying Theorem 2.4.7, the above condition can be replaced with the following: for any execution \( a' \) of the transformed algorithm, starting from a state in \( Q'' \), there is an execution \( a \) of the original algorithm, such that \( a' \) simulates \( a \).
2.5 Other Transformation Techniques

Our definition for correctness of a transformation is a version of the ones presented in the literature, applied specifically to self-stabilizing algorithms. We will describe some simulation techniques presented in the literature and compare them with our technique, described in Section 2.4, that uses the simulation of executions.

Lynch and Vaandrager [19] present a number of different simulation techniques for non self-stabilizing algorithms. They define mappings from an algorithm \( A' \) to an algorithm \( A \) that are used to prove that the set of traces of \( A' \) is contained in the set of traces of \( A \). Note that this is the same as our second condition for the correctness of a transformation.

One of the main differences of their definition as compared to ours is that they have to deal with initial states. We do not because, in self-stabilization, the system can be initialized to any state. Instead, we deal with possible "bad" initial behavior of the algorithms. In particular, we allow \( A' \) not to simulate \( A \) for some initial period of time.

Their first simulation technique is refinement. Formally, a refinement map of \( A' \) to \( A \) is a function \( f \) from the states of \( A' \) to the states of \( A \), such that the following hold:

1. If a state \( s' \) is an initial state of \( A' \) then \( f(s') \) is an initial state of \( A \).

2. If \((s', \pi', s'')\) is a transition of \( A' \) then there is an execution fragment of \( A \) that starts with \( f(s') \) and ends with \( f(s'') \) and has the same trace as \( \pi' \).

This method is used for proving correctness of a refined (i.e. more detailed) version of an algorithm. For example, suppose that \( A' \) has the transitions: \((s'_1, t, s'_2)\) and \((s'_2, t, s'_2)\) and that \( A \) has the single transition \((s, t, s)\), where \( s'_1 \) is the single initial state of \( A' \) and
s is the single initial state of $A$. These simple algorithms will produce the same sets of traces. Although there is a refinement map from $A'$ to $A$ (mapping both $s'_1$ and $s'_2$ to state $s$), there is no refinement map from $A$ to $A'$. We have to map $s$ either to $s'_1$ or to $s'_2$. But, we cannot map $s$ to $s'_2$, because $s'_2$ is not an initial state and we cannot map $s$ to $s'_1$, because then transition $(s, t, s)$ would map to $(s'_1, t, s'_2)$ implying that $s$ maps to $s'_2$.

To deal with cases such as this, Lynch and Vaandrager [19] generalize refinement to new simulation techniques that allow a set of states of $A$ to correspond to a single state of $A'$. In particular, they define forward and backward simulations as follows. A forward simulation from $A'$ to $A$ is a relation $f$ over states of $A'$ and $A$ that satisfies the following:

1. If a state $s'$ is an initial state of $A'$ then at least one of the states in $f(s') = \{ s | (s, s') \in f \}$ is an initial state of $A$.

2. If $(s', \pi', s'')$ is a transition of $A'$ and $u' \in f(s')$ then there is an execution fragment of a valid execution of $A$ that starts with $u'$, ends with some $u'' \in f(s'')$ and has the same trace as $\pi$.

A backward simulation from $A'$ to $A$ is a relation $b$ over states of $A'$ and $A$ that satisfies the following:

1. If a state $s'$ is an initial state of $A'$ then all the states in $b(s')$ are initial states of $A$.

2. If $(s', \pi', s'')$ is a transition of $A'$ and $u'' \in b(s'')$, then there is an execution fragment of $A$ that starts with $u' \in b(s')$, ends with $u''$, and has the same trace as $\pi$. 
Backward simulation has this name because it goes backwards in time in the execution. We illustrate the usefulness of going backwards in time in the following example.

Let \( \mathcal{A} \) be a deterministic algorithm that has two transitions, \((s_1, t_1, q_1)\) and \((s_2, t_2, q_2)\) and let \( \mathcal{A}' \) be a non-deterministic algorithm that has the transitions \((s', t_1, q'_1)\) and \((s', t_2, q'_2)\). Then, from the definitions above, there is no forward simulation or refinement map from \( \mathcal{A}' \) to \( \mathcal{A} \), but there is a backward simulation from \( \mathcal{A}' \) to \( \mathcal{A} \). Algorithm \( \mathcal{A} \) has more states than \( \mathcal{A}' \) and \( \mathcal{A} \) can be viewed as a refined version of \( \mathcal{A}' \). Note that there is a refinement map from \( \mathcal{A} \) to \( \mathcal{A}' \).

Backward simulation is not applicable for self-stabilizing transformations because if we go backward in time starting from a safe state we may reach a state that is not safe, i.e., a state before the transformed algorithm stabilized.

The last simulation technique of Lynch and Vaandrager that we present is the history relation. A history relation from \( \mathcal{A}' \) to \( \mathcal{A} \) is a relation \( h \) between states of \( \mathcal{A}' \) and states of \( \mathcal{A} \) such that \( h \) is a forward simulation from \( \mathcal{A}' \) to \( \mathcal{A} \) and \( h^{-1} \) is a refinement from \( \mathcal{A} \) to \( \mathcal{A}' \). History relations are used to prove a stronger result: if there is a history relation from \( \mathcal{A}' \) to \( \mathcal{A} \) then the set of traces of \( \mathcal{A}' \) is equal to the set of traces of \( \mathcal{A} \).

We note that the definitions of refinement map, forward simulation, and our definition of simulation of executions are used to prove that traces of \( \mathcal{A}' \) are produced by \( \mathcal{A} \) as well.

The first difference between our definition and the ones proposed by Lynch and Vaandrager is that they deal with initial states. We do not, because there are no initial states in self-stabilization.

Second, recall that, in the simulation techniques by Lynch and Vaandrager, every transition of algorithm \( \mathcal{A}' \) is mapped to an execution fragment of \( \mathcal{A} \) that has the same
trace. On the other hand, in our method, described in Section 2.4.4, every operation in \( A \) will be simulated by a sequence of one or more operations of \( A' \).

In this thesis, we are interested in proving correctness of transformations between models. For this particular application our method is more natural, because the original algorithm \( A \) will have fewer transitions than the transformed algorithm \( A' \).

With refinement maps or forward simulations, any transition in \( A' \) that does not contain an interface operation can be mapped to any fragment of \( A \) that contains zero interface operations (even to a single state). This is not the case with our definition of simulation of execution.

Some of these differences are demonstrated in the following example, which is illustrate in Figure 2.4. Assume that \( \mathcal{M} \) contains a single processor and a single operation \( read(R_1, R_2) \), where \( R_1 \) and \( R_2 \) are registers. Consider an algorithm \( A \) that has the following two transitions: \( (s_1, read(R_1, R_2), s_2) \) and \( (s_2, op, s_1) \), where \( op \) is an interface operation. Now, assume that \( \mathcal{M}' \) contains a single processor and the two operations: \( read(R_1) \) and \( read(R_2) \), where \( R_1 \) and \( R_2 \) are registers. The sequence of operations \( read(R_1) \), \( read(R_2) \) will simulate the operation \( read(R_1, R_2) \) and that is expressed formally by our method. The transformed algorithm \( A' \) has the following transitions: \( (s'_1, read(R_1), s'_2) \), \( (s'_2, read(R_2), s'_3) \), and \( (s'_3, op, s'_1) \).

Now, we can apply a refinement map (without considering the condition of the initial states) to show that the traces of executions in \( A' \) are also traces of executions in \( A \). There are two ways to do this:

- We can map \( s'_1 \) to \( s_1 \), \( s'_2 \) to \( s_2 \), and \( s'_3 \) to \( s_2 \). Based on this mapping, the transi-
Figure 2.4: Example

The transformation $(s'_1, \text{read}(R_1), s'_2)$ maps to the transition $(s_1, \text{read}(R_1, R_2), s_2)$, the transition $(s'_2, \text{read}(R_1), s'_3)$ maps to the single state $s_2$, and the transition $(s_2, \text{op}, s_1)$ maps to the transition $(s'_3, \text{op}, s'_1)$.

- We can map $s'_1$ to $s_1$, $s'_2$ to $s_1$, and $s'_3$ to $s_2$. Based on this mapping, the transition $(s'_2, \text{read}(R_2), s'_3)$ maps to the transition $(s_1, \text{read}(R_1, R_2), s_2)$, the transition $(s'_1, \text{read}(R_1), s'_2)$ maps to the single state $s_1$, and the transition $(s_2, \text{op}, s_1)$ maps to the transition $(s'_3, \text{op}, s'_1)$.

In both solutions, the mapping is artificial, since we map a transition that contains a computation that affects the distributed system to an empty execution fragment.
2.6 Message Passing Models for Self-Stabilization

When we consider self-stabilizing algorithms in message-driven models, a problem arises. Consider an initial system state where the links are empty and all the processors are waiting to receive a message. Starting from such a state the system is deadlocked. This kind of deadlock is called communication deadlock.

The majority of the authors assume the existence of time-outs to deal with communication deadlock. Dolev [14] defines time-outs as environmental operations that will be executed when such a deadlock occurs. Dolev, Israeli, and Moran [15] assume a time-out device, which detects communication deadlocks and initializes the system to a state after which no communication deadlock ever occurs. If we carefully examine this choice, we see that this is behaving like an initialization procedure of the global state of the system. It is a very powerful assumption for self-stabilizing results, where initialization is not allowed.

Katz and Perry [17] provide another solution to the problem of communication deadlock. They assume that in any state there is at least one processor whose next operation is to send a message. Thus, they consider algorithms that stabilize from the subset of the possible initial states where this assumption holds, but not from arbitrary states of a message passing model.

The model that we propose solves the problem of communication deadlock and does not need these strong assumptions for self-stabilizing algorithms. That model is not message-driven. We will prove it is at least as powerful as the message-driven model. Specifically, we prove that, in our model, we can solve any problem that can be solved in
the message-driven model. We give a transformation of self-stabilizing algorithms from the message-driven model to our message passing model. This transformation is the composition of the following two transformations: the transformation from the message-driven model to the shared memory model with single-reader, single-writer registers that will be presented in Section 5.1, and the transformation from the shared memory model with single-writer single-reader registers to our message passing model that will be presented in Section 4.4.

Observe that there is also a transformation of self-stabilizing algorithms from our message passing model to the message-driven model (assuming that there is always one message in some link in any state or the existence of a time-out mechanism). This transformation is the composition of the following two transformations: the transformation from our message passing model to the shared memory model with single-reader, single-writer registers that will be presented in Section 4.2, and the transformation from the shared memory model with single-writer single-reader registers to the message-driven model given by Dolev, Israeli, and Moran [15] and presented in Section 3.3.

