CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

A Compositional Framework for Designing Self-Stabilizing Distributed Algorithms

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

von

Abhishek Dhama, M.Sc.

Gutachter:

Prof. Dr. Oliver Theel
Prof. Dr. Ernst-Rüdiger Olderog

Tag der Disputation: 3. Mai 2013

Acknowledgements

> *Perseverance in (seeking to gain) the knowledge of the Supreme Spirit, and perception of the gain that comes from knowledge of the truth: This is called knowledge : all that is contrary to this is ignorance.*
>
> Šrimadbhagavadgītā 13:11

As I put finishing touches to my dissertation, I realize that this endeavour would not have been successful without support and encouragement I received all along. I owe my deepest gratitude to Prof. Oliver Theel for taking me under his tutelage. His insights, comments and patience helped me immensely to see through the period when the research was not particularly productive. Without guidance and constant support of Prof. Theel this dissertation would not have been possible. I would like to thank Prof. Ernst-Rüdiger Olderog for being kind enough to be second refree of the dissertation. His feedback helped to improve the readability and the clarity of presentation. I would also like to express my gratitude to Prof. Michael Sonnenschein and Dr. Marco Grawunder for agreeing to serve on dissertation committee. Financial support provided by Deutsche Forschungsgemeinschaft via Graduiertenkolleg Trustsoft and SFB AVACS is also gratefully acknowledged.

A huge thanks to colleageus at Chair of System Software and Distributed Systems, Graduiertenkolleg Trustsoft and AVACS sub-project S3 –especially Jens Oehlerking, Nils Müllner, Timo Warns, and Kinga Lipskoch, with whom I had the pleasure of co-authoring papers and sharing office– for good-spirited discussions and friendly atmosphere which made my stay at University of Oldenburg worthwhile. I would like to thank my friends Satya Srinivas, Kunal Sachdeva and Mani Swaminathan for maintaining the sanity levels.

I would also like to take this opportunity to thank my parents and my sister for their unwavering support and faith in my abilities. My thanks go to my wife for practically infinite patience and love.

Abstract

The proliferation of numerous computing devices in the various facets of life has remarkably elevated the premium placed on fault tolerance of the algorithms running on such devices. Ideally a fault-tolerant distributed algorithm should recover from a fault without any degradation in the service. A fault-tolerant distributed algorithm whose recovery is completely transparent to the user is said to be masking fault-tolerant. The design of masking fault-tolerant distributed algorithms –notwithstanding the desirability of masking fault tolerance– is hardly trivial, because algorithm designers must take all possible faults into account. Should the requirement provide an allowance for the phases where a distributed algorithm may deviate from its specification for a finite time period, non-masking fault tolerance becomes a viable design alternative. Although a user might experience outage of service for short finite periods, a non-masking fault-tolerant distributed algorithm will fulfill its specifications once recovery is complete.

Self-stabilization is a novel method to provide non-masking fault tolerance. A distributed system is said to be self-stabilizing if and only if 1) it –when perturbed– reaches a closed set of legal states in finite time, and 2) does not leave this set voluntarily. The first condition is referred to as convergence, and the second condition is called closure. However, designing and subsequently proving convergence of a self-stabilizing system is not easy. Convergence is proven by showing existence of a so-called ranking function that decreases for any execution step of the system and attains its minimum value in the set of legal states. Although various automated methods have been proposed to determine ranking function for proving convergence, such methods do not scale well. Unfortunately the compositional methods found in the literature are also rather restrictive with the respect to the conditions imposed on component algorithms.

We investigate whether the conditions under which component algorithms are self-stabilizing can be transcended while composing them. To that end, this dissertation presents a suite of compositional methods which can be used to compose self-stabilizing algorithms, though component algorithms themselves might be self-stabilizing under mutually incompatible conditions. The compositional framework exploits the knowledge of the ranking function used to prove the convergence of a component algorithm under its original scheduler. We show that embedding of ranking functions in the guards of the composed algorithm preserve the self-stabilization properties of component algorithms under much stronger schedulers. Since evaluation of ranking function requires global system state, a transformer is provided to perform on-the-fly evaluation of ranking function in any state. The transformer *per se* can also be used to transfer self-stabilization property of a self-stabilizing algorithm under a much stronger scheduler. Furthermore, the compositional framework contains operators to compose self-stabilizing algorithms with bidirectional variable dependencies.

Zusammenfassung

Die zunehmende Verwendung verteilter Systeme im alltäglichen Leben für sicherheitskritische Anwendungen stellt besondere Anforderungen an deren Zuverlässigkeit. Ein hoher Grad an Zuverlässgkeit solcher kritischer verteilter Systeme kann zum Beispiel gewährleistet werden, indem man sie *fehlertolerant* entwirft. *Stabilisierungstechniken* werden in diesem Zusammenhang dazu genutzt, *nicht-maskierende* Fehlertoleranz einzuführen. Ein System ist *selbststabilisierend* genau dann, wenn, unabhängig vom initialen Zustand gewährleistet ist, dass das System mit endlich vielen Schritten den *legalen* Zustandsraum erreicht und diesen ohne Fremdeinwirkung nicht mehr verlässt. Der legale Zustandsraum ist durch ein Prädikat definiert. Ein selbststabilisierendes System ist fähig sich von vorübergehenden Fehlern zu erholen, unabhängig von der Anzahl der Fehler.

Obwohl Selbststabilisierung eine wünschenswerte Eigenschaft für verteilte Algorithmen (welche von verteilten Systemen ausgeführt werden) ist, ist die Beweisbarkeit dieser Eigenschaft nicht trivial. Die Relevanz der Selbststabilisierung bedingt sowohl in der Entwurfs- wie auch in der Verifikationsphase selbststabilisierender Algorithmen ein *ganzheitliches* und systematisches Vorgehen.

Diese Dissertation befasst sich mit der Entwicklung eines kompositorischen Rahmens für den Entwurf selbststabilisierender verteilter Algorithmen. Die kompositorische Rahmen ausnutzt das Wissen der Ranking-Funktion, die verwendet werden, um die Konvergenz einer Komponente Algorithmus unter seiner ursprünglichen Scheduler zu beweisen. Da Auswertung von Ranking-Funktion erfordert globale Systemzustand, ein Transformator ist so konzipiert, on- the-fly Auswertung von Ranking-Funktion in jedem Systemzustand durchzuführen. Der Transformator kann auch verwendet werden, um die Selbst-Stabilisierung Eigenschaft einer selbst-stabilisierenden Algorithmus unter einer viel stärkeren Scheduler zu übertragen.

Contents

List of Figures

List of Symbols

$\triangle$	Lifting Composition
$\triangle_G$	Generic Lifting Composition
$\triangle_S$	Symmetric Lifting Composition
$\triangle_\kappa$	Lifting Composition with κ-Local Mutual Exclusion
$\triangle_B$	Lifting Composition with Bidirectional Dependency
$\mathbb{A}$	Distributed Algorithm
$\mathcal{A}_i$	Sub-Algorithm run by Process i
$\mathcal{G}_{\mathcal{A}_{i\,j}}$	j^{th} Guarded Command of the Sub-Algorithm run by Process i
s_i	Local State of Process i
σ	Global State of a Distributed Algorithm
Ξ	Execution of a Distributed Algorithm
$\mathbb{D}$	Scheduler
ϱ	Scheduling Strategy
$\mathcal{P}$	Predicate
P_i	Process i
$\varDelta$	Ranking Function
$\varDelta_{Ai_x}$	Function obtained by replacing the variables modified by $\mathcal{G}_{Ai_x}$ by their respective assignment expressions in $\varDelta_A$
δ_{Aix}	Difference between the values of $\varDelta_{Ai_x}$ and $\varDelta_A$
$\breve{\Xi}_{\mathbb{A}\vert\mathbb{B}}$	Projection of an Execution of Algorithm $\mathbb{A}$ on Algorithm $\mathbb{B}$
$\hat{\Xi}$	Maximal Execution of a Distributed Algorithm
$\sigma_{\vert\mathbb{A}}$	Projection of a Global State on Algorithm $\mathbb{A}$

1

Introduction

1.1 Motivation

There has been a remarkable increase in the usage of distributed systems in the past decade due to the ever decreasing price of hardware components. The spectrum of such systems is wide and ranges from home appliances to battlefield control systems. However, the pervasiveness of distributed systems also necessitates a high degree of trustworthiness and resilience. For instance, consider a intruder detection system deployed on a border consisting of multiple sensor nodes spread over the area of interest [1]. The nodes collaborate to track any foreign object and report its location and trajectory to the control system. However, due to the inherent nature of the terrain, some of the sensor nodes may not be accessible after the deployment or they may get damaged during the course of the operation. In order to fulfill its mission, it is important that the system has allowance for such scenarios. Thus, endowing distributed systems with fault tolerance is imperative for the system designers.

Fault tolerance can be divided into two categories: *masking* and *non-masking* [2]. Masking fault tolerance implies that, in the event of a system being effected by faults, the system recovery is transparent to an external observer. A masking fault-tolerant system fulfills its specification even when faults are occurring. A fault tolerant system is said to be non-masking fault-tolerant if during the recovery period the system deviates from its specified behavior.