Self-stabilizing algorithms designed for our message passing model may cause flooding of the links with messages. For example, consider a processor, \( P \), whose algorithm does not contain an unbounded loop with a receive(⊥) operation in which it does not send any message. Now, assume that \( P \) is very fast compared to all of its neighbours. Suppose that the queues from which \( P \) receives messages are empty. Then, \( P \) will perform many receive(⊥) operations and consequently, send many messages until some neighbour finally sends it a message.
Chapter 3

Related Work

In this section, we describe the results in the literature that are related to the work in this thesis. We discuss known transformations of general algorithms between the shared memory model and the message passing model. Next, we briefly describe known general tools for transforming arbitrary algorithms to self-stabilizing ones. Applying these tools to the non-self-stabilizing transformed algorithms, we get self-stabilizing versions of them. However, the resulting algorithms that we get with this method have some disadvantages which we will discuss. In Section 3.3, we present a known transformation of self-stabilizing algorithms from the shared memory model to the message-driven model. We modify this transformation in Section 4.4 to get a transformation from the shared memory model to our message passing model. In particular, we modify the token-passing algorithm for two processors used by this transformation and, for that reason, we present token-passing algorithms for the message-passing model in Section 3.4. There are no self-stabilizing transformations from the message passing model to the shared memory model in the
literature that I am aware of. In Chapter 4, we construct a transformation of self-stabilizing algorithms from the message passing model to the shared memory model, using an algorithm that solves a special kind of token-passing, that we call the Round-Robin algorithm. We present a survey of self-stabilizing token-passing algorithms that work in the shared memory model, in order to compare them with our Round-Robin algorithm. Finally, in Section 3.6, we present some lower bounds in message passing models that we refer to in the chapter of future work.

3.1 General Transformations between the Message Passing Model and the Shared Memory Model

In this section, we consider transformations of algorithms between the message passing model and the shared memory model. These transformations are designed to transform general algorithms, without considering self-stabilization. As we will see in Section 3.3, these transformations do not work for self-stabilizing algorithms. Specifically, even if the original algorithm is self-stabilizing, the resulting algorithm may not be.

In transformations from the shared memory model to the message passing model, the shared variables are stored in the local memory of the processors of the message passing model. First, consider a system where no processor crashes. There is a straightforward transformation which is presented by Lynch [20], where a shared variable is stored in the local memory of exactly one processor. Any access to the shared memory will be
simulated by sending an invocation to the processor that stores the shared variable. All operations performed by the processor that sent the invocation are suspended until a response is received. A processor that receives an invocation, applies it to the shared variable and responds to the sender of the invocation. To make the resulting algorithm work correctly even if \( f \) processors crash, at least \( f + 1 \) processors should contain copies of the shared variables. In fact, in all the transformations from the shared memory model to the message passing model that we discuss below, every processor has a copy of every shared variable.

Attiya et al. [6] and Lynch [20] prove that for all transformations of algorithms from the shared memory model to the asynchronous message passing model, fewer than \( \frac{n}{2} \) of the simulating processors may crash. They prove it by contradiction, as follows. They divide the processors into two disjoint groups of at most \( f \) processors each, where \( f (\geq \frac{n}{2}) \) is the number of processors that may crash. We assume that the communication between the processors of the two groups is very slow. Then, any processor in one group should complete its operations even if it does not receive anything from the \( f \) (at most) processors of the other group, because the processor of the other group might have crashed. Now, it is possible that a processor from the first group simulates a \texttt{WRITE}(1) to a register, and after it is completed, a processor from the second group simulates a read to this register and reads a value different than 1. This is because the information about the write performed by the first processor comes very slowly to the second processor.

Bar-Noy and Dolev [9] present an easy way to get transformations between the shared memory model and the message passing models. They do this by restricting attention
to algorithms in which the processors communicate information using snapshots. In particular, they implement snapshot for the completely connected message passing model (with possible delays of incoming messages and possible reordering of messages in the links), and for the shared memory model with multi-reader, single-writer registers.

Next, we present a transformation, proposed by Attiya, Bar-Noy and Dolev [5], of algorithms from the shared memory model with multi-reader, single-writer, registers to two different message passing models: the complete network with reliable links but processor failures, and arbitrary networks with reliable processors but dynamic link failures. For an arbitrary network with faulty links, a processor is considered faulty if it cannot communicate (through any path) with a majority of the processors. Since it is required that at most \( f \) processors are faulty, where \( f < \frac{n}{2} \), then at least \( n - f > \frac{n}{2} \) processors are connected to each other.

We will describe the simulation of write and read operations. For each register, each processor locally stores a copy of the value of the register together with a time-stamp. A processor that simulates a read operation of a register \( R \) sends a request to every processor and waits until it gets a response from a majority of the processors. Any processor that receives this request, responds by sending the latest value of register \( R \) that it knows, together with its corresponding time-stamp. The value returned by the reader is the value with the maximum time-stamp among the values received. A processor that simulates a write operation to a register \( R \) updates its local memory with the new value and a new time-stamp that corresponds to this register. Then, it sends a message containing the new value of the register, together with a new time-stamp, to every processor and waits
until it gets a response containing an acknowledgment from a majority of the processors. Every processor that receives a message with a value of a register and a larger time-stamp than the time-stamp corresponding to its local copy of the register, updates its local value of this register and the corresponding time-stamp. The time-stamps they use are bounded. Attiya [3] improved the space complexity and message complexity of this transformation.

Now, we present a transformation of algorithms from the shared memory model to a more powerful message passing model, the totally ordered reliable broadcast model proposed by Attiya and Welch [7, 4]. There are two possible external operations available in the totally ordered broadcast model: receive a message, and broadcast a message. When a processor broadcasts a message, it sends it through a channel accessed by all the processors. This channel has the property the messages sent will be received by all the processors in the same order. The totally ordered reliable broadcast model is similar to the shared memory model because all the messages in the channel are eventually received by every processor. This resembles the situation where all the processors can read the same information stored in a multi-reader register.

The main idea of the transformation is to specify an order of the read and write operations that will be known to all the processors. Such an order can be specified by the messages received by the channel. Based on this order, every processor is able to simulate the computation performed by all the processors using the registers in its local memory.
We will describe in more detail how the read and write operations are simulated. Initially, all processors have, in their local memory, a copy of the registers in the system with the same initial values. A processor that simulates a read operation, broadcasts a message with a read request, and the identifier of the register. Then, it keeps receiving messages from the channel, until it receives its own message back. If it receives a message with a write request it updates the appropriate register in its local memory with the new value determined by the message received. When it receives its own request to read, then it returns the value of the register based on the information in its local memory. Similarly, when a processor simulates a write operation it broadcasts a message containing a write request, the identifier of the register to be written, and the new value. Then it keeps receiving messages from the channel, and updating the registers with the values received in the write messages, until it receives its own message back. Finally, it updates its local copy of the register that it wants to write to, and returns acknowledgment.

Attiya and Welch [7, 4] give two transformations of algorithms from the totally ordered reliable broadcast mode1 to the message passing model (that supports receive and send steps but not broadcast). In the first transformation, one special processor collects all messages to be broadcast, assigns a sequence number to each of them, and sends them to all processors. Each processor simulates receive operations according to the sequence numbers of the messages received. Each processor simulates broadcast of a message by sending this message to the special processor.

In the second transformation, each processor $P_i$ stores in its local memory an array $T_i$ of (unbounded) time-stamps (one for each processor) and an (unbounded) set $Pending$,
of messages. When processor $P_i'$ simulates broadcast of a message $m$, it increments its local time-stamp $T_i(i)$. adds the triple $< T_i(i), i, m >$ to its $Pending$ set, and sends this triple to the other processors. When a message $< t, i, m >$ is received by some processor $P_j'$, this processor will update $T_j(i)$ in its local memory with the new time-stamp $t$ and add this triple to its $Pending_j$ set. Finally, if $t > T_j(j)$, then $P_j'$ sets $T_j(j)$ to $t$ and will send a message containing the new time-stamp $t$ and its identifier $j$ to every processor. to inform them about the update. When some other processor $P_k'$ receives $< t, j >$, it will update $T_k(j)$ with the value $t$.

To simulate a receive operation of the totally ordered broadcast model, a processor $P_j'$ finds the lexicographically smallest triple $< t, i, m >$ in its local $Pending_j$ set. If $t < T_j(h)$ for all $h$, then $P_j'$ simulates receive by returning $m$ as the received message and removes $< t, i, m >$ from its $Pending_j$ set. Otherwise $P_j'$ cannot yet simulate a receive operation.

To see why this transformation works correctly, first note that it is not possible for some triple $< t, i, m >$ to remain in $Pending_k$ forever. Otherwise, there is some $j$ such that $T_k(j) < t$ forever. Because links are reliable, this can only happen if $P_j'$ stops updating its time-stamp $T_j(j)$, leaving it at some value smaller than $t$. Since $P_j'$ will eventually receive the triple $< t, i, m >$, which was sent by $P_i'$ to all processors, it will eventually update its time-stamp to a value larger than $t$. This is a contradiction.

Now, we present an example to illustrate that the total order in which the messages are received (from the pending arrays) is the same for all processors. Suppose $P_i'$ has
\( \text{Pending}_1 = \{ <t_3, 3, m_3>, <t_4, 4, m_4> \} \) and suppose \( P_2 \) has \( \text{Pending}_2 = \{ <t_4, 4, m_4> \} \), where \( t_3 \leq t_4 \). We will show that \( P'_1 \) will not be able to simulate \( \text{receive}(m_4) \) until it receives \( <t_3, 3, m_3> \) from \( P_3 \). Since \( P'_2 \) did not receive \( <t_3, 3, m_3> \) from \( P_3 \) yet, and links do not reorder messages, \( T_2(3) < t_3 \). Then \( T_2(3) < t_4 \), so \( P'_2 \) cannot simulate \( \text{receive}(m_4) \).

If we compose a transformation from the shared memory model to the totally ordered reliable broadcast model described with a transformation from the totally ordered reliable broadcast model to the message passing model, we get a transformation from the shared memory to the message passing model. This transformation is not as general as the one presented by Attiya, Bar-Noy, and Dolev [5], because it does not deal with processor or link failures. Furthermore, the time-stamps used by Attiya, Bar-Noy, and Dolev [5] are bounded.

A transformation from the message passing model to the shared memory model is presented by Lynch [20]. For every queue \( L_{i,j} \) of the message passing model, she uses a single-reader, single-writer register \( M_{i,j} \), writable by \( P'_i \), and readable by \( P'_j \). In its local memory, processor \( P'_i \) will contain the sequences of messages that are stored in \( M_{i,j} \), and a prefix of the messages that are stored in \( M_{j,i} \) for all \( j \in \text{NBR}_i \). Processor \( P'_i \) simulates the \( \text{send}(msg) \) operation to queue \( L_{i,j} \) by appending \( msg \) to its local copy of \( M_{i,j} \), and writing the resulting sequence of messages to \( M_{i,j} \). A receive operation from \( L_{j,i} \) is simulated in the following way: Processor \( P'_i \) reads \( M_{j,i} \); if the local copy of \( M_{j,i} \) is a proper prefix of \( M_{j,i} \), then \( P'_i \) updates its local copy of \( M_{j,i} \), increasing the length of the prefix in its local copy, and returns the last message. Otherwise, \( M_{j,i} \) and its local copy
A message is never removed from a register. Thus, the registers may need unbounded space. Because the processors keep copies of the registers, they also need unbounded memory space.

### 3.2 Transformations of General Algorithms to Self-Stabilizing Algorithms

There are tools for transforming non-self-stabilizing algorithms to self-stabilizing ones. We present these techniques and then discuss the problems that they may cause.