Masking fault tolerance is an extremely appealing property however, designing a masking fault-tolerant system is equally challenging. The designer needs to know all the possible failure scenarios *a priori* which is not trivial. A failure scenario not foreseen during the design phase can render the system useless or may prove hazardous for mission-critical applications. However, if the deployment scenario permits periods of non-conformance with the system specification then, a stabilization technique can be used to design non-masking fault-tolerant systems.

Self-stabilization [3] is an elegant technique for imbibing distributed systems with non-masking fault tolerance in the presence of transient faults. Self-stabilizing systems are guaranteed to converge to their specified behavior within finite time irrespective of their initial state.

Although the definition of self-stabilization appears trivial, its implication in the context of designing a fault-tolerant distributed system is immense. As long as the faults inflicting the system are transient in nature, the designer does not need to account for each one of them while designing a self-stabilizing system. Indeed, the paradigm shift caused by the wide-spread usage of distributed systems has given rise to the scenarios where distributed systems span multiple administrative domains. There is, therefore, a need to equip such systems with “autonomy” so that the system can adapt and calibrate its behavior based on the changes in the system or the deployment environment [4]. Additionally, the advent of wireless sensor networks [5] and the associated dynamism has made it imperative to design distributed systems whose correct behavior does not depend on a pre-defined initial state. Indeed, self-stabilizing algorithms have been used as the kernels of the self-organizing [6] and self-healing wireless sensor networks [7]. Similarly, the formalization frameworks [8, 9, 10] of the various self-* properties of autonomic computing systems essentially mirror that of self-stabilization.

The increased relevance of self-stabilization in the design of large scale distributed systems has led to a heightened emphasis on devising the design techniques for self-stabilizing systems as well. The challenge is compounded by the intricacy of the correctness proofs of self-stabilizing systems since, it needs to be shown that, the system converges to the correct behavior for every possible initial state. Model checking [11] based automated verification techniques appear to be ideal for designing self-stabilizing system because they require minimal assistance from the system designer. However, these techniques do not scale well because the number of possible executions – in addition to the number of system states– grows exponentially if the system size or complexity is increased. Compositional design of self-stabilizing algorithms has been, consequently, proposed as an alternative to overcome the scalability issues.

Typically, the compositional methods for self-stabilizing algorithms guarantee that a composed algorithm is self-stabilizing if the component algorithms are self-stabilizing as well. Moreover, component algorithms must be "compatible" with each other for the composition methods to work. The notion of compatibility encapsulates the similarity of the assumptions under which the component algorithms are shown to be self-stabilizing and the dependency between the component algorithms. Dependency between component algorithms is induced by the communication between components. The other aspect of the compatibility, –namely, the assumptions made while proving that a distributed algorithm is self-stabilizing– abstract away the implementation details of the distributed algorithm. The assumptions primarily correspond to the following aspects of the implementation scenario: 1) relative speeds of the processors implementing the distributed algorithm, 2) synchrony between processor clocks or the lack thereof and, 3) the mode of communication between the processors. The proofs of self-stabilization are extremely sensitive to these assumptions [12]; that is, a distributed algorithm shown to be self- stabilizing under a set of assumptions may not remain self-stabilizing if the implementation scenario is altered. The sensitivity of self-stabilization to the proof assumptions also renders the composition methods for self-stabilizing algorithms fragile. Therefore, the compositional methods for self-stabilizing algorithms implicitly require that component algorithms must be self-stabilizing under similar assumptions. Dependency between component algorithms also leads to additional constraints that must be fulfilled by the component algorithms to ensure that the composed algorithm is self-stabilizing. Consequently, the scope of the compositional methods for self- stabilizing algorithms reported so far in the literature is limited to algorithms which either have no dependency, that is, components do not have common variables or have only unidirectional dependency –wherein the variables modified by a component algorithm are read by its counterpart, thereby inducing a dependency between the component algorithms.

The literature is, however, replete with numerous self-stabilizing algorithms under varying assumptions owing to past two decades of research on stabilization. The constraints imposed by the hitherto proposed compositional methods restrict the set of self-stabilizing algorithms that can be designed using these methods. More specifically, a system designer may be confronted with a situation where, although component algorithms solve the subproblems in self-stabilizing manner, the algorithms cannot be composed owing to the mutually incompatible proof assumptions of the component algorithms. The lack of suitable composition methods in such scenarios may lead to designing a self- stabilizing solution to a problem for which a solution –albeit under different assumptions– already exists. Thus, notwithstanding the increased usage of self-stabilizing algorithms, present design techniques lack the capability to construct provably correct large scale self-stabilizing systems.

A compositional framework that provides techniques to compose self-stabilizing algorithms regardless of respective proof assumptions is required in the light of the relevance of self-stabilization in fault-tolerant system design and the deficiencies of the prevalent design techniques. The desired framework should be able to transcend the constraints imposed by the incompatible assumptions made while proving correctness of the component algorithms. In addition, the desired framework should provide compositional methods for the components which have bidirectional dependency wherein each component algorithm read variables modified by its counterpart. It is also desirable that the compositional framework should require minimum possible input from the designer so that non-experts can use the framework to design self-stabilizing algorithms. The compositional methods of the framework should also have potential to be automated so that tool support can be provided for designing self-stabilizing algorithms.

1.2 Contributions

The primary result of this dissertation is a suite of compositional methods for self-stabilizing algorithms. A system designer can select a compositional method based on the respective proof assumptions of the component algorithms. The specific contributions of this dissertation are listed below.

Scheduler-Oblivious Transformation. We present a method to transform a distributed algorithm, shown to be self-stabilizing under a restrictive scheduler, to a distributed algorithm that is self-stabilizing any weakly fair scheduler [13]. Informally, a scheduler is an abstract entity that resolves any inherent non-determinism in a distributed algorithm. The transformation exploits the proof artifacts which are generated while proving self-stabilization of a distributed algorithm under a restrictive scheduler. We also provide a method to increase the concurrency of the transformed algorithm without affecting the self-stabilization property of the transformed algorithm.

Lifting Composition. We define a compositional method –referred to as *lifting composition*– that preserves the self-stabilization property of a component algorithm under the scheduler of its counterpart. Lifting composition also utilizes the proof artifacts to preserve the self-stabilization property of a distributed algorithm under unfavorable schedulers. Additionally, we define the variants of lifting composition which preserves the self-stabilization property of both components under a much stronger scheduler. We also show how to exploit the structure of proof artifacts to increase the concurrency in the composed algorithm.

Composition with Variable Dependencies. We address the challenge of incorporating variable dependencies –induced by components reading the variables modified their counterparts– by defining the composition methods that preserves self-stabilization of the components algorithm even if –in addition to incompatible schedulers– components read each other's variables. To that end we define the variants of lifting composition for algorithms with unidirectional and bidirectional variable dependencies.

The above-mentioned contributions result in a rich set of compositional methods which can be used to design large correct-by-construction self-stabilizing algorithms. These methods transcend the respective proof assumptions of the component algorithms, thereby allow a system designer to select a component algorithm solely based on the subproblem to be solved.

1.3 Thesis Outline

The dissertation consists of seven chapters and is organized as follows.

Chapter 2. We introduce the system model and provide the associated definitions. In addition to self-stabilization, we recall the concepts related to non-determinism and the weaker forms of self-stabilization.

Chapter 3. We provide a brief survey of the verification techniques used for self-stabilizing algorithms. The techniques discussed include both, the formal verification techniques and the control theoretic techniques. Chapter 3 also provides an overview of the compositional methods proposed for self-stabilizing algorithms.

Chapter 4. The approach of using proof artifacts to facilitate the composition of self- stabilizing algorithms with incompatible schedulers is introduced. The interplay between the respective schedulers of component algorithms and the self-stabilization of the composed algorithm is also studied in detail. We further show the usage of lifting composition with the help of illustrative examples.

Chapter 5. We present a transformation method that preserves the self-stabilization property of a self-stabilizing algorithm under a much more powerful scheduler than the one for which the algorithm has been designed for. The transformation method forms the bedrock of lifting composition. We also show how the structure of proof artifacts can be exploited to increase the concurrency in the transformed algorithm.

Chapter 6. The transformation method and lifting composition are mated with each other in Chapter 6 to define a generic composition method for self-stabilizing algorithms. Compositional methods suited to specific conditions are defined by using the structure of proof artifacts to instantiate the generic lifting composition. Furthermore, we provide extensions of lifting composition that preserve the self-stabilization even if the communication paradigm is changed. The scope of lifting composition is extended even further by defining variants which work in spite of variable dependencies.

Chapter 7. We provided a summary of the contributions of this work. Potential extensions of our results are also proposed prior to concluding the dissertation.

2

Self-Stabilizing Distributed Algorithms

We dwell on the fundamental concepts used to construct the compositional infrastructure in this chapter. We begin with the underlying system model and briefly explain the various constituents of a distributed system. The focus then shifts on self-stabilization and the significance of self-stabilization in designing non-masking fault tolerant distributed systems. The chapter ends with a survey of weaker forms of convergence.