Katz and Perry [17] give a tool for extending an algorithm for the message-driven model to an algorithm in the same model that stabilizes starting from any initial state where at least one processor will next perform a send operation (note that this is how they deal with the communication deadlock, as explained in Section 2.6). The idea is to interleave operations of a self-stabilizing control program with operations of the original algorithm. The self-stabilizing control program takes snapshots of the global state, tests whether these snapshots correspond to an unsafe state, and, if so, the state is reset to a default safe state. There is a specific processor that collects all the information needed to coordinate the detection of unsafe states. It sends messages triggering a reset, if needed. The self-stabilizing algorithms created by this method have bad message complexity because many messages are sent to perform snapshots.
The self-stabilizing algorithms created by this method may be more complicated than necessary. For example, the first of the four self-stabilizing mutual exclusion algorithms presented by Lamport [18] is a simple extension (only one check is added) of a non-self-stabilizing algorithm for the same task. Instead, if we apply the method above, we will get a self-stabilizing algorithm with a mechanism for taking snapshots.

Ideally, when applying the method described above, we have a characterization of all possible unsafe states or all possible safe states of an algorithm. This might be difficult to obtain. If the transformed algorithm recognizes only a subset of the safe states, then unnecessary resets may occur.

Another problem is that the self-stabilizing algorithms created by the method above may be inefficient. This may happen if the frequency with which snapshots of the global or local state of the system are triggered is not well chosen. If we adjust the system to take snapshots very often, then overhead comes from the control algorithm that takes more snapshots than necessary. However, if snapshots are taken too infrequently, the algorithm may run for a long time without performing correct computation, since the system is slow to detect that the algorithm is running starting from an unsafe state.

Awerbuch, Patt-Shamir, and Varghese [8] present a transformation that takes as input a restricted type of message passing algorithm and produces a self-stabilizing algorithm that solves the same problem. By a restricted type of algorithm, we mean that it should satisfy the following property: if a global state is unsafe, then there exists a subsystem consisting of some pair of neighbouring processors and their common link which is also
unsafe. They interleave the original algorithm with a checking/correcting algorithm at each edge of the communication graph. The checking/correcting algorithm is triggered by a timer at one of the two processors at the endpoint of the edge. Similar results are presented for the shared memory model. In particular, Varghese et al. [21] showed that a local detection mechanism can be applied to algorithms in the shared memory model if the graph of neighbouring processors is a tree.

Note that there are problems for which it is impossible to design algorithms that allow local detection of an unsafe state, unless the environment of the processors contain information about the global state of the system. For example, consider a token-passing algorithm, such that every processor contains information about the existence of a token in its neighbourhood. In a state where multiple tokens exist, local detection will not work correctly, unless at least two of these tokens exist in the same neighbourhood of some processor. Although it is argued that local detection and correction gives faster self-stabilizing algorithms, this is not always the case: Local correction of an unsafe state might create more problems in other parts of the system that have to be corrected eventually.
3.3 Transformations of Self-Stabilizing Algorithms from the Shared Memory Model to the Message-Driven Model

The transformation from the message passing model to the shared memory model proposed by Lynch [20] and described in Section 3.1 does not preserve self-stabilization. To see why, consider processor $P_i'$ that simulates a receive operation starting from a state where its local memory contains a sequence of messages that is not a prefix of the sequence in register $M_{j,i}$. The algorithm described in [20] does not deal with this case. It is easy to modify this transformation to one that transforms self-stabilizing algorithms from the message passing model to the shared register model. The main modification is that after $P_i'$ reads $M_{j,i}$, it updates its local memory with the sequence of messages in $M_{j,i}$. Eventually, every processor performs this update, and then we can apply Lynch's transformation. However, Lynch's approach requires registers of unbounded size and unbounded local memory in the processors.

The transformations from the shared memory model to the message passing model, presented in Section 3.1, are not self-stabilizing since they rely on the initialization of the variables. Extending these transformations to preserve self-stabilization may introduce unnecessary complexity, because they deal with processor crashes which are not relevant for self-stabilization.

Dolev, Israeli, and Moran [14, 15] present a transformation of self-stabilizing algo-
rithms from the shared memory model with single-reader single-writer registers to the message-driven model. The single-reader, single-writer register \( M_{i,j} \) that is writable by \( P_i \) is stored in the local memory of \( P_i' \). The transformed algorithm for \( P_i' \) is the fair composition of self-stabilizing 2-processor token-passing algorithms (i.e. \( P_i' \) shares a token with each of its neighbours, separately) and a modified version of the original algorithm. In addition to the information needed to perform token circulation, every message sent from \( P_i' \) to \( P_j' \) contains the value of the register \( M_{i,j} \). In the modified version of the original algorithm, a write to \( M_{i,j} \) performed by processor \( P_i \) is simulated by an internal operation, WRITE, performed by \( P_i' \) on its local copy of the register. Specifically, WRITE updates the local copy of the register with its new value. A read of \( M_{i,j} \), performed by \( P_j \), is simulated by having \( P_j' \) perform the algorithm READ: \( P_j' \) waits until receiving the token it shares with \( P_i' \); it gives the token back to \( P_i' \) and waits until receiving the token from \( P_i' \) again. Finally, \( P_j' \) returns the value of \( M_{i,j} \) from the message that gives it the second token.

To prove correctness of the transformation, they show that there is a linearization of the READ and WRITE procedures such that, if READ is performed on a register, it will return the value that was last written to this register. A WRITE is linearized exactly when its single operation is performed. They linearize a READ performed by \( P_j' \) immediately after \( P_i' \) sends the message that contains the second token received by \( P_j' \). To see why this is a correct linearization, observe that every message sent by \( P_i' \) contains the value of \( M_{i,j} \) at the time it is sent. During READ, two tokens are received instead of one in order to ensure that the linearization point of a READ occurs after it has started.
3.4 Self-stabilizing Token-Passing Algorithms for Message Passing Systems with Two Processors

A token-passing algorithm ensures that eventually no more than one processor holds the single token at any time in an execution, and each processor holds the token infinitely many times. We will present some self-stabilizing token-passing algorithms proposed for different message passing models that have two processors.

Angluin [2] showed that the two processors this system should perform different algorithms to solve the token-passing algorithm. Otherwise, we can create an execution where at the end of each round, the processors are in the same state. We can initialize them to the same state. Since they will receive the same messages, send the same messages, and execute the same algorithm, they will move to the same states. Then, if at the end of some round a processor has the token then the other processor has it as well. Similarly, if at the end of some round one of the processors does not have the token, then the other processor does not have it either, which violates the token-passing task.

Next, we present a self-stabilizing token-passing algorithm by Dolev, Israeli, and Moran [14, 15], that is based on the token-passing algorithm of Dijkstra [12]. This algorithm is designed for the message-driven model with unbounded link capacity. Every message contains an unbounded time-stamp. Both processors store such a time-stamp in their local memory. One of the processors copies the contents of every message it receives to its local memory and then sends a message to the other processor containing
the new values. It gets the token whenever it receives something different from its local
time-stamp. When the other processor receives a message it compares it with its local
time-stamp. If the time-stamp received is larger or equal to its local time-stamp, a token
has arrived. it increments its received value. stores it. and sends this new value to the
other processor. Otherwise. it sends the unchanged value of its time-stamp. Correctness
follows from the fact that eventually the value of the time-stamp of the second processor
will be larger than all the values in the system.

Dolev [14] presents a token-passing algorithm for a message-driven model assuming
a bound. $B$, on the number of messages in both links. This algorithm works exactly
the same way as the one presented above. with the difference that the time-stamp is
incremented modulo $B + 1$ by the second processor. and this processor gets the token
only when the time-stamp received is the same as its local time-stamp. Since the time-
stamps are bounded. the processors need bounded space to store them. This algorithm
works correctly because. eventually. the time-stamps sent will not exist in any message
in the links.

for the message-driven model with unbounded link capacity. This algorithms uses ideas
from the alternating-bit protocol [1]. Instead of a time-stamp. each message contains a
variable. called colour. that can take three possible values. One of the processors. $P_1'$
behaves as in the previous results. (When it receives a message. it stores the colour and
sends the message back. It gets the token when it receives a different colour than the one
stored in its local memory). When the other processor $P_2'$ receives a message. it compares
the colour of the message with the colour stored in its local memory. If they are different, $P_2'$ send a message with the colour in its local memory. If they are the same, then a token arrives and $P_2'$ chooses a new colour, stores it, and sends a message with this colour.

To describe how a processor chooses a new colour, the authors use the following definitions. A sequence $(a_1, a_2, a_3, \ldots)$ is a **periodic sequence** if, for some positive integer $k$ and for all $i \geq 1$, $a_i = a_{i+k}$. A sequence is **eventually periodic** if it has a suffix which is periodic. An **aperiodic sequence** is a sequence that is not eventually periodic.

The colours chosen by $P_2'$ form an aperiodic sequence. To see why this is necessary, we present an example where $P_2'$ chooses colours that form a periodic sequence and both processors infinitely often receive tokens simultaneously. Consider the execution starting from a state where both $P_1'$ and $P_2'$ have local colour $c_1$. In link $L_{2,1}$ there are two messages with colours $c_2, c_1$, and in link $L_{1,2}$ there is a single message with colour $c_1$. We schedule both processors to perform an operation. Then $P_1'$ will update its colour to $c_2$ and send it to $P_2'$. Concurrently, $P_2'$ will receive a message with colour $c_1$, create the new colour $c_2$, and send it to $P_1'$. We again schedule both processors with $P_2'$ choosing colour $c_1$. The state reached is the same as the initial state. Note that starting from this state, both processors received a token simultaneously. If we let $P_2'$ create colours $c_1$ and $c_2$ alternately, and schedule $P_1'$ and $P_2'$ to perform operations as above, they will continue receiving tokens simultaneously. In the paper the authors show that aperiodicity is necessary condition for correctness of the algorithm.
Although there are some randomized solutions to the token-passing problem for a system with two processors, they are beyond the scope of this thesis.

3.5 Self-Stabilizing Token-Passing Algorithms in Shared Memory

Most of the early results give self-stabilizing algorithms for the composite atomicity model, a special case of shared memory model. Dijkstra [12] introduced self-stabilization and presented a token-passing algorithm for the composite atomicity model, for a ring with one distinguished processor. Burns and Pachl [11] present a self-stabilizing token-passing algorithm for the composite atomicity model for a ring with prime number of identical processors, all of which execute the same algorithm. They also prove that there is no uniform self-stabilizing token-passing algorithm on a ring with a composite number of processors.

Brown, Gouda, and Wu [10] present a token-passing algorithm for the composite atomicity model on a chain of processors. They achieve the property that if an operation is applicable, it remains applicable until it is performed, regardless of any other operations performed. This property, which does not hold for Dijkstra’s self-stabilizing token-passing algorithm, makes hardware implementation easier. In particular, they implement the above system of processors as a delay-insensitive circuit and they use flip-flops to store the state of the system.
Dolev, Israeli, and Moran [14, 13] present a token-passing algorithm for the shared memory model (with atomic read/write operations). This is achieved by composing a self-stabilizing spanning tree algorithm with a variant of Dijkstra’s mutual exclusion algorithm for the shared memory model, applied to the virtual ring that an Euler tour defines on the spanning tree. The algorithm for the spanning tree needs $4k\delta$ rounds to stabilize, where $k$ is the maximum distance of a node from the root and $\delta$ is the maximum degree of a node. The token-passing algorithm on a ring stabilizes in $O(n^2)$ rounds. Hence, the composition of these two algorithms stabilizes in $O(n^2)$ rounds.

3.6 Lower Bounds for Self-Stabilization

We will briefly discuss lower bounds on the resources used by self-stabilizing algorithms. Dolev, Israeli, and Moran [15] prove that infinitely many safe states of a message-driven message passing model are required to get self-stabilizing solutions of a task in the weak exclusion class. Specifically, if there is an execution that contains the same state twice, then there is another execution that contains this state infinitely many times and never stabilizes. Therefore, all the states in an execution of a self-stabilizing algorithm are distinct. For infinite executions, the set of safe states must be infinite. The lower bound also holds for the message passing model used in this thesis.

The proof of the lower bound needs links with unbounded capacity, because it is based on the construction of a state by adding some messages to the links. This will not be possible if the sequence of messages added causes the capacity of the link to be exceeded.
In fact, the lower bound does not hold for the message-driven model with bounded link capacity. For example, consider Dolev's self-stabilizing token-passing algorithm for the message-passing model with bounded link capacity, presented in Section 3.4. (Recall that token passing is in the weak exclusion task.)

Gouda and Multari [16] define a system where a send operation is triggered by an internal operation or a receive operation. A receive operation is performed only if there is a message in the incoming link of the processor that performs it. They prove that any self-stabilizing algorithm designed for this message passing model must use time-outs and must have an infinite number of safe states. To prove this, they assume that there is a self-stabilizing algorithm with finitely many safe states. As a result, there is a bound on the maximum number of messages in the links for the safe states. To derive a contradiction, they create an infinite execution where this bound is exceeded infinitely many times. When this happens, the states reached are not safe, and thus, the algorithm does not stabilize.
Chapter 4

Transformations

In this chapter, we present a transformation of self-stabilizing algorithms from the asynchronous message passing models (our model and the message-driven model) to the asynchronous shared memory model that uses atomic read/write registers. To do this, we use an algorithm that achieves Round-Robin execution of the critical section of the processors of the systems. Then, we modify the above transformation and create a new one that transforms self-stabilizing algorithms from the synchronous message passing models to the asynchronous shared memory model. Finally, we modify the transformation of self-stabilizing algorithms from the shared memory model to the message-driven model, given by Dolev, Israeli and Moran [15] and presented in Chapter 3, so that it transforms self-stabilizing algorithms from the shared memory model to our message passing model.
4.1 Round-Robin Algorithm

In this section, we will present an algorithm that is used in the next section (4.2) for the transformation of self-stabilizing algorithms from the message passing model to the shared memory model. This algorithm solves a special form of mutual exclusion, where the processors enter the critical section in a round-robin way. That is why we call this algorithm the Round-Robin algorithm. The processors enter the critical section in a round-robin way, using a locking mechanism based on a binary tree.

4.1.1 Binary Tree Structure

We define a strict binary tree, $T$, whose leaves are labeled by the processors in increasing order of their identifiers. Each internal node, $n_0$, is labeled by a pair of processors, $< P', P'' >$. Processor $P'$ has the minimum identifier among the processors that label the leaves of the left subtree, $T_l$, rooted at $n_0$. Processor $P''$ has the minimum identifier among the processors that label the leaves of the right subtree, $T_r$, rooted at $n_0$. Note that the identifier of $P'$ is smaller than the identifier of $P''$. Since $n$ (the number of the processors in the system) is known to the processors, the structure of the tree is also known.

Since $P'_i$ labels only one leaf, it occurs as part of the label of at most one node at any given depth. If we look at the levels going up the tree, then processor $P'_i$ (for $i \neq 1$) last labels a node as the second component of the label.

Observation 4.1.1. For each $i$, such that $1 < i \leq n$, there is exactly one $k_i \in \{1, \ldots, i - 1\}$ such that $< P'_{k_i}, P'_i >$ labels a node.
Let $N_i$ denote the set $\{j | P_i^j, P_j^j \text{ labels a node}\}$, and let $N_i(h)$ denote the $h$th smallest element of $N_i$. Note that $N_i$ does not include $k_i$. Let $N'_i = N_i$ if $i = 1$, otherwise let $N'_i = N_i \cup \{k_i\}$.

### 4.1.2 Round-Robin Algorithm

The Round-Robin algorithm that we describe in this subsection appears in Figure 4.1. For each non-leaf node with label $< P_i^j, P_j^j >$, our algorithm uses a 1-bit multi-reader, multi-writer register, $C_{i,j}$, readable and writable by $P_i^j$ and $P_j^j$. The processors take turns owning a node. We say that **processor $P_i^j$ owns the node** when $C_{i,j} = 0$ and **gives the node to $P_j^j$** by writing 1 to $C_{i,j}$. Similarly, **$P_j^j$ owns the node** when $C_{i,j} = 1$ and it **gives the node to $P_i^j$** by writing 0 to $C_{i,j}$.

**Observation 4.1.2.** In any state, no two processors can own the same node.

Each processor $P_i^j$ tries to own the nodes shared with the processors with identifiers in $N'_i$, in increasing order, except that $k_i$ is last when $i \neq 1$. For each of these nodes, first $P_i^j$ gives the node to the other processor and then waits to get the node back. After $P_i^j$ owns all the nodes it labels, it can enter the critical section. At that point, we say that $P_i^j$ **owns the tree**.

Note that entry to the critical section happens when $pc_i = crit$. After leaving the critical section, processors re-enter the trying section.

In Figure 4.2, we present a diagram of the possible values of the program counter of a processor can have when it executes the algorithm of Figure 4.1. Every edge linking two
neighbouring circles corresponds to a transition. Note that \( pc_i \) will never get the values \( \text{readmin} \) or \( \text{writemin} \). Also \( pc_i \) where \( i \) is even will only get one of the values \( \text{readmin} \), \( \text{writemin} \), and \( \text{crit} \).

### 4.1.3 Properties of the Round-Robin Algorithm

We will prove correctness of the self-stabilizing Round-Robin algorithm, presented in the previous section. First, we give some useful observations. The only way a processor \( P' \) can loop in a state is by performing a read operation. There are two different types of read operations that may cause this. The first type consists of \( \text{READ}_{h} \), with response 1 where \( 1 \leq h \leq |N_i| \). In this case, \( P'_i \) is waiting for the ownership of the node labeled by \( < P'_i, P'_{N_i(h)} > \). The second type consists of \( \text{READ\ MIN} \), with response 0. In this case, \( P'_i \) is waiting for the ownership of the node labeled by \( < P'_k, P'_i > \). We say that \( P'_i \) is **blocked** if it executes the same read operation forever.

**Lemma 4.1.3.** *(Deadlock freedom)* There is no state in any execution of the Round-Robin algorithm where all processors are blocked.

**Proof.** Assume there is an execution in which every processor is blocked. That means that each processor executes the same \( \text{READ}_{h} \) or \( \text{READ\ MIN} \) operation forever. Specifically, for every \( i \), there is a \( j \) such that either \( C_{i,j} = 1 \) or \( (i > 1 \text{ and } C_{j,i} = 0) \) forever. Consider the lowest level, \( l \), of the tree where there is a register \( C_{i,j} \) with value 1. Since processor \( P'_j \) is blocked, there exists \( g \) such that \( C_{j,g} = 1 \). The node labeled by \( < P'_j, P'_g > \) belongs to the right subtree of the node labeled by \( < P'_i, P'_j > \). Therefore the register \( C_{j,g} \) corresponds to a node at a lower level than \( l \). This contradicts the choice of \( l \). Therefore.
all registers $C_{i,j}$ have value 0 in the tree. But this implies that $P'_i$ is not blocked, which
contradicts the assumption that every processor is blocked. \qed

**Lemma 4.1.4.** (*Lockout freedom*) Every processor will enter the critical section infinitely
many times.

*Proof.* By the algorithm, if a processor never enters the critical section, it has to be
blocked in some read operation. First, suppose that there is a processor blocked in a
$READ_h$ operation, for some $h$. Consider the deepest node $n_0$ that is partially labeled by
some processor $P'_i$ blocked in a $READ_h$ operation. Let $j = \mathcal{N}_i(h_i)$. Then, throughout
the execution $C_{i,j} = 1$. Therefore, processor $P'_j$ cannot be blocked in the $READMIN$
operation. But $P'_j$ has to be blocked in some node because, otherwise, it would execute
$WRITEMIN(C_{i,j}, 0)$, setting $C_{i,j} = 0$. As a result, $P'_j$ has to be deadlocked in another
node, executing some operation $READ_{h_j}$. Processor $P'_j$ labels nodes in the right subtree
rooted at $n_0$. We conclude that $P'_j$ is blocked in a descendant, $n_1$, of $n_0$, which contradicts
the choice of $n_0$.

Now, we will prove by contradiction that no processor will be blocked in the $READMIN$
operation. Consider the least deep node, $n_2$, labeled by $< P'_{k_i}, P'_i >$, such that $P'_i$ is
blocked in the $READMIN$ operation. Then, throughout the execution, $C_{k_i,i} = 0$. From
the previous part of this proof, we know that $P'_{k_i}$ is not blocked at any $READ_h$ operation,
where $1 \leq h \leq |N_{k_i}|$. But processor $P'_{k_i}$ has to be blocked at some node; otherwise, it
would eventually set $C_{k_i,i}$ to 1. As a result, $P'_{k_i}$ has to be blocked at another node, $n_3$,
executing the operation $READMIN$. Node $n_3$ is an ancestor of $n_2$, which contradicts
the choice of $n_2$. 


We conclude that every processor executes all the operations of the Round-Robin algorithm infinitely often. As a result, every processor enters the critical section infinitely often.

We say that a processor $P'_i$ has executed a cycle of the algorithm starting from a state $s'$, if it reaches a state $s''$ with the following properties: $pc_i$ in $s'$ is equal to $pc_i$ in $s''$ and during the execution fragment that starts with $s'$ and ends with $s''$. $P'_i$ executed the operations of its critical section at least once. We say that the property AllCompleteCycle holds in a state of an execution that is reached after every processor has executed at least one cycle of the algorithm. Note that if AllCompleteCycle holds in some state of an execution, it holds for all subsequent states in that execution.

Since the Round-Robin algorithm is deadlock free and lockout free, every processor will execute a cycle of the Round-Robin algorithm infinitely many times.

**Observation 4.1.5.** Starting from any state, a state where AllCompleteCycle holds is reached eventually.

Now, we will prove that a state for which AllCompleteCycle holds is a safe state for the Round-Robin task. First, we give some observations and prove some preliminary lemmata.

Consider any state $s'$ reached after $P'_i$ has executed a cycle. We consider the following possibilities for $pc_i$ in state $s'$. When $pc_i = read_h$. $P'_i$ has read value 0 from $C_{i,N_i(j)}$ for $j \in \{1, \ldots , h - 1, h + 1, \ldots , |N_i|\}$ since last writing 1 to $C_{i,N_i(j)}$. If $i \neq 1$, then $P'_i$ has also read value 1 from $C_{k_{i,i}}$ since last writing 0 to $C_{k_{i,i}}$. When $pc_i = readmin$, $P'_i$ has read value 0 from $C_{i,N_i(j)}$ for $j \in \{1, \ldots , |N_i|\}$ since last writing 0 to $C_{k_{i,i}}$. 
Finally, when $pc_i = \text{write}_h$ or $pc_i = \text{writemin}$, then $P_i'$ has read value 0 from $C_{i,N_{i}(j)}$ for $j \in \{1, \ldots , |N_i|\}$ since last writing 1 to $C_{i,N_{i}(j)}$, and if $i \neq 1$, has read value 1 from $C_{k_{1,i}}$ since last writing 0 to $C_{k_{1,i}}$. Since $P_{N_{i}(j)}'$ only writes 0 to $C_{i,N_{i}(j)}$. $P_i'$ owns node $< P_i', P_{N_{i}(j)}'>$ if it has read value 0 from $C_{i,N_{i}(j)}$ since last writing 1 to $C_{i,N_{i}(j)}$. Similarly, $P_{k_{1,i}}'$ only writes 1 to $C_{k_{1,i}}$. Therefore, $P_i'$ owns the node labeled by $< P_{k_{1,i}}', P_i'>$ if it has read 1 from $C_{k_{1,i}}$ since last writing 0 to $C_{k_{1,i}}$.