2.1 Process Model

A distributed system consists of multiple computers interacting with each other using some form of communication infrastructure. Such a constituent set of mutually interacting computers may consist of computers which are located at different facilities.

Definition 2.1 (Process [14]). *A process refers to an instance of the implementation of an algorithm running on a constituent computer of a distributed system.*

A process, thus, encompasses the program counter, the source code and the memory registers used by an implementation of an algorithm. Note that, unless explicitly stated otherwise, throughout this text a process synonymously refers to the computer running an algorithm.

Definition 2.2 (Neighbor). *Two processes are termed as neighbors if they can interact with each other directly.*

The neighbor relation over the set of constituent processes of a distributed system, as defined above, is symmetric, *i.e.*, process P_i is neighbor of process P_j implies that process P_j is neighbor of process P_i. There are instances of distributed systems, *e.g.* wireless sensor networks with directional antennas, where neighbor relation is not symmetric [15]. In such networks, a process might have *in-neighbors* and *out-neighbors* [16]. A process can only receive data from an in-neighbor and it can only send data to a out-neighbor. However, in scope of this work we consider distributed systems with a symmetric neighbor relation. Communication infrastructure is typically abstracted away with help of a *communicaton graph* while reasoning about the properties of a distributed system.

Definition 2.3 (Communication Graph). *A communication graph representing a distributed system is an undirected graph $G = (V, E)$ such that set of nodes V is equal to the set of constituent processes and each edge in the set E connects the pair of nodes representing neighboring processes.*

The term *node* is often used interchangeably with the term *process* in the literature. The communication graph of a distributed system with an asymmetric neighbor relation is a directed graph with the directed edges denoting flow of data between the neighboring processes.

2.1.1 Communication Model

Given the vastness of communication media supporting distributed systems, it becomes imperative to use abstract models while analyzing distributed systems. There are two prevalent communication models assumed while designing distributed systems, namely, the *message passing model* and the *shared memory model*.

Message Passing Model

In the message passing model two neighboring processes interact by exchanging messages over directed point-to-point communication channels. Each process maintains a message queue to buffer incoming messages. It is usually assumed that each communication channel has finite message capacity and that messages are delivered in FIFO fashion to the message buffer. Messages are sent using a `send(m)` primitive and received using a `receive(m)` primitive. As shown in Figure 2.1, the message queue is part of the process that uses it to receive messages from its neighbor.

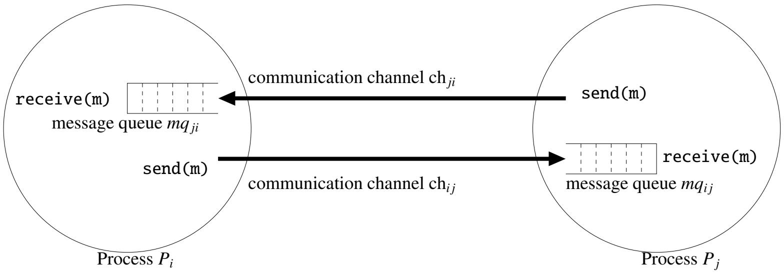

Fig. 2.1: Message-Passing Model

Shared Memory Model

In shared memory model, neighboring processes interact by reading and writing to shared memory registers. The shared memory registers used for interaction are termed as *communication registers*. Communication registers are not used for any purpose other than the exchange of data. Figure 2.2 shows how communication takes place between two neighboring processes P_i and P_j in the shared memory model. The set of communication registers of each process is divided into two classes: *read reigsters* and *write registers*.

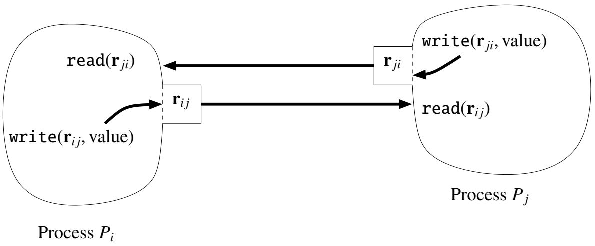

Fig. 2.2: Shared-Memory Model

Definition 2.4 (Read Register). *A read register is a shared-memory register which is used to receive the information sent by a neighbor process. A process P_j accesses a read register using a* `read`$(\mathbf{r}_{ij})$ *primitive.*

Definition 2.5 (Write Register). *A write register is a shared memory register which is used by a process P_i to send information to its neighbor via* `write`$(\mathbf{r}_{ij}, value)$ *primitive.*

A write register is said to be "owned" by the process which has exclusive right to write in it and, therefore, a write register –as depicted in Figure 2.2– is a part of the memory space of the process that owns it. Similarly a read register is read by a single process which uses it to gauge the current state of the process which owns it. A write register of a process is the read register of its neighbor. Although due to syntactical correctness the read register of a process has to appear in its code space, the read register does not belong to process that reads it. There are variants of shared memory models that assume communication primitives other than `read` and `write` primitives such as `read-modify-write` [17], `fetch&add` and `test&set` [18]. However, Herlihy [19] showed that these primitives are *computationally weaker* than `read` and `write` primitives.

We now take a slight detour in order to emphasize that the seemingly incompatibility of the two communication models can be bridged with the help of the emulator methods presented in the literature.

Relative Merits of Shared Memory and Message Passing Models

Message passing models are closer to implementation scenarios because more often than not processes in a distributed system communicate by transmitting messages. However, designing distributed systems assuming the message passing model is non-trivial, because the designer has to consider various parameters such as the size of message queues and delay induced by the channel. The channel might also change the order of the messages. The design task becomes even more intricate if the solution is required to be immune to failures of both, processes and communication channels. Designing distributed systems assuming the shared memory model is relatively easier. The solution can be designed without considering message delays and failures can be projected on individual processes. Nonetheless, one has to ensure that the access to the communication registers is *atomic*.

Definition 2.6 (Atomicity [20]). *Atomicity of a communication register guarantees that* `read` *or* `write` *operations on it are carried out* sequentially *in some order –thereby prohibiting concurrent operations– in presence of multiple* `read` *or* `write` *requests,*

However, there exist algorithms which can emulate the behavior of an algorithm designed assuming shared memory model over a communication infrastructure that provides only `send` and `receive` primitives. The availability of such transformers often motivates designers to build a distributed system assuming the shared memory model though implementation might not offer the assumed communication primitives.

Emulation of the Shared Memory Model

Various emulators, *e.g.* [21, 22], exist that can emulate shared memory systems However, many of these emulators can not function correctly if constituent processes fail. As this work concerns itself with fault tolerant systems in general, we briefly describe an algorithm by Attiya *et al.* [23] that can emulate shared memory system as long as majority of processes do not fail.

The primary idea behind the algorithm of Attiya *et al.* is to maintain the copies of every shared memory register at all of the n processes and guarantees that this implementation is atomic. The algorithm also ensures that a read operation on a shared memory registers fetches the result of the latest preceding write operation. The algorithm uses an integer variable *counter* to carry out `read` and `write` operations. When a process wants to carry out a `write` operation over a shared memory register, it selects the smallest value of *counter* not assigned yet and sends the value to be written and the value of *counter* to all the other processes in the system. Every process, on the receipt of the message, (1) updates the local copy of the shared memory register (2) updates the value of its local *counter*, and (3) sends back acknowledgement. A `write` operation is deemed complete by the initiating process when it receives acknowledgement that a majority of the processes have a *counter* value equal to its own. A process initiating a `read` operation over a shared memory register sends request to all other processes with its latest local value of the shared memory register and *counter*. All other process, in turn, reply back with latest value of the shared-memory register and *counter*. The initiating process selects the value with the largest *counter* when it receives replies from a majority of the processes. In the next step, the initiating process sends the value with largest *counter* and the value of *counter* to all other processes. The `read` operation is complete when the initiating processes receives acknowledgement to the effect that the majority of the processes have *counter* value equal to at least that of its own. The algorithm works correctly as long as at the most $f < \lceil n/2 \rceil$ processes fail. Figure 2.3 illustrates a `read` and a `write` operation in a system with five processes. The writer process sends a `write` message with new value and current counter to all the processes. Two of the processes (P_1 and P_4) fail to respond to the message, nevertheless, `write` operation succeeds as majority of processes send back the acknowledgement. Similarly, a majority of processes respond to the `read` message and, consequently, reader process copies the value with highest *counter*. The algorithm bounds the value of *counter* –and, hence, referred to as bounded emulator– by keeping track of *counter* values used in the system. The following theorem summarizes the seminal property of the emulator algorithm.

Theorem 2.1 ([23]). *There exists a bounded emulator of an atomic, single-write multi-reader register in an arbitrary network in the presence of link failures that do not disconnect a majority of processes.*

We now describe process structure used in this work and give the associated definitions.

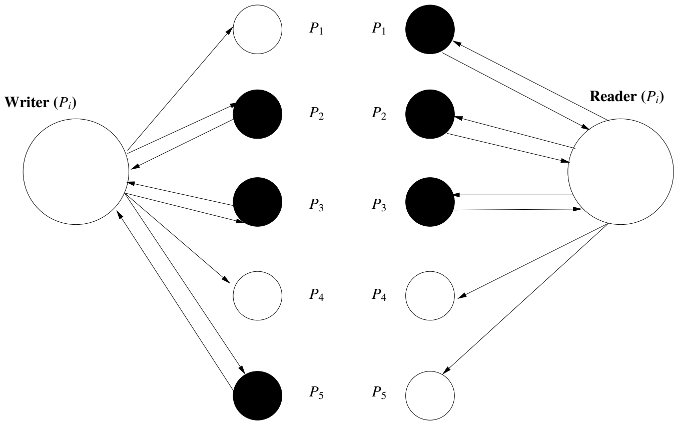

Fig. 2.3: Shared-Memory Emulator Execution [23]

2.1.2 Guarded Commands

Actions of every process P_i in a distributed system are specified as a set of *guarded commands*.

Definition 2.7 (Guarded Command [24]). *A guarded command is a program statement that consists of a label, a guard and an assignment expression specified as follows:*

$$\langle \texttt{label} \rangle :: \langle \texttt{guard} \rangle \rightarrow \langle \texttt{expression} \rangle \, ;$$

where (1) ⟨label⟩ is a process-wide unique identifier of a guarded command, (2) ⟨guard⟩ is a boolean expression over the variables belonging to the process and communication registers of its neighbors and (3) ⟨expression⟩ is a set of assignment functions over the variables of the process and its communication registers.

The behavior of every process are governed by the *sub-algorithm* it implements.

Definition 2.8 (Sub-Algorithm [24]). *A sub-algorithm $\mathcal{A}_i$ implemented by a process P_i is specified as a finite set of guarded commands specified as follows*

```
do
      ⟨label_1⟩ :: ⟨guard_1⟩ → ⟨assignment_1⟩;
                          ⋮
  []  ⟨label_n⟩ :: ⟨guard_n⟩ → ⟨assignment_n⟩;
od    while (true)
```

where '[]' represents non-deterministic *choice between various guarded commands.*

Definition 2.9 (Enabled Guarded Command). *A guarded command is said to be* enabled *if the boolean expression in its guard is true.*

Definition 2.10 (Execution of a Guarded Command). *A guarded command is said to be* executed *by a process if it is enabled and local variables of the process are modified according to the assignment functions specified in the guarded command.*

Definition 2.11 (Enabled Process). *A process is said to be* enabled *if it has* at least *one enabled guarded command*

Definition 2.12 (Local State of a Process). *Local state s_i of a process P_i at any time instant is the Cartesian product of the current values of its local variables and the contents stored in the communication registers it owns.*

$$s_i = \langle x_1, x_2, \cdots, x_n \rangle$$

In the following section we provide definitions pertaining to distributed algorithms and their properties.

2.2 Distributed Algorithm

Definition 2.13 (Distributed Algorithm). *Let $\Pi = \{P_1, P_2, \cdots, P_n\}$ be a set of processes such that each process runs a sub-algorithm $\mathcal{A}_{i|i=1,\cdots,n}$. A distributed algorithm $\mathbb{A}$ run by set Π is the union of sub-algorithms run by all the processes in Π.*

$$\mathbb{A} = \bigcup_{i=1}^{n} \mathcal{A}_i$$

Remark 2.1. A distributed algorithm $\mathbb{A}$ can alternatively be defined as a set of guarded commands where the set consists of all the guarded commands which can be potentially executed by the constituent processes. More specifically,

$$\mathbb{A} = \{\mathcal{G}_{Ai_j} \mid \mathcal{G}_{Ai_j} \in \mathcal{A}_i \ \wedge \ i \in \{1, \cdots, n\} \ \wedge \ j \in \{1, \cdots, l_i\}\},$$

where n is the total number of processes and l_i is the total number of guarded commands in sub-algorithm $\mathcal{A}_i$.

The set of sub-algorithms constituting a distributed algorithm can be heterogeneous in the sense that each of the constituent sub-algorithm may be unique. On the other end of the spectrum there are distributed algorithms where every sub-algorithm is similar modulo the communication registers. That is to say, apart from change in the communication registers representing the neighborhood of a process, guarded commands of all the sub-algorithms are identical. Such distributed algorithms are sometimes referred to as *uniform* distributed algorithms [25].

A critical attribute of distributed algorithms is the assumption about the process identifiers. For example, uniform distributed algorithms do not assume that constituent processes have a unique identifier. In scope of this work, we assume that every process has a system-wide unique identifier represented as an integer unless otherwise stated.

We have hitherto used the term *distributed system* informally describing a set of mutually interacting processes. We are now ready to give a more structured definition.

Definition 2.14 (Distributed System). *A distributed system is specified by (1) a finite set of process $\Pi = \{P_1, P_2, \cdots, P_n\}$, (2) an underlying communication infrastructure used by the processes to interact with each other, and (3) a distributed algorithm implemented by the set Π.*

We use the terms distributed system and distributed algorithm interchangeably in this work where the underlying communication model and the set of process is obvious from the context.

Definition 2.15 (Global System State). *Global state σ of a distributed system is a vector comprised of local states of its constituent processes.*

$$\sigma \ = \ \langle s_1, s_2, \cdots, s_n \rangle$$

2.3 Non-Determinism and Schedulers

The complexity of designing and analyzing a distributed algorithm is compounded by the fact that at any time instant more than one process may have enabled guarded commands. The presence of multiple enabled guarded commands makes distributed algorithms inherently *non-deterministic* because at first sight it is not obvious which of the enabled guarded commands are executed. Alternatively, a global state of a distributed system may have multiple enabled guarded commands. Thus, non-determinism in the context of distributed algorithms implies the presence of more than one enabled process in any global system state.

2.3.1 Execution Semantics

In order to reduce the inherent complexity of the analysis of a distributed algorithm it is imperative to reasonably limit the "amount of non-deterministic behavior" that the algorithm can exhibit. Therefore, various notions of execution semantics have been defined to precisely bound the amount of non-determinism while analyzing distributed algorithms.

Definition 2.16 (Serialized Execution Semantics [26, 3]). *In* serialized execution semantics*, in each global state* exactly *one of the enabled processes executes its enabled guarded command.*

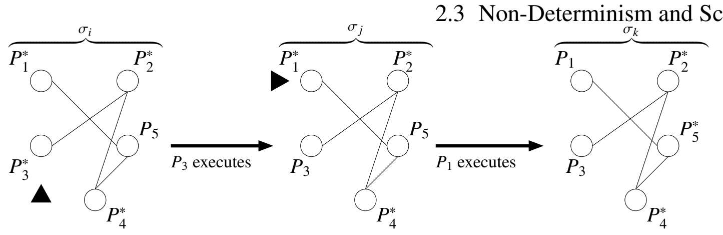

Fig. 2.4: Serialized Execution Semantics

Figure 2.4 shows three execution steps (*cf.* Defintion 2.10) of an example distributed system under serialized execution semantics. In state σ_i processes $P_1,\ P_2,\ P_3$ and P_4 (marked with '$*$') are enabled. However only process P_3 executes (marked by the '▲') its guarded commands. Similarly, in state σ_j out of processes $P_1,\ P_2$ and P_4 only process P_1 executes a guarded command. The serialized execution semantics is understandably the simplest execution model. It is primarily used during the design phase to prove that a distributed algorithm fulfills its specifications. Serialized execution semantics is often further strengthened by increasing the granularity of process' actions. It is assumed that a process, under serialized execution semantics, 1) reads the contents of communication registers, 2) does some local computation and, 3) writes the communication registers in one go and these operations are *indistinguishable* to an external observer. Such an execution model is referred to as *composite atomicity serialized execution model* [27].

Definition 2.17 (Parallel Execution Semantics [28, 29]). *In* parallel execution semantics, every *enabled process executes its enabled guarded command in every global state.*

Figure 2.5 shows execution of an example distributed system under parallel execution semantics. In states σ_i and σ_j all the enabled processes execute their guarded command. The parallel execution semantics may also be strengthened with composite atomicity of the processes' actions. *Synchronous execution semantics* [29] is a variant of parallel execution semantics and ensures that *all* the processes in the system execute their guarded command in every state.

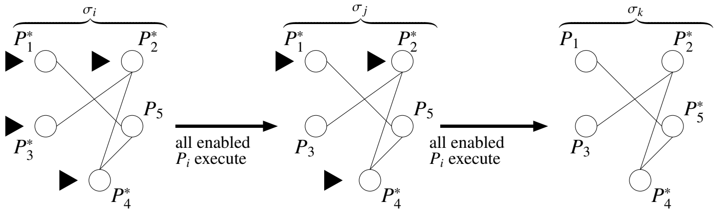

Fig. 2.5: Parallel Execution Semantics

Definition 2.18 (Distributed Execution Semantics [30]). *In* distributed execution semantics*, in every global state,* a subset *of enabled processes execute their guarded command. A process can either read the contents of communication register or write to a communication register after doing local computation during the execution of a guarded command.*

Figure 2.6 shows the execution of a distributed system under distributed execution semantics. In state σ_i, only three enabled processes ($P_1,\ P_2$ and P_4) perform read operations on their respective communication registers. Similarly, in state σ_j processes P_2 and P_4 complete write operations on communication registers. The distributed execution semantics explicitly precludes composite atomicity. However, *synchronous distributed semantics* [28] –a variant of distributed execution semantics– allows composite atomicity as long as a strict subset of enabled processes execute their guarded commands in every state.

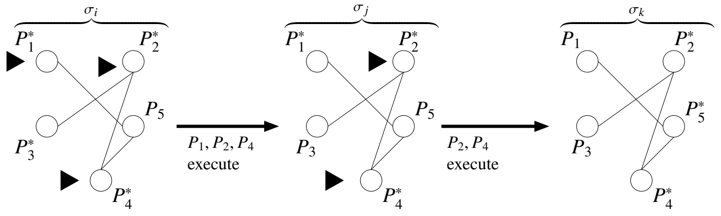

Fig. 2.6: Distributed Scheduler Semantics

The distributed execution semantics is closest to the actual implementation scenarios, however verifying properties of distributed algorithms under this model is not trivial. The relation between the serialized execution semantics and the distributed semantics can be intuitively seen as refinement mapping [26]. The serialized semantics is used in the design phase whereas the distributed execution semantics is used to reason about the behavior of a distributed algorithm in an implementation scenario. Nevertheless, properties proven under the serialized semantics do not transfer trivially to the system under the distributed semantics. Rigorous proofs and transformation methods are required to guarantee transfer of properties between these execution semantics. Figure 2.7 summarizes the relationship between various execution semantics. The rows correspond to the number of enabled processes –denoted by c– performing an execution in any global system state with n enabled processes, while the columns correspond to the atomicity of each execution step.

Atomicity **#(Processes)**	**Read/Write Atomicity**	**Composite Atomicity**
$c = 1$	Serialized Semantics [26]	Composite Atomicity serialized Semantics [27].
$c \leq n$	Distributed Semantics [30]	Synchronous Distributed Semantics [28]
$c = n$	–	Parallel Semantics [29]

Fig. 2.7: Comparison of Various Execution Semantics

2.3.2 Schedulers, Properties and Fairness

The motivation behind characterizing execution semantics, communication infrastructure and structure of a process is to analyze and predict the behavior of a distributed algorithm. However, precise definition of a behavior or property of a distributed algorithm is needed to utilize the concepts defined so far.

Definition 2.19 (Execution). *An execution Ξ of a distributed algorithm is a, possibly infinite, sequence of global system states:*

$$\Xi = \langle\ \sigma_0, \sigma_1, \cdots, \sigma_i, \cdots \rangle$$

such that σ_0 is the inital state; thereafter every subsequent state σ_i results from the execution of a subset of enabled guarded commands in the previous state σ_{i-1}.

Note that above definition holds for any of the execution semantics defined in Section 2.3.1 as the subset of guarded commands executed in each state can be specified accordingly.

Definition 2.20 (Maximal Execution). *A maximal execution $\hat{\Xi}$ is either 1) a finite execution such that no guarded command is enabled in the last state* or *2) an infinite execution.*

A finite execution can be converted into an infinite one by repeating the final state if need arises. An execution is also synonymously referred to as “trace” or “path” in the literature.

Definition 2.21 (Property). *A property Pr of a distributed algorithm is a set of* infinite *sequences of the global system states.*

Definition 2.22 (Property Satisfaction $\models$). *An execution Ξ of a distributed algorithm* satisfies *a property Pr, i. e. $\Xi \models Pr$, if*

$$\Xi \;\in\; Pr$$

Ξ belongs to the set of infinite global system state sequences defining property Pr.

A distributed algorithm, more often than not, produces multiple executions from a single initial state due to inherent non-determinism. Hence, it is important to characterize the satisfaction of a property by a distributed algorithm as such.

Definition 2.23 (Property Satisfaction $\models$ for a Distributed Algorithm). *Let $exec(\mathbb{A})$ be the set of all possible executions of a distributed algorithm $\mathbb{A}$. A distributed algorithm $\mathbb{A}$ satisfies a property Pr, i. e. $\mathbb{A} \models Pr$, if all of its possible executions belong to the property Pr.*

$$\mathbb{A} \models Pr \;\equiv\; (\forall\, \Xi : \Xi \in exec(\mathbb{A})) : \Xi \models Pr$$

Properties of distributed algorithms are typically specified using predicates in various flavors of temporal logic [31] rather than specifying them explicitly as a set of state sequences.

The properties of distributed algorithms are divided broadly into two categories, namely *safety* properties and *liveness* properties, to ease their analysis and verification.

Definition 2.24 (Safety Property [32]). *Let Σ be the set of all possible global system states and Σ^{ω} be the set of all possible infinite sequences of global system states. Let ϵ_i denote the prefix of length i of an element ϵ of Σ^{ω}. A property Pr is said to be a* safety property *if and only if the following condition holds:*

$$\forall \epsilon \in \Sigma^{\omega} : \; \epsilon \not\models Pr \;\Rightarrow\; (\exists i : \; 0 \leq i : \; (\forall \beta : \; \beta \in \Sigma^{\omega} : \epsilon_i \beta \not\models Pr))$$

where $\epsilon_i\beta$ denotes the concatenation of the two sequences.

Informally, if an infinite sequence of global states ϵ does not satisfy a property Pr then, there exists a prefix ϵ_i of ϵ, such that no extension of ϵ_i –obtained by concatenating any infinite global system sequence– satisfies the property Pr. A safety property characterizes the executions ensuring “no bad thing happens.” Thus, if an execution does not belong to a safety property then, after a certain point (or a particular system state) the “bad thing” must manifest itself in the execution. As the intuitive definition says that “bad thing” must never happen, therefore, once an execution contains an instance of a “bad thing” it can never satisfy a safety property. For example, consider a junction with traffic lights and the sequence in which the lights turn green. A system state of the traffic junction could be the Cartesian product of state of the traffic lights . The safety property of the traffic junction would be set of all sequences where not more than one traffic light is green at same instant. In this case, “bad thing” would be the state where more than one traffic light is green, since once that happens, no matter how traffic lights operate afterwards, the sequence of traffic lights states will not satisfy the property.

Definition 2.25 (Liveness Property [32]). *Let Σ be the set of all possible global system states and Σ^* be the set of all possible finite sequences of global system states. Let Σ^{ω} be the set of all possible infinite sequences of system states. A property Pr is said to be a* liveness property *if and only if the following condition holds:*

$$\forall \alpha : \alpha \in \Sigma^* : \ (\exists \beta : \ \beta \in \Sigma^\omega : \ \alpha\beta \models Pr)$$

Informally, a property Pr is a liveness property if for every finite sequence of global system states α there exists an infinite sequence of global system states β such that concatenation of α with β satisfies property Pr. A liveness property specify executions which ensure that "eventually something good happens." This implies that every execution *per se* is "remediable," that is, any finite sequence of states that has no "good thing" can be extended such that the resulting (infinite) sequence has at least one instance of "good thing." Consider an elevator servicing requests to move to various floors in a building. Let $request_i$ represent the state in which a button is pushed at floor i and $serve\_floor_i$ denote the state in which the elevator door opens at floor i. The sequences of the states of the elevator represent a liveness property because eventually door opens at the floor where a button is pressed. More precisely, any sequence of the elevator states with an unserviced request $request_i$ has a corresponding $serve\_floor_i$. Thus, any finite sequence of the elevator states can be extended by adding missing $serve\_floor_i$ states.

Two different notions of liveness exist in literature, namely *uniform liveness* [33] and *absolute liveness* [34]; uniform liveness requires that a *single* (infinite) execution can be concatenated with any finite sequence such that the resultant sequence satisfies a liveness property. The definition of absolute liveness stipulates that *any* (infinite) execution can be appended to all finite sequences. However, Alpern and Schneider [35] showed that these definitions are restrictive in the sense that these two notions of liveness exclude some properties that are intuitively liveness properties.

Safety and liveness properties do not characterize all the properties that a distributed algorithm can exhibit. There are properties which are neither safety properties nor liveness properties; for instance, properties which state "eventually a distributed system will do a good thing and thereafter will keep doing it" are neither safety nor liveness properties. However, as theorem below states, it is possible to express every property as conjunction of a safety and a liveness property.

Theorem 2.2 (Decomposition of a property [35, 32]). *Every property Pr is the intersection of a safety property and a liveness property.*

Fairness

It is typically shown that the set of all possible executions of a distributed algorithm forms a subset of the executions specified by a property while verifying that the algorithm satisfies the property. However, a distributed algorithm might exhibit some executions which are not realistic. Such "pathological" executions might incorrectly show that a distributed algorithm does not satisfy a property during the verification phase. The notion of *fairness* is used at the semantic level to avoid such pitfalls while verifying a distributed algorithm.