**Observation 4.1.6.** In a state where AllCompleteCycle holds, if processor $P_i'$ does not own node $< P_i', P_{N_{i}(k_{1,i})}'>$ then $pc_i = \text{read}_h$. and if $P_i'$ does not own node $< P_{k_{1,i}}', P_i'>$ then $pc_i = \text{readmin}$.

Processor $P_i'$ partially labels $|N_i'|$ nodes, so we get the following observation.

**Observation 4.1.7.** If $P_i'$ has executed a cycle, then it owns at least $|N_i'| - 1$ nodes.

Before we prove mutual exclusion, we give some definitions. We denote by $\text{PATH}_i$, the simple path in the tree $T$ that starts from the leaf labeled by $P_i'$ and goes up to the root. For $i \neq 1$. $\text{FirstInPath}_{i,l}$ is the first node in $\text{PATH}_i$ partially labeled by $P_i'$.

**Lemma 4.1.8.** After AllCompleteCycle holds. if $P_i'$ is in the critical section, processor $P_i'$ does not own $\text{FirstInPath}_{i,l}$.

**Proof.** Suppose $P_i'$ is in the critical section. We will prove the lemma by contradiction. Assume that the lemma does not hold. Let $n_0$ be the first node in $\text{PATH}_i$, such that for some $l \neq i$, $P_i'$ owns $n_0=\text{FirstInPath}_{i,l}$. Suppose $n_0$ is also labeled by $P_{k_{1}}'$. Since processor $P_{k_{1}}'$ does not own $n_0$. Observation 4.1.7 implies that $P_{k_{1}}'$ owns all the other nodes that it partially labels.
Since \( n_0 = \text{FirstInPath}_{i,k} \), processor \( P_i' \) does not partially label any earlier node in PATH. By the construction of tree \( T \), every child is partially labeled by exactly one of the processors that label its parent. Thus, \( P_k' \) has to label the child of \( n_0 \) in PATH. As a result, \( \text{FirstInPath}_{i,k} \) is a descendant of \( n_0 \). Since \( P_k' \) owns \( \text{FirstInPath}_{i,k} \) this contradicts the choice of \( n_0 \).

Lemma 4.1.9. (Mutual exclusion) No two processors have their program counters equal to \( \text{crit} \) in any state of an execution where \( \text{AllCompleteCycle} \) holds.

Proof. We will prove this by contradiction. Assume that \( pc_i = pc_j = \text{crit} \). Without loss of generality, let \( i < j \). Because of the \( \text{AllCompleteCycle} \) property, \( P_i' \) and \( P_j' \) have all the tokens in all nodes that they partially label. That means that \( P_i' \) and \( P_j' \) cannot label the same node. Let \( n_0 \) be the first node in the intersection of PATH\( _i \) and PATH\( _j \). Suppose it is labeled by \( < P_h', P_k' > \). Since \( i < j \), processor \( P_i' \) labels a leaf in the left subtree, \( T_{left} \), rooted at \( n_0 \), and processor \( P_j' \) labels a leaf in the right subtree, \( T_{right} \), rooted at \( n_0 \).

Since \( P_i' \) labels a leaf in \( T_{left} \), PATH\( _i \) contains the left child of \( n_0 \). But the left child of \( n_0 \) is partially labeled by \( P_h' \). Therefore, \( n_0 \neq \text{FirstInPath}_{i,k} \). Similarly, \( n_0 \neq \text{FirstInPath}_{j,k} \). Then by Observation 4.1.7 and Lemma 4.1.8, both \( P_h' \) and \( P_k' \) own node \( n_0 \). This is impossible, because of Observation 4.1.2.

Lemma 4.1.10. (Round-Robin execution of critical section) If \( \text{AllCompleteCycle} \) holds, after \( P_i' \) exits the critical section, \( P_{(i mod n)+1}' \) will be the next processor to execute the critical section.

Proof. Consider a state \( s \) in which \( P_i' \) is in the critical section. In this proof, we consider the value of \( pc_{(i mod n)+1}' \) in state \( s \). Then, we will prove that after \( P_i' \) exits the critical
section, $pc_{(i \mod n)+1}$ is the next program counter to get the value $\text{crit}$.

Starting from state $s$ there is no sequence of operations performed by processors that do not label nodes in $\text{PATH}_i$ that would lead them to enter the critical section. This is because otherwise they could enter the critical section concurrently with $P'_i$ violating mutual exclusion. After $P'_i$ exits the critical section but before the next processor enters, processors that label a node in $\text{PATH}_i$ will change the registers corresponding only to nodes of $\text{PATH}_i$. Therefore, the processors not labeling nodes in $\text{PATH}_i$ will still not be able to enter the critical section. We look at the change in the registers corresponding to the nodes in $\text{PATH}_i$ that happens after $P'_i$ exits the critical section and prove that this leads to $P'_{(i \mod n)+1}$ being the next to enter the critical section.

Suppose $\text{PATH}_i$ consists of the nodes $n_0, n_1, \ldots, n_k$, where node $n_0$ is the leaf labeled by $P'_i$.

First, consider the case where $i = n$. So every node in $\text{PATH}_i$ is a right child. Then, by the construction of the binary tree, the first component of the label of node $n_i$ is the same as the second component of the label of $n_{i+1}$. Therefore, if $< P'_i, P'_{i+1} >$ labels some node $n_i$ in $\text{PATH}_i$, then $n_i \neq \text{FirstInPath}_{i+1}$. From Observation 4.1.7 and Lemma 4.1.8, $P'_{i+1}$ owns node $n_i$. Thus, each of the nodes $n_1, \ldots, n_k$ is owned by the processor that is the second component of its label. By Observation 4.1.6, if $< P'_i, P'_{i+1} >$ labels node $n_i$ in $\text{PATH}_i$, then $pc_{i+1} = \text{read}_{n_{i+1}}$.

When $P'_n$ exits the critical section, it sets $pc_n = \text{writemin}$, performs the WRITEMIN($C_{k,n}, 0$), and then continuously performs READMIN($C_{k,n}, 0$). By an easy induction we get that eventually each of the processors $P'_i$ that is the second component of the label of a node in $\text{PATH}_i$ performs WRITEMIN($C_{k_i}, 0$) and then continuously performs READMIN($C_{k_i}, 0$).
Let $s'$ be the state reached after these operations are performed. Then, in $s'$, every processor that is the first component of the label of a node in PATH, owns that node. If $n_k$ is labeled by $< P_i', P'_j >$, then after $P'_j$ performs WRITEMIN($C_{i,j}, 0$), $P'_i$ will perform READ$_{N_i}(C_{1,N_i(N_i)}, 0)$ and get $pc_i = crit$ as desired.

The second case is when $n_0$ is a left child. Then, $n_1$, the parent of $n_0$, is labeled by $< P'_i, P'_i+1 >$. Since processor $P'_i$ owns node $n_1$ in state $s$, $P'_i+1$ does not. Then, by Observation 4.1.6, $pc_{i+1} = readmin$. When $P'_i$ exits the critical section, it sets $pc_i = write_i$, performs WRITE$_i(C_{i,i+1}, 1)$, and then continuously performs READ($C_{i,i+1}, 1$). Then, processor $P'_i+1$ will execute README($C_{i,i+1}, 1$) and enter the critical section.

The last case is when there is $0 < j < k$ such that $n_0, \ldots, n_{j-1}$ are right children and $n_j$ is a left child. Suppose $n_j$ is labeled by $< P'_j, P'_j+1 >$ and $T_{left}$ and $T_{right}$ are the left and right subtrees of node $n_{j+1}$. Processor $P'_i$ is the processor with the maximum identifier in $T_{left}$ and processor $P'_{i+1}$ is the processor with the minimum identifier in $T_{right}$. As a result, $n_{j+1}$ is labeled by $< P'_j, P'_{i+1} >$.

As in the first case, in state $s$ each of the nodes $n_1, \ldots, n_j$ is owned by the processor that is the second component of its label. By Observation 4.1.6, if $< P'_l, P'_l+1 >$ labels node $n_l$, where $1 \leq l \leq j$, then $pc_{l+1} = readh$, where $N_{l+1}(h) = l_2$. By the construction of the tree we get that $h = |N_{l+1}|$. Furthermore, since $P'_j$ owns $n_{j+1}$, $P'_{i+1}$ does not own $n_{j+1}$. Then, by Observation 4.1.6, $P'_i+1$ has $pc_{i+1} = readmin$ in $s$.

When $P'_i$ exits the critical section, it sets $pc_i = writemin$, performs WRITEMIN($C_{k,i}, 0$), and then continuously performs README($C_{k,i}, 0$). By a similar induction as in the first case, we get that eventually each of the processors $P'_i$ that is the second component of the label of any of the nodes $n_1, \ldots, n_j$ performs WRITEMIN($C_{k,i}, 0$) and then
continuously performs \textsc{Readmin}(C_{k_i}, 0). Let \( s' \) be the state reached after these operations are performed. After \( s' \), \( P'_{i_1} \) will perform \textsc{write}(C_{j_{i,i+1}}, 1). then \( P'_{i+1} \) will execute \textsc{Readmin}(C_{j_{i,i+1}}, 1), and will enter the critical section.

\[ \square \]

### 4.1.4 Complexity of the Round-Robin Algorithm

In this section we will calculate the maximum number of rounds until the Round-Robin algorithm stabilizes.

**Observation 4.1.11.** Consider an execution fragment during which a processor writes to the same multi-reader, multi-writer register twice. During this fragment the processor performs a cycle.

**Lemma 4.1.12.** If \( P'_i \) and \( P'_j \) label the same node and \( P'_i \) performs \( c \) cycles, then \( P'_j \) performs at least \( c - 2 \) cycles.

**Proof.** Assume that \( P'_i \) and \( P'_j \) label some node \( n_0 \). By the algorithm, if processor \( P'_i \) executes \( c \) cycles, processor \( P'_j \) executes at least \( c - 1 \) writes to \( n_0 \). Then, by Observation 4.1.11. \( P'_j \) performs at least \( c - 2 \) cycles.

Let \( s(n) \) denote the maximum number of operations performed by a processor during a critical section.

**Lemma 4.1.13.** In any execution of the Round-Robin algorithm, at least one write operation is performed during \( s(n) + 2 \) consecutive rounds.

**Proof.** We will prove this lemma by contradiction. Assume that during \( s(n) + 2 \) consecutive rounds in some execution of the Round-Robin algorithm, no processor performs
a write operation. Then, during these rounds, the processors will execute either read operations, or some operations of a critical section. If some processor executes some operation of a critical section during the first or the second round, then, it exits the critical section by round \( s(n) + 1 \) and performs a write by round \( s(n) + 2 \). This is because a critical section contains at most \( s(n) \) operations and the next operation performed by a processor after it exits the critical section is a write operation. As a result, all the processors execute read operations during the first two rounds. But then the processors could read these same registers forever, which contradicts deadlock freedom of the Round-Robin algorithm.

\[ \square \]

**Theorem 4.1.14.** The Round-Robin algorithm stabilizes, within \( (s(n) + 2)(4 \log_2 n + 1) \) \( n(\lceil \log_2 n \rceil - 1) \) rounds.

**Proof.** We proved in Section 4.1.3 that a state where AllCompleteCycle holds is safe for the Round-Robin task. We want to see how many rounds are needed until AllCompleteCycle holds.

If we look at any path in the tree \( T \), at least every two consecutive nodes are partially labeled by one common processor. Since we are creating a balanced tree, the maximum height of the tree is \( \lceil \log_2 n \rceil \). Therefore, the length of the simple path between the leaves labeled by \( P_i' \) and \( P_j' \) is at most \( 2 \lceil \log_2 n \rceil \). Thus, if any processor performs at least \( k = 4 \lceil \log_2 n \rceil + 1 \) cycles, then, by Lemma 4.1.12, every processor that labels a leaf performs at least \( k - 4 \log_2 n \geq 1 \) cycles and, hence, AllCompleteCycle holds. So, we want to calculate how many rounds are needed for some processor to perform at least \( k \).
cycles.

From Lemma 4.1.13. during the first \((s(n) + 2)w\) rounds, at least \(w\) writes have occurred. Then, by the Pigeonhole principle, there is some processor \(P\) that has performed at least \(\frac{w}{n}\) write operations. But any processor partially labels at most \([\log_2 n] - 1\) nodes in the tree (we exclude the leaf). So, it performs at most \([\log_2 n] - 1\) write operations per cycle. As a result, after \((s(n) + 2)w\) rounds, there is some processor \(P\) that performs at least \(\frac{w}{n([\log_2 n] - 1)}\) cycles. So, some processor \(P\) performs at least \(k = 4([\log_2 n] + 1)\) cycles after \((s(n) + 2)n([\log_2 n] - 1)(4\log_2 n + 1)\) rounds. \(\Box\)

In the case where operations of the critical section are not operations of the distributed system, \(s(n) \in O(1)\), so the Round-Robin algorithm stabilizes in \(O(n(\log n)^2)\) rounds.

4.2 Transforming Self-Stabilizing Algorithms from the Message Passing Model to the Shared Memory Model

We present a transformation of self-stabilizing algorithms from message passing models to the shared memory model that uses read/write registers. Consider an execution in the message passing model starting from an arbitrary state. If no further faults occur, all the messages sent during the execution will be received eventually. A message is received only once, because a receive operation removes it from the queue. However, in the shared memory model, a value written to a shared register may never be read, as a processor may overwrite it before it is read by any processor, or it may be read many
times. In the first case, the value will be lost. Our transformation from the message passing model to the shared memory model will ensure that every value written to a single-reader, single-writer register will be read exactly once.

The transformation is based on the following observation: Consider a system with only two processors and two single-reader, single-writer registers: each register is read by one processor and is written by the other processor. Consider an execution where repeatedly: one processor performs one read and one write and then the other processor performs one read and one write. In this execution, each processor will read the contents of the register written by its neighbour exactly once after it updates them. For general systems, we will ensure that each processor executes a sequence that contains a read from each register that can be read by the processor and a write to each register that can be written by the processor so that the following holds: If the system has $n$ processors, after processor $P'_i$ executes this sequence, processor $P'_j$ is allowed to do so. Note that the transformation we present transforms self-stabilizing algorithms from the asynchronous message passing model with the Round-Robin scheduler to the asynchronous shared memory model with an arbitrary scheduler.

A transformation from the asynchronous message passing model with the Round-Robin scheduler to the asynchronous shared memory model with no restriction imposed on the scheduler is more powerful than a transformation from the asynchronous message passing model to the asynchronous shared memory model with arbitrary schedulers for both models. To see this, consider the set, $E_{RR}$, of all possible executions of the asynchronous message passing system with Round-Robin scheduler, the set, $E_{MP}$, of all possible executions of the asynchronous message passing system with arbitrary scheduler.
and the set, \( E_{SM} \), of all possible executions of the asynchronous shared memory system with arbitrary scheduler. The Round-Robin scheduler is a special case of an arbitrary scheduler, so \( E_{RR} \subseteq E_{MP} \). The transformation from the asynchronous message passing model with Round-Robin scheduler to the asynchronous shared memory model ensures that every execution \( a \in E_{SM} \) simulates an execution \( b \in E_{RR} \). Since \( E_{RR} \subseteq E_{MP} \), it follows that \( b \in E_{MP} \). This means that the above transformation is also a transformation from the asynchronous message passing model to asynchronous shared memory model with arbitrary schedulers.

### 4.2.1 The Composed Algorithm

As we already described in the definition of transformation in Chapter 2, each processor in the message passing model is simulated by one processor in the shared memory model. We describe how we simulate the communication medium of the message passing model to the shared memory model. For every unidirectional link from \( P_i \) to \( P_j \) implemented as a queue \( L_{i,j} \) in the message passing model, we are using a single-reader, single-writer register \( M_{i,j} \) in the shared memory model that will be written by \( P_i' \) and read by \( P_j' \).

We will define the operations in the two models that we consider for this transformation. In the shared memory model, the read and write steps are defined to be atomic operations. An operation in the message passing model is a sequence of receive steps, (where a processor receives one message or \( \bot \) from each neighbour), followed by internal steps and a sequence of send steps (during which at most one message is sent to each neighbouring processor).
Next, we describe the sequence of operations in the shared memory model that sim-
ulates an operation of the message passing model. We will prove this in Section 4.2.3.
The sequence of operations is divided in two phases. The first phase is the **read phase**, 
which consists of the sequence of $d_i$ input operations: $\text{Read}_1, \ldots, \text{Read}_{d_i}$ that copy the 
values of the $M_{j,i}$ registers to the local memory of $P'_i$. for all $j \in NBR_i$. The last read 
operation $\text{Read}_{d_i}$ will end with the internal computation performed by $P_i$ as part of this 
one in the message passing model. The second phase is the **write phase**, which 
consists of the sequence of $d_i$ output operations $\text{Write}_1, \ldots, \text{Write}_{d_i}$.

In the local memory of every processor $P'_i$, we have two arrays each of length $d_i$: $IN_i$ 
and $OUT_i$. The array $IN_i$ contains the values of the registers $M_{j,i}$ copied to the local 
memory during the read phase, for all $j \in NBR_i$. The array $OUT_i$ contains the values 
computed during the internal computation at the end $\text{Read}_{d_i}$. This computation can be 
represented by a function $f_i$ that takes as input the array $IN_i$, and produces as output 
the array $OUT_i$. The sequence of operations described above is called a **Simulation Cycle** 
and is presented in Figure 4.3. The operations in this sequence are performed by 
processor $P'_i$, for some $1 \leq i \leq n$.

The **Simulation Cycle Algorithm** ensures that the processors will perform the 
simulation cycle in the Round-Robin order of their identifiers. The **transformed algo-
rihthm** is the fair composition of the Round-Robin algorithm and the Simulation Cycle 
algorithm.

The composed algorithm is implemented as follows: the processors perform a simu-
lation cycle every time they enter the critical section of Round-Robin algorithm. This 
is a fair composition, since the Round-Robin algorithm preserves deadlock freedom and
lockout freedom, as shown in Section 4.1.3. Thus, in the composed algorithm each processor performs the simulation cycle infinitely many times. The set of states $Q''$ of the transformed algorithm corresponds to states that are safe for the Round-Robin algorithm.

**Observation 4.2.1.** In any execution of the transformed algorithm starting from a state in $Q''$, the processors execute the simulation cycle in a Round-Robin way in increasing order of their identifiers.

Next, we define the simulation of states and operations. Then, simulation of executions will be used to prove correctness of the transformation.

### 4.2.2 Simulation of states

Consider some execution $e'$ and some state $q''$ in $a'$. Let $q'$ be the state immediately after the first write to $M_{j,i}$ that occurs before $q''$ of $a'$. (If no write to $M_{j,i}$ is performed before $q''$ in $a'$, then $q'$ is the initial system state in $a'$.) No write to $M_{j,i}$ occurs between $q'$ and $q''$. We say that $P_i'$ can read a new value from $M_{j,i}$ in $q''$, if no read operation of $M_{j,i}$ is performed by $P_i'$ between $q'$ and $q''$. Recall that, during a simulation cycle, a processor reads all the registers that it can read and writes to all the registers that it can write to.

Consider a state $s' \in Q''$, in which processor $P_i'$ has just entered the critical section. Then, whether $P_j'$ can read a new value from $M_{i,k}$ in $s'$ depends only on the values of $i$, $j$, and $k$. For example, in $s'$, processor $P_{r-1}'$ just finished executing a simulation cycle. Therefore, processor $P_{r-1}'$ cannot read new values, since it just finished reading all the registers it can read from. All the values it wrote are new for its neighbours. because no other processor has executed any operations of the simulation cycle yet. Processor
$P'_i$ can read a new value written by every neighbour and all the values written by $P'_i$ are not new for its neighbours. Processor $P'_{i+1}$ can read new values from all its neighbours, except for $P'_i$, if $P'_i \in NBR_{i+1}$. and only $P'_i$ can read a new value written by $P'_{i+1}$.

Now, we define the function $\chi$ that maps a state in $Q''$ in the shared memory model to states of the original algorithm in the message passing model. Let $\chi(s') = s$ for states $s' \in Q''$ and $s$ of the transformed algorithm and the original algorithm, respectively. If, in $s'$, some processor $P'_r$ can read a new value from register $M_{l,r}$, then in $s$, the queue $L_{l,r}$ contains exactly one message with the value of $M_{l,r}$ (or $L_{l,r}$ is empty if $M_{l,r} = \bot$). If $P'_r$ cannot read a new value from $M_{l,r}$ in $s'$, then the queue $L_{l,r}$ is empty in $s$. The values of the $\text{OUT}_i$ and $\text{IN}_i$ arrays of the processors $P'_i$ and $P_i$ are the same in both $s'$ and $s$.

If $P'_i$ has already started executing the simulation cycle in $s'$, then we need to modify the above slightly, as some of the reads or writes might have already been performed by $P'_i$. If $p_{c_i} = \text{Read}_k$, then $P'_i$ has already performed the operations $\text{read}_1, \ldots, \text{read}_{k-1}$ of the simulation cycle. The only modification is that $\text{IN}_i(1) \ldots \text{IN}_i(k-1)$ of processor $P'_i$ contain the values of the messages in the queues $L_{NBR_i(1),i}, \ldots, L_{NBR_i(k-1),i}$, respectively. If $p_{c_i} = \text{Write}_k$, then the only modification is that the local variables $\text{IN}_i(1) \ldots \text{IN}_i(d_i)$ of processor $P'_i$ contain the values of the messages in the queues $L_{NBR_i(1),i}, \ldots, L_{NBR_i(d_i),i}$, respectively, and the array $\text{OUT}_i(1) \ldots \text{OUT}_i(k)$ of $P'_i$ has the newly computed values.