Fairness itself is a liveness property [36]. It entails that if a process or guarded command has remained enabled for *sufficiently* long, then the process or the guarded command is executed *frequently* enough. Various notions of fairness are defined depending on how often a process or a guarded command has been enabled.

Definition 2.26 (Unconditional Fairness [37]). *An execution is unconditionally fair iff every process executes its guarded commands infinitely often.*

Unconditional fairness is the *weakest* notion of fairness because, it doesn't put any restrictions on how often a process or a guarded command must be enabled to be executed. However, this notion of fairness is only feasible for the distributed systems where constituent processes don't terminate, since, the executions where a terminated process never executes after a certain point violate unconditional fairness [38].

Definition 2.27 (Weak Fairness [39]). *An execution is weakly fair iff a continuously enabled process executes its guarded commands infinitely often.*

Note that, a process is enabled if it has at least one enabed guarded command (*cf.* 2.11). Weak fairness is also referred to as *justice* [31].

Definition 2.28 (Strong Fairness [39]). *An execution is strongly fair iff a infinitely often enabled process executes its guarded commands infinitely often.*

A strongly fair execution is also weakly fair however *vice versa* is not true. Thus, the set consisting of the set of all strongly fair executions and the set of all weakly fair executions are totally ordered by inclusion.

The notions of fairness described so far require that a guarded command or a process must be enabled long enough (or at all) before it can be executed. This requirement however fails in scenarios where a guarded command in a process may not be enabled due to "race conditions." These conditions can arise in systems where a processes requires the availability of multiple resources in a single global system state to execute. Such processes can be denied enabling of a guarded command by making subsets of necessary resources available in infinitely many states but never all the resources in a single state. Consider, for example, the famous *dining philosophers problem* [40]. Two philosophers (Philosphers P_i and P_j in Figure 2.8) can use forks alternatively to ensure that their common neighbor (Philospher P_k in Figure 2.8) never gets the two forks in a single system state. Thus, no fairness notion would help as the action is not enabled at all. In order to circumvent scenarios where an action is not enabled *purely* due to race conditions, the notion of *hyperfairness* is used. The notion of hyperfairness, unlike other notions of fairness, is defined over the set of all possible execution of a distributed system. We say a global system state σ_i is reachable from global system state σ_j, if there exist a sequence of execution steps which starts in state σ_j and ends in σ_i and all the intermediate execution steps are due to the execution of guarded commands by constituent processes.

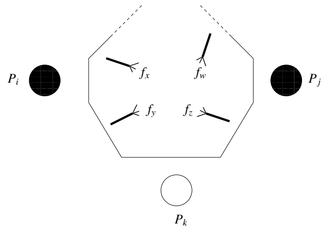

Fig. 2.8: Race Condition in Dining Philosophers Problem

Definition 2.29 (Hyperfairness [41]). *The set of executions of a distributed system is* not *hyperfair with respect to a guarded command $\mathcal{G}$ iff $\mathcal{G}$ is executed at most finitely often and for each system state σ_i there exists a state σ_j that is reachable from σ_i and $\mathcal{G}$ is enabled in σ_j.*

The notion of hyperfairness was introduced by Attie *et al.* [26] and later generalized in [42]. However, hyperfairness in [26] was defined for a *multi-party interaction* [43, cf. Chapter 14] –a communication primitive that abstracts away from communication between two processes.

The notion of hyperfairness is similar to the notion of 0-transition fairness [44] in the sense that both notions of fairness rule out race conditions. A notion of fairness, that is strictly stronger than hyperfairness [41], called ∞-fairness, is defined in [45]. ∞-fairness requires that an action that is *reachable* infinitely often in an execution is executed infinitely often. A hyperfair execution is also strongly fair, however, *vice versa* does not hold [41].

Scheduler

An abstract entity referred to as *scheduler* is used during the design phase of a distributed algorithm to combine the execution semantics and fairness characteristics of the target implementation scenario. A scheduler, essentially, “selects” a subset of enabled process in every state without violating the assumed fairness constraints.

Definition 2.30 (Scheduler). *Let ρ_i be an element of 2^{Π}, where Π is the set of the constituent processes. Let ϱ_i be a sequence such that $\varrho_i = \langle \rho_i, \rho_j, \rho_k, \cdots \rangle$. A scheduler $\mathbb{D}$ is defined as a countable infinite set $\mathbb{D} = \{\varrho_1, \varrho_2, \cdots, \varrho_i, \cdots\}$.*

Each element ϱ_i of $\mathbb{D}$ is also termed as a *strategy* of scheduler $\mathbb{D}$. In case a scheduler $\mathbb{D}$ applies a strategy ϱ_i on a distributed algorithm, the strategy ϱ_i manifests itself “indirectly” as an execution Ξ of the algorithm.

An enabled process may have more than one guarded command. In such cases, intra-process fairness constraints are used to select the guarded command to be executed. However, such intra-process non-determinism is transparent to a scheduler.

Fairness constraints are embedded in the scheduler in the sense that, a scheduler $\mathbb{D}$ does not have a strategy that manifests into an execution that violates the desired fairness criterion. A suitable execution semantics is ensured by putting a restriction on the size of subset ρ that a scheduler can choose in a state. For instance, a scheduler can be constrained to select only one process in each step thereby ensuring serialized execution semantics. A scheduler $\mathbb{D}_A$ is said to be *stronger* than another scheduler $\mathbb{D}_B$ if the constraints imposed on $\mathbb{D}_A$ for selecting a strategy are *weaker* than the constraints imposed on scheduler $\mathbb{D}_B$.

A suitable scheduler can be "devised" by combining the target execution semantics and a feasible fairness constraint. Figure 2.9 shows some often-used schedulers as combinations of their respective execution semantics and fairness notion in the “scheduler space.” For example, a serialized weakly fair scheduler $\mathbb{D}_{swf}$ chooses a single process in each global system state while ensuring that no continuously enabled process is ignored forever.

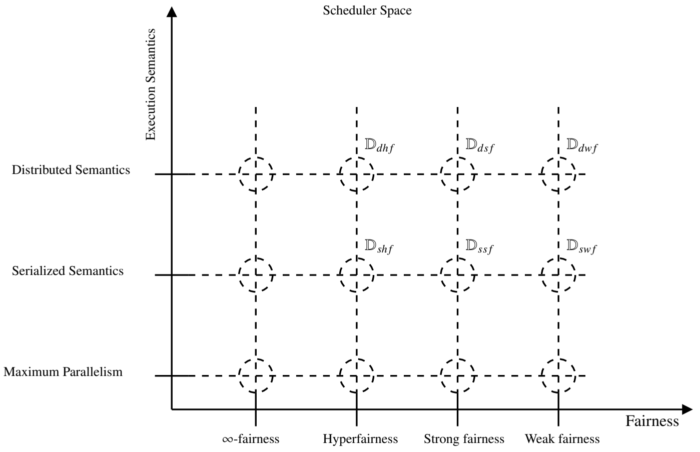

Fig. 2.9: Various Canonical Schedulers

2.4 Self-Stabilization

We have thus far focused on the definitions pertaining to basic model and the general abstract concepts used while designing and verifying distributed algorithms. We now focus on a very specific property of distributed algorithms, namely, *self-stabilization*. Typically, properties of distributed algorithms are expressed with the help of predicates, wherein a distributed algorithm satisfies a property if variables belonging to the constituent process satisfy the corresponding predicate after a certain point in all possible executions. Likewise, we formally define self-stabilization with respect to a predicate $\mathcal{P}$ in the following.

Definition 2.31 (Self-Stabilizing System [46]). *A system S is* self-stabilizing *with respect to a predicate $\mathcal{P}$ if and only if it satisfies the following two properties:*

Closure*: An execution starting in a system state satisfying predicate $\mathcal{P}$ never reaches a state that does not satisfy $\mathcal{P}$.*

Convergence*: Any execution starting in an arbitrary system state is guaranteed to reach a state satisfying the predicate $\mathcal{P}$ in finite number of execution steps.*

Closure is a safety property because it ordains that a self-stabilizing system should never falsify the predicate $\mathcal{P}$ *–once it holds–* thus, ensuring in a certain sense that "nothing bad" ever happens during the execution after system reaches a state satisfying $\mathcal{P}$. Figure 2.10 depicts state space of a self-stabilizing system such that gray states satisfy a certain predicate $\mathcal{P}$ (thus representing “good states”) and black states do not (*i.e.* representing “bad states”). Transitions between individual states are represented by arrowed lines. Thus, if any execution starting in a good state is traced in Figure 2.10, it never reaches a bad state owing to the closure property of the self-stabilizing system. Convergence is a liveness property because it stipulates that eventually a self-stabilizing system *always* reaches the set of states satisfying predicate $\mathcal{P}$. Thus, any execution of a self-stabilizing system has a suffix with states satisfying $\mathcal{P}$. For example, consider Figure 2.11 which shows state space and transitions of a self-stabilizing system. In case the system starts in a bad state (*e.g.*, σ_{b1}), it always reaches a good state (*e.g.*, σ_g) irrespective of intermediate transitions. Self-stabilization has been defined for a distributed system instead of a distributed algorithm, because it is clear from the context that self-stabilization is a property of a distributed algorithm that manifests itself under a certain execution environment. However, it also implies, as discussed later, that self-stabilization often depends on the underlying execution environment.