### 4.2.3 Simulation of Operations

**Lemma 4.2.2.** The simulation cycle simulates an operation in the message passing model.
Proof. Consider two states $q_1$ and $q'_1$ such that $\chi(q'_1) = q_1 \text{ and } q'_1 \in Q''$. Suppose that in $q'_1$, processor $P'_i$ will next perform a simulation cycle. Consider a transition $(q_1, t, q_2)$ by processor $P_i$ in the message passing model, where $t$ is an operation as defined in Section 4.2.1. Also consider the sequence of transitions by $P'_i$ that contain the operations of a simulation cycle: $(q'_1, \text{READ}_1, q'_2), (q'_2, \text{READ}_2, q'_3), \ldots, (q_k, \text{WRITE}_d, q'_k+1)$. We will show that $\chi(q'_k+1) = q_2$ which implies that $\sigma(t)$ is a simulation cycle.

Since it is $P'_i$'s turn to execute the simulation cycle in state $q'_1$ and $\chi(q'_1) = q_1$, then, in $q_1$, queue $L_{j,i}$ contains at most one message with the same information as the register $M_{j,i}$ in $q'_1$, and queue $L_{i,j}$ is empty queues for all $j \in NBR_i$. From the definition of operation $t$, $P_i$ will receive at most one single messages from each queue $L_{j,i}$, perform some local computation, and send at most one message to each queue $L_{i,j}$. As a result, in $q_2$ the queues $L_{j,i}$ will be empty and the queues $L_{i,j}$ will each have at most one message.

Now, we will examine what happens during the simulation cycle. During the read phase of the simulation cycle, $P'_i$ will read from $M_{j,i}$ the same value that processor $P_i$ will receive from $L_{j,i}$ during $t$, for all $j \in NBR_i$. Then, the value in the shared registers, $M_{j,i}$, will not be new for $P'_i$ until state $s'_{k+1}$. The local arrays $IN_i$ and $OUT_i$ of processors $P_i$ and $P'_i$ have the same values because, at the end of the read phase, the internal computation performed by $P'_i$ will be the same as the computation performed by $P_i$ during the internal steps of $t$. The arrays $IN_i$ and $OUT_i$ will not change during the write phase. During this phase $P'_i$ will write $\bot$ to $M_{i,j}$ if no message will be send to $L_{i,j}$. Otherwise, $P'_i$ will write the same value to register $M_{i,j}$ that $P_i$ will send in a message to the queue $L_{i,j}$. The state reached after the write phase is finished is $q'_{k+1}$. Therefore, from the discussion above and the definition of the function $\chi$, we get that $\chi(q'_{k+1}) = q_2$. \qed
4.2.4 Correctness of Transformation

Since the Round-Robin algorithm is self-stabilizing (as we proved in Section 4.1.3), the transformed algorithm (the composition of the Round-Robin algorithm and the simulation cycle algorithm) stabilizes to the Round-Robin task. The first condition for the correctness of the transformation holds, because we chose the set $Q''$ to be the set of states of the transformed algorithm that are safe for the Round-Robin task. We will show that the second condition of the transformation holds as well.

Theorem 4.2.3. For each execution $a'$ of the transformed algorithm in the shared memory model that starts from a state in $Q''$, there is an execution $a$ of the original algorithm of the message passing model such that $a'$ simulates $a$.

Proof. Let $a'$ be an arbitrary execution of the transformed algorithm in the shared memory model that starts from a state in $Q''$. By Observation 4.2.1, starting from a state in $Q''$, the processors execute simulation cycles in a Round-Robin way in increasing order of identifiers. As a result, $a'$ can be viewed as a sequence of execution fragments, which each contains one simulation cycle and some auxiliary operations. We denote as $sc'_i$ the execution fragment that starts with the first operation of the $i^{\text{th}}$ simulation cycle and ends with the last operation of that simulation cycle. Assume that $s'_i$ is the state immediately before $sc'_i$, and $q_i'$ is the state immediately after $sc'_i$. We denote by $aux'_i$ the execution fragment that starts from the operation immediately after $s'_{i-1}$ and ends with the operation immediately before $q_i'$. Using the above notation, execution $a'$ can be written as $(q'_0, aux'_1, s'_1, sc'_1, q'_1, aux'_2, s'_2, sc'_2, q'_2, \ldots)$, where $q'_0 \in Q''$. 

The execution fragments $aux'_i$ contain only auxiliary operations which do not simulate any operation in model $\mathcal{M}$. In particular, they only and change the variables used in the tree construction of the Round-Robin algorithm. From the definition of $\chi$, if $(v'_1, t', v'_2)$ is a transition of $a'$ where $t'$ is an auxiliary operation, then $v'_1$ and $v'_2$ simulate the same state in $a$. It follows that

$$\chi(q'_k) = \chi(s'_{k+1}), k \geq 0. \quad (1)$$

Now, we will construct execution $a$ following the definition of simulation of executions. Execution $a$ starts with the state $u_1 = \chi(s'_1)$. By (1), we get that $u_1 = \chi(q'_0)$. If, in $s'_1$, it is $P_{g_1}$'s turn to execute the simulation cycle, then we schedule $P_{g_1}$ to execute an operation of the message passing model starting from $u_1$. Suppose that the resulting state is $u_2$. During the execution fragment $s_{c_1}$ where $P_{g_1}'$ performs the simulation cycle, all the other processors may perform only auxiliary operations. Since the simulation cycle simulates an operation in the message passing model and the auxiliary operations do not change the simulating parts of the system, we get that $u_2 = \chi(q'_1)$, from the definition of the simulation of operation. Then, by (1), we get that $u_2 = \chi(s'_2)$.

Now, we construct (in the way described above) a valid finite execution $\hat{a}$ which ends with state $u_\hat{a} = \chi(s'_\hat{a})$, such that $a$ is simulated by a suffix of execution $a'$. We will show that we can add one more operation to expend $a$ so that the new execution is simulated by another suffix of $a'$. Similarly, as described above, if in $s'_i$ it is $P_{g_i}'$'s turn to execute the simulation cycle, then we schedule $P_{g_i}$, execute an operation of the message passing model starting from $u_i$. We define the state $u_{i+1}$ to be the resulting state after $P_{g_i}$ performed
this operation and similarly, as above, \( u_{i+1} = \chi(q_i') = \chi(s_{i+1}') \).

We conclude that for every execution \( a' \) of the transformed algorithm starting from a state in \( Q'' \), we can created an execution \( a \) of the original algorithm such that \( a' \) simulates \( a \).

We observe that the same transformation presented in Section 4.2 can be used to transform self-stabilizing algorithms from the message-driven model to the shared memory model. This is because in every state \( q \) of the message passing model, such that \( \chi(q') = q \) and \( q' \in Q'' \), there is always at least one message in some link. This is true since in \( q' \) exactly one processor is allowed to execute the simulation cycle, because the Round-Robin task holds. Thus, this processor can read at least one value different than \( \perp \). We conclude that any execution that is simulated by an execution of the transformed algorithm that starts from a state in \( Q'' \) will not contain time-outs. As a result, we do not need to simulate time-outs, because we ensure that they will never happen starting from a state in \( Q'' \).

### 4.2.5 Complexity of the Transformed Algorithm

We will analyze the stabilization time of the transformed algorithm. This time depends on the stabilization time of the original algorithm. Suppose that the original algorithm stabilizes in \( f(n) \) rounds, where \( n \) is the number of processors. Also, assume that the transformed algorithm will stabilize to the Round-Robin task in \( g(n) \) rounds.

Observe that after the transformed algorithm stabilizes to the Round-Robin task, then the processors simulate operations of the original algorithm, one at a time, in a
round-robin order. We consider the execution of the transformed algorithm after the Round-Robin task holds. Note that concurrently with any operation of the simulation cycle performed by some processor, all the rest may perform some auxiliary operation. As a result, any operation of the simulation cycle in the worst case counts as a round. Thus, \( s(n) \) is the maximum number of rounds performed during the execution fragment during which some processor executes the simulation cycle. This is bounded by the maximum number of operations in the simulation cycle of some processor, so \( s(n) = \max_{1 \leq i \leq n}(2d_i) \).

One round of the original algorithm corresponds to an execution fragment where all processors execute the simulation cycle at least once. By the nature of the Round-Robin algorithm, we can divide the executions in such a way so that in every simulated round, every processor executes the simulation cycle exactly once. Thus, a round of the original algorithm will be simulates by at most \( ns(n) \) rounds of the transformed algorithm.

Now, we will add the number of rounds that are performed between the simulation cycles, when all the processors perform auxiliary operations (operations of the Round-Robin algorithm not in the critical section). We denote the maximum number of rounds performed in between two consecutive executions of the simulation cycle by \( k(n) \). Note that the Round-Robin task already holds since we already counted \( g(n) \) rounds. By the Round-Robin algorithm, \( k(n) = O(\log n) \). This is because after some processor \( P_i' \) exits the critical section and before the next processor enters the critical section, as we explain in the proof of Lemma 4.1.10, the operations that need to be performed are by processors that label some node in the \( \text{PATH}_i \). Since this path has length at most \( \lceil \log_2 n \rceil \), then \( k(n) = O(\log n) \).

Therefore, from above, the time complexity of the transformed algorithm is equal
to the stabilization time of the transformed algorithm to the Round-Robin task plus
the stabilization time of the original algorithm multiplied with the number of rounds
needed for the transformed algorithm to simulate the operations of a round of the original
algorithm. This is equal to \( g(n) + f(n) \times n(c(n) + k(n)) \). Recall that in Section 4.1.4 we proved that \( g(n) = (s(n) + 2)(4 \log n + 1) \), \( n \log_2 n \) rounds.

4.3 Transforming Self-Stabilizing Algorithms from Synchronous Message Passing to Asynchronous Shared Memory

We use the transformation presented in the previous section to simulate a synchronous
round by a round-robin cycle. In a synchronous round, the processors simultaneously read
the values from the single-reader, single-writer registers, then simultaneously change their
local states, and finally simultaneously write some new values to the single-writer single-
reader registers. By achieving round-robin execution, we do not immediately simulate
a synchronous system because, in every simulation cycle, exactly one processor changes
the values in its IN array and the values of the single-reader, single-writer registers it
can write to. Thus, there will be no round in which all processors read the same values
when they perform a round-robin cycle.

In order to simulate a synchronous step by an asynchronous Round-Robin cycle, we
keep two consecutively computed values in the single-writer single-reader registers: we
call these values the **previous value** and the **current value**. Now, let processor \( P_i \) perform a simulation cycle. For \( j \in NBR_i \), if \( j < i \), processor \( P_i' \) will read the previous value in \( M_{j,i} \) and store it in \( I_N(j) \) and, if \( j \geq i \), \( P_i' \) will read the current value in \( M_{j,i} \) and store it in \( I_N(j) \). We note that a synchronous round will start with \( P_1' \) executing a simulation cycle and end with \( P_n' \) ending its execution of the simulation cycle. The modification described above will ensure that all the processors during the simulated synchronous round will use the same array \( I_N \) in order to perform the local computation described by the function \( f \) during the simulation cycle.

### 4.4 Transforming Self-Stabilizing Algorithms from the Shared Memory Model to the Message Passing Model

We describe how we can get a transformation from the shared memory model using single-reader, single-writer registers to the message passing model defined in this thesis. We will modify the transformation of self-stabilizing algorithms by Dolev, Israeli, and Moran [15], from the shared memory model (that uses single-reader, single-writer registers) to the message-driven model. This transformation is presented in Section 3.3.