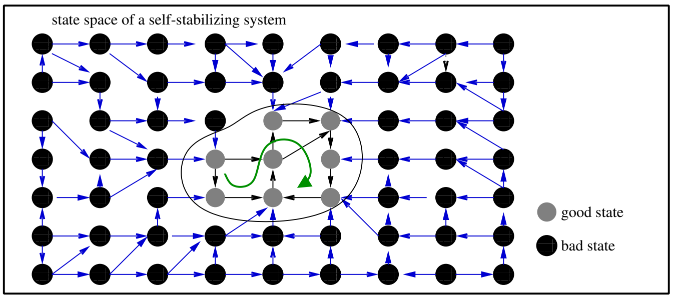

Fig. 2.10: Closure Property of a Self-Stabilizing Algorithm

Predicate $\mathcal{P}$ is typically a state predicate. The system states satisfying $\mathcal{P}$ are usually referred to as *legal* (or *legitimate*) states and all other states are termed as *illegal* (or *illegitimate*) states. Predicate $\mathcal{P}$ is sometimes also called *safety* predicate of a self-stabilizing system.

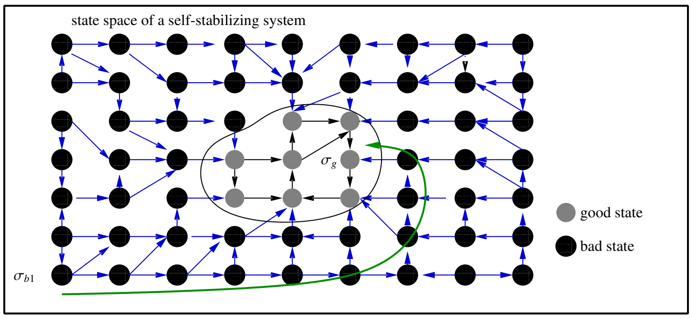

Fig. 2.11: Convergence Property of a Self-Stabilizing Algorithm

Example 2.1 (Self-stabilizing token ring algorithm [3]). Consider the self-stabilizing token ring algorithm (see Figure 2.12) devised by Dijkstra [3]. The algorithm is defined over a ring of n processes. The ring has one "distinguished process" (P_0) and all other processes (P_i, $1 \leq i \leq n-1$) execute identical sub-algorithms modulo variable names. The task of the algorithm is to ensure that in any system state only a single process has an enabled guarded command and, thus, can change its local state (also referred to as "privilege"). Hence, the safety predicate $\mathcal{P}_{\mathcal{LE}}$ of Dijkstra's algorithm holds true for all the states where only a single process has an enabled guarded command. It is assumed that the algorithm is executed using the shared memory communication model under a weakly-fair serialized scheduler. The distinguished process continuously compares its local state (an integer x_0) with that of its left neighbor and if they are equal, then P_0 increments x_0 modulo the number of processes in the ring. A non-distinguished process P_i also continuously compares its local state with that of its left neighbor and if they are not equal, then it copies the local state of its left neighbor. The algorithm is self-stabilizing

```
Process $P_0$
        local var integer $x_0$;
        do
            $x_0 = x_{n-1} \rightarrow x_0 := (x_0 + 1) \bmod n$;
        while (true)

Process $P_i$ $(1 \leq i \leq n-1)$
        local var integer $x_i$;
        do
            $x_i \neq x_{i-1} \rightarrow x_i := x_{i-1}$;
        while (true)
```

Fig. 2.12: Self-Stabilizing Token Ring Algorithm of [3]

with respect to the predicate $\mathcal{P}_{\mathcal{LE}}$ under any weakly-fair serialized scheduler. That is, irrespective of the starting state, the algorithm reaches a state such that only a single process has its guarded command enabled and thereafter never reaches a state where more than one process have an enabled guarded command. Figure 2.13 shows an example execution of the token ring algorithm. The algorithm is run on a ring with five processes. The initial state $\sigma_1 = \langle 1, 2, 3, 4, 1 \rangle$ is an illegal state because all the processes have a privilege. Process P_1 is selected by the scheduler to take a step resulting in state σ_2 with multiple privileges. The system reaches state σ_5 after processes P_3, P_2, and P_4 are selected by the scheduler in the states σ_2, σ_3, and σ_4, respectively. Note that $\sigma_5 = \langle 1, 1, 1, 3, 3 \rangle$ is a legal state because only P_3 has an enabled guarded command. The system remains in legal state thereafter, as only one process has an enabled guarded command in subsequent states –for instance, only P_0 has an enabled guarded command in state σ_7.

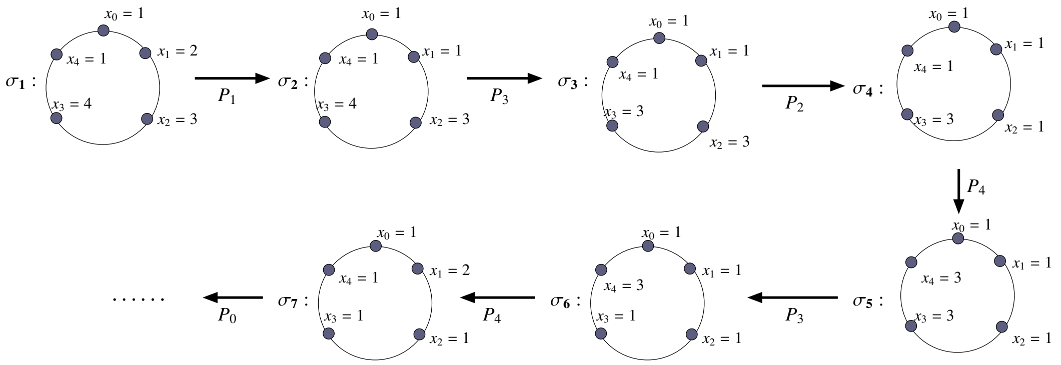

Fig. 2.13: Sample Execution of the Self-Stabilizing Token Ring Algorithm

We have defined self-stabilization as a property of distributed algorithms. However, we have not discussed the practical significance of self-stabilization. In the next section, we discuss how self-stabilization helps in designing fault-tolerant distributed systems.

2.5 Self-Stabilization and Fault Tolerance

During the design phase of a distributed system, it is precisely specified what properties it must exhibit in order to function correctly. System designers also strive to ensure that a distributed system functions correctly even if some of the constituent processes do not function properly. A distributed system must, therefore, fulfill certain properties in order to function correctly when constituent processes function incorrectly. Self-stabilization is one of such desirable properties, however, few definitions are in order before we delve on the importance of self-stabilizing systems.

Definition 2.32 (Failure [47]). *A failure is an event which occurs when a distributed system deviates from its specified functionality.*

A failure of a distributed system signifies its incorrect behavior and scenarios where it does not deliver the intended service.

Definition 2.33 (Error [47]). *An error is a part of the system state that can potentially cause a failure.*

Definition 2.34 (Fault [47, 48]). *A fault is the cause of an error at the lowest level of abstraction.*

A fault can potentially cause an error if it becomes *active*. An execution step or a change in system environment may activate a fault. A fault that has not been activated is termed as a *dormant* fault [47]. An error manifests itself as a failure if it can be detected by an external observer of the system [48]. An error that remains undetected is called a *latent* error [47]. We are now ready to define a fault-tolerant distributed system.

Definition 2.35 (Fault-Tolerant Distributed System [2]). *A distributed system is fault tolerant to a certain class of faults F if it continues to function correctly according to its specification despite the occurrence of the faults from the fault class F.*

A fault tolerant distributed system delivers correct service as long as faults from a specified class of faults occur. Faults are divided into various classes. A fault class typically specifies a set of fault actions. Faults can also be divided into two broad classes based on their duration, namely, *transient* and *permanent* faults.

Definition 2.36 (Transient Fault [48, 47]). *A transient fault is a fault whose duration is bounded in time.*

Definition 2.37 (Permanent Fault [48, 47]). *A permanent fault is a fault whose duration –once it appears– is continuous and unbounded in time.*

A distributed system that is tolerant to a certain class of faults satisfies its safety and liveness properties as long as faults from the fault class occur, because every property of a distributed system can be expressed as conjunction of a safety and a liveness property. However, this condition might be violated if faults which do not belong to the fault class occur. The response of a distributed systems to faults divide them into three classes: *masking*, *fail-safe* and *non-masking* fault tolerant systems.

Definition 2.38 (Masking Fault-Tolerant System [2]). *A masking fault-tolerant system preserves both safety and liveness properties under a given class of faults.*

Definition 2.39 (Non-masking Fault-Tolerant System [2]). *A non-masking fault-tolerant system preserves only liveness properties under a given class of faults.*

Masking fault tolerant distributed systems are more attractive from operational perspective because, they provide tight guarantees of desired service and, a user does not experience any service outage. However, more often than not designing such systems is expensive and difficult and, under some conditions, even impossible [49, 50]. Non-masking fault tolerant systems present a viable alternative for such scenarios as long as specification allows for incorrect behavior for bounded time periods.

Self-stabilization for Non-masking Fault Tolerance

Let $\mathbb{D}$ be a self-stabilizing distributed system such that program code of each process is immutable [51]. Suppose that $\mathbb{D}$ runs in an environment which is prone to bursts of transient faults. Thus, each burst of transient faults may leave memory contents and communication registers of the constituent processes in an arbitrary state. Recall that a self-stabilizing system satisfies its safety predicate within finite time irrespective of the initial state. Thus, $\mathbb{D}$ will satisfy its safety predicate provided two bursts of transient faults are sufficiently separated in time. Alternatively, a self-stabilizing is an “eventually safe” system. This implies that a self-stabilizing system preserves its liveness (convergence) property under the class of transient faults. Hence self-stabilization can be used to construct non-masking fault-tolerant systems under transient faults.

2.6 Weaker Forms of Convergence

The attractiveness of self-stabilizing systems lies in the fact that they do not depend on an initial state for correct behavior. However, this characteristics renders them some of the most difficult systems to design because a designer must consider *all possible states* and *all possible transitions* emanating from them. In order to circumvent this difficulty, various weaker forms of convergence have been proposed. Indeed, for many problems, distributed systems with weaker convergence properties provide sufficient fault-tolerance [52, 53]. Additionally, it is easier to prove that a distributed system satisfies a weaker form of convergence.

Definition 2.40 (Weak Convergence [53]). *A distributed system* $\mathbb{D}$ weakly converges *to a predicate* $\mathcal{P}$ *iff, for every state* σ*, there exists an execution* Ξ *of* $\mathbb{D}$ *such that* Ξ *starts in* σ *and has a state which satisfies* $\mathcal{P}$.

Definition 2.41 (Weakly-Stabilizing System [53]). *A system W is weakly-stabilizing with respect to a predicate* $\mathcal{P}$ *iff it satisfies 1) closure and 2)* weak convergence *with respect to* $\mathcal{P}$.

Definition 2.42 (Pseudo Convergence [52]). *A distributed system* $\mathbb{D}$ pseudo-converges *to a predicate* $\mathcal{P}$ *iff for every execution* Ξ *of* $\mathbb{D}$*, there exists an integer i such that* Ξ_i *satisfies the predicate* $\mathcal{P}$*, where* Ξ_i *is a suffix of* Ξ *starting in* i^{th} *state.*

A self-stabilizing system satisfies its specification when it eventually reaches the states satisfying safety predicate. A pseudo-stabilizing system *eventually* reaches a state after which it *does not* violate its specification unlike a self-stabilizing system which *cannot* violate its specification. Pseudo-stabilization does not bound the number of steps to reach a state after which system does not violate its specification. Pseudo-stabilization is weaker than self-stabilization implying a self-stabilizing system is also a pseudo-stabilizing system, however the reverse does not hold.

Definition 2.43 (Probabilistic Convergence [54]). *A distributed system* $\mathbb{D}$ *probabilistically converges to a predicate* $\mathcal{P}$ *iff, for every system state* σ*, every execution of* $\mathbb{D}$ *reaches a state that satisfies the predicate* $\mathcal{P}$ *with probability* 1.

Probabilistic stabilization is used to provide self-stabilizing solutions to the problems which have no deterministic solutions such as token circulation in anonymous rings [55]. Randomization –as used by probabilistic self-stabilizing systems– breaks the symmetry without using auxiliary variables or communication registers.

2.7 Summary

We introduced the system model and the relevant definition to prepare groundwork for the introduction of self-stabilization. Furthermore, we discussed the importance of self-stabilization in the context of fault-tolerant distributed systems. We also provided an overview of weaker forms of convergence which are used for the systems where a self-stabilizing solution is unwieldy to design.

3

Design and Verification of Self-Stabilizing Algorithms

Self-stabilizing systems constitute a large class of non-masking fault tolerant systems. However, designing a self-stabilizing system is not trivial because, it involves constructing a system which is live *and eventually* safe. This chapter surveys the techniques used for verification of self-stabilizing systems, that is, designing provably correct self-stabilizing systems. Furthermore, we examine the compositional mechanisms provided in the literature for designing self-stabilizing systems. The chapter concludes by making a case for a richer compositional framework.

3.1 Verification Techiques for Self-Stabilizing Algorithms

The very property of self-stabilizing systems, that is critical for realizing non-masking fault tolerance, makes their verification rather difficult. In particular, proving that a self-stabilizing system converges to the set of legal states is not trivial. This has led to various methods being proposed to (automatically) verify self-stabilization ever since the property was first defined. The verification methods found in the literature can be divided into three major categories: *deductive* verification methods, *algorithmic* verification methods, *control-theoretic* verification methods and, *term-rewriting* based methods. We now briefly describe methods belonging to each of the three categories.

3.1.1 Algorithmic Verification Techniques

Symbolic Model Checking

Algorithmic verification techniques, also referred to as *model checking*, are used to verify systems with finite state spaces. *Symbolic model checking* [56] is used in [57] to verify self-stabilizing systems. A self-stabilizing system is formally specified as a SMV [56] program. Closure and convergence properties to be verified are specified in branching time temporal logic CTL [58]. The model checker SMV represents the state space of a given self-stabilizing system as ordered binary decision diagrams, thereby reducing the memory required to complete the verification. In case a given system does not satisfy desired closure and convergence properties, SMV presents a “counterexample” showing an execution where the properties do not hold.

Model Checking with Parametrization

A method to verify *parametrized* self-stabilizing systems, *i. e.* systems with arbitrary number of processes, is presented in [59]. The method abstracts away an arbitrary number of processes with help of a *network invariant*. A network invariant is used to represent all the processes that are similar upto their identifiers. Convergence and closure properties and the system to be verified are specified in TLA+ [60], and the explicit state model checker TLC [60] is used to verify the abstracted system.

Discussion

Algorithmic verification techniques provide a completely automatic method for verifying self-stabilizing systems. They also provide a counterexample in case the given system is not self-stabilizing. However, such methods are intended for finite state systems. Unfortunately, self-stabilizing systems can have infinite state space. For example, a self-stabilizing algorithm designed for message-passing communication model with unbounded channels has infinite number of states [61, 62]. Another challenge is *state space explosion*; the state space grows exponentially as systems grow larger. For instance, in [57], SMV could not verify systems with more than seven processes. Although abstraction can be used to reduce the state space, generally the abstraction step cannot be automated.

3.1.2 Deductive Verification Techniques

Theorem proving methods form the core of deductive verification techniques. As these methods are amenable to systems with infinite state space, they have been used to verify self-stabilizing systems.

Temporal logic based verification

A proof system based on temporal logic is proposed in [63]. The method uses *fair transition systems* [31] to formally describe a given self-stabilizing system. Closure and convergence properties of a self-stabilizing system are specified via LTL^{-} [64], which is a future fragment of *linear temporal logic*. The proof system is completed by providing proof rules in linear temporal logic. The proof rules exploit certain semantic peculiarities of self-stabilizing systems. More specifically, generic temporal logic proof rules for proving closure are simplified by the property that self-stabilizing systems do not have a specific initial state, and generic proof rules for convergence are simplified for systems that are closed under a certain predicate. The proof system uses the theorem prover *PVS* [65] to check whether a given system satisfies closure and convergence properties.

UNITY based verification

A deductive verification built on UNITY programming logic [43] is presented in [66]. A self-stabilizing system is formally specified as a UNITY program. The standard theorems of UNITY are extended to verify self-stabilizing systems. A new progress operator is defined for proving theorems related to the convergence property of self-stabilizing systems. This new operator also permits "encoding" a well-founded relation used to exhibit progress towards a "stable" predicate. The progress operator is further strengthened to allow compositional reasoning of the convergence property of a self-stabilizing system. The extended UNITY theorems also allow to prove convergence properties by partitioning executions of a self-stabilizing system into rounds. Convergence is subsequently proven by showing that only a finite number of such rounds exist. This proof system requires support of the theorem prover HOL [67] to draw complete formal proofs.

Discussion

Deductive verification techniques, in general, require considerable support from an algorithm designer during the verification phase. The algorithm designer is required to supply auxiliary intermediate constructs to guide a theorem prover along with the proof rules. More specifically, auxiliary invariants are required while proving a closure property and a well-founded relation is needed to prove a convergence property. Thus, the deductive techniques for self-stabilizing systems need expert external input, although they can handle infinite state systems.

3.1.3 Term Rewrite Systems based Technique

Beauquier *et al.* [68] showed that convergence of self-stabilizing systems can be verified using term rewrite methods [69]. The method represents global system states as words of a formal language. State transitions are represented by rewrite rules in the resultant length-preserving term rewrite system. A convergence property of a given self-stabilizing system is proven by showing the nonexistence of a cyclic