Recall that this transformation uses a self-stabilizing token-passing algorithm for two processors in the message-driven model. We only need to modify this algorithm to get the desired transformation. The token-passing algorithm used is based on Dijkstra's token-passing algorithm in a ring: One processor always sends what it receives. It updates its
local value if it receives something new and in that case it gets the token. The processor at the other end of the link gets the token when it receives the same value as the one in its local memory. In this case, it creates a new value and sends it. Otherwise, it sends its value unchanged. The only modification needed is that, if a processor performs a receive operation that returns \(\bot\), it will next send a message with its local value. Note that this is exactly what it does if it does not receive the token. Applying this modified token-passing algorithm to the transformation described in [15], we get a transformation from the shared memory with single-reader single-writer registers to the message passing model defined in this thesis.
**WRITE**<sub>h</sub>  
**Precondition:**  
\[ pc \leftarrow \text{write}_h \]  
**Effect:**  
\[ j \leftarrow N_i(h) \]  
\[ C_{i,j} \leftarrow 1 \]  
\[ pc \leftarrow \text{read}_h \]

**READ**<sub>h</sub>  
**Precondition:**  
\[ pc \leftarrow \text{read}_h \]  
**Effect:**  
\[ j \leftarrow N_i(h) \]  
\[ c \leftarrow C_{i,j} \]  
if \( c = 1 \) then  
\[ pc \leftarrow \text{read}_h \]  
else if \( |N_i| > t \) then  
\[ h \leftarrow h + 1 \]  
\[ pc \leftarrow \text{write}_h \]  
else if \( i = 1 \) then  
\[ pc \leftarrow \text{crit} \]  
else  
\[ pc \leftarrow \text{write}_h \]

**WRITEMIN**  
**Precondition:**  
\[ pc \leftarrow \text{writemin} \]  
**Effect:**  
\[ /* \text{If } < P'_k, P'_l > \text{ labels the single node where } P'_i \text{ has the maximum identifier }*/ \]  
\[ C_{k,i} \leftarrow 0 \]  
\[ pc \leftarrow \text{readmin} \]

**READMIN**  
**Precondition:**  
\[ pc \leftarrow \text{readmin} \]  
**Effect:**  
\[ /* < P'_k, P'_l > \text{ labels the single node where } P'_i \text{ has the maximum identifier }*/ \]  
\[ c \leftarrow C_{k,i} \]  
if \( c = 0 \) then  
\[ pc \leftarrow \text{readmin} \]  
else \( pc \leftarrow \text{crit} \)

**CRITICALSECTION**  
**Precondition:**  
\[ pc \leftarrow \text{crit} \]  
**Effect:**  
Perform the critical section.  
if \( |N_i| > 0 \) then  
\[ pc \leftarrow \text{write}_i \]  
else \( pc \leftarrow \text{writemin} \)

Figure 4.1: Round-Robin algorithm for processor \( P_i \), using Read/Write registers
Figure 4.2: State Diagram of Round-Robin algorithm
Read\(_k\), \(1 \leq k \leq d_i\)

Precondition:

\(pc_i = \text{read}_k\)

Effect:

\(j \leftarrow NBR_i(k)\)
\(IN_i(j) \leftarrow M_{i,j}\)
if \(d_i > k\)
then \(pc_i \leftarrow \text{read}_{k+1}\)
else \(\text{OUT}_i \leftarrow f_i(IN_i)\)
\(pc_i \leftarrow \text{write}_1\)

Write\(_k\), \(1 \leq k \leq d_i\)

Precondition:

\(pc_i = \text{write}_k\)

Effect:

\(j \leftarrow \text{NBR}_i(k)\)
\(M_{i,j} \leftarrow \text{OUT}_i(j)\)
if \(d_i > k\)
then \(pc_i \leftarrow \text{write}_{k+1}\)
else \(pc_i \leftarrow \text{read}_1\)

Figure 4.3: Simulation Cycle
Chapter 5

Discussion and Future Work

In this thesis, we formally defined the transformation of self-stabilizing algorithms from one model to another. We gave a transformation of self-stabilizing algorithms from the asynchronous message passing model to the asynchronous shared memory model. This transformation uses a new Round-Robin algorithm that stabilizes to the Round-Robin task in $O(n \log^2 n)$ rounds. We modified this transformation so that it transforms self-stabilizing algorithms from the synchronous message passing model to the asynchronous shared memory model. In this chapter, we discuss the parts of this thesis we believe can be extended.

First, we propose some extensions of our work. We modify the Round-Robin algorithm so that it works for a model that uses only single-reader, single-writer registers. Also, we propose a new transformation from the message passing model to the shared memory model that achieves better concurrency than the one presented in Chapter 4. In the final section, we propose some directions for future work.
5.1 Extensions of our Work

We discuss a variant of the Round-Robin algorithm that works for a different shared memory model. For the Round-Robin algorithm in Chapter 4, we use single-reader, single-writer and multi-writer, multi-reader registers. The multi-reader, multi-writer registers are used only to achieve the token-passing task in a system with two processors. Recall that we used them for the internal nodes of the tree $T$. We used multi-reader, multi-writer registers for the token-passing task for two processors because we get a simple algorithm that stabilizes immediately. (A processor owns the node if and only if it has the token in the ring of two processors that label the same internal node of the tree used by the algorithm.) If we do not want to use multi-reader, multi-writer registers, we can replace them by single-reader, single-writer registers, and compose our algorithm with self-stabilizing algorithms for the token-passing task for two processors that use only single-reader, single-writer registers, one for each internal node. Such algorithms exist in the literature, for example, Dijkstra's algorithm for a ring of two processors, as described in [14]. Note that the time complexity of this algorithm for only two processors is constant. We conclude that we can modify the Round-Robin algorithm so that it uses only single-writer, single-reader registers. If we use this modified Round-Robin algorithm then we get a transformation from the message passing model to the shared memory model that utilizes only single-writer, single-reader registers.

The transformation from the asynchronous message passing model to the asynchronous shared memory model presented in this thesis does not have maximum concurrency. This is because it allows only one processor at a time to perform operations that simulate the
operations of the original algorithm. However, it is possible that some processors could proceed with their computation independently. We observe that it is enough to ensure that the processors of every clique of the graph (of the network of the message passing system) execute the simulation cycle in a Round-Robin order. The resulting algorithm has better concurrency, since two processors that do not belong to the same clique can execute the simulation cycle concurrently.

Based on this observation, we can create a new transformation that uses the Round-Robin algorithm of Chapter 4, as follows. We use one tree \( T \) for each maximal clique of the graph. The leaves of such a tree are the processors in the clique. This is illustrated in Figure 5.1. A processor **belongs in a tree** if that tree has this processor as a leaf. We assume a total order on the trees, which is known to the processors. Every processor executes the Round-Robin algorithm in some tree to which it belongs, until it owns the tree. Then, it continues with the next tree to which it belongs. Finally, it will own all the trees to which it belongs. Then it will execute a simulation cycle. After that, it will release ownership of these trees. Then, it will execute the Round-Robin algorithm starting again, from the first tree to which it belongs.

The processors interleave operations of a control algorithm that does the following: Each processor checks if it owns all the trees to which it belongs that are before the tree to which it is currently running the Round-Robin algorithm. If not, it releases the ownership of these trees and then executes the Round-Robin algorithm starting from the first tree to which it belongs. Thus, except for a brief time at the beginning of the execution, every processor will own a prefix of the trees to which it belongs.

We need this total order on the trees to prevent a deadlock, caused as we next describe.
Consider two processors $P'_i$ and $P'_j$ that belong in both trees $T_1$ and $T_2$. Assume that the order in which the processors try to own the trees did not matter. Then $P'_i$ could own tree $T_1$, and wait forever to own tree $T_2$, and $P'_j$ could own tree $T_2$, and wait forever to own tree $T_1$. We leave the proof of correctness of the algorithm as future work.

Note that if the network corresponds to a complete graph, the new algorithm has only one tree, and, hence, is the same as the algorithm in Chapter 4. The time complexity of the new algorithm depends on the topology of the corresponding graph of the system. The complexity of this algorithm is left as future work as well.

![Graph and Trees Diagram](Image)

Figure 5.1: An example of the new transformation from message passing to shared memory

### 5.2 Future Work

We conclude with some subjects for future work. A question that arises is whether the algorithm with improved concurrency is optimal. There might be transformations from the asynchronous message passing model to the asynchronous shared memory model that have improved complexity or concurrency. It is possible that there is a trade-off between
these two factors.

Dolev, Israeli, and Moran [15] give a transformation from the shared memory model to the message-driven model. The resulting algorithm needs unbounded space because it uses a token-passing algorithm whose task is in the weak exclusion class. and they prove that any task in the weak exclusion class needs unbounded space. However, we do not know if every transformation of self-stabilizing algorithms from the shared memory model to the message passing model will produce algorithms that need unbounded space.

In this thesis, we gave a transformation from the message-driven model to our message passing model. We propose as future work, to find relations between: the message-driven models with bounded or unbounded links and our model with bounded or unbounded links. Before talking about transformations between these models, we need to study them. For our model with bounded links, we have to specify what happens when a processor sends a message to a full queue. For the message-driven models, we need a formal description of the time-outs.

Transformations of self-stabilizing algorithms between different shared memory models are a subject for future work as well. If such transformations do not exist, then we may be able to create an hierarchy of shared memory models for self-stabilization, similarly to the consensus hierarchy.
Bibliography


aperiodic sequence. 58
asynchronous system. 8
    execution. 8
    interleaving scheduler. 8
    round-robin scheduler. 8
    round-robin cycle. 8
    transition. 8
    round. 16
atomicity property. 25
backward simulation. 37
blocked processor. 65
communication deadlock. 41
composite atomicity model. 12
correctness of transformation. 20
current value. 86
distributed system. 5
environment of a processor. 7
fair composition of two algorithms. 17
fair composition technique. 18
fair execution. 13
FirstInPath. 68
forward simulation. 37
history relation. 38
intermediate state. 24
legal execution. 13
message complexity. 17
message passing model. 12
    receive step. 12
    send step. 12
message-driven model. 13
neighbouring processors. 7
new value. 78
operation. 5
    applicable. 6
    atomic. 7
    auxiliary. 26
    external. 5
    input. 6
    output. 6
    interface. 6
    input. 6
    output. 6
    internal. 5
    sequence of operations applicable. 6
ownership. 64
    of a node. 64
    of a tree. 64
PATH. 68
periodic sequence. 58
    eventually periodic. 58
previous value. 86
range. 9
read-modify-write object. 11
refinement map. 36
register. 11
    multi-reader. 11
    multi-writer. 11
    single-reader. 11
    single-writer. 11
Round-Robin algorithm. 64
    AllCompleteCycle property. 67
cycle. 67
safe state. 13
scheduler. 7
self-stabilizing algorithm. 2, 14
    given some task. 18
shared memory model, 11
  read step, 11
  write step, 11
simulation cycle, 77
  read phase, 77
  write phase, 77
Simulation Cycle Algorithm, 77
simulation of execution, 24
simulation of operations, 22
simulation of states, 20
  of processor, 20
  of communication medium, 21
  of environment, 22
  of system, 22
snapshot object, 12
  scan, 12
space complexity, 16
stabilization time, 16
state, 5
  of a distributed system, 5
  of a processor, 5
  of the communication medium, 5
step, 5
  external, 5
  input, 5
  output, 5
  internal, 5
superrange, 33
synchronous system, 7
  execution, 7
  round, 7
  transition, 8

task, 9
  weak exclusion, 10
time-out, 41
totally ordered reliable broadcast model, 47
trace, 9
transient fault, 2, 14
work, 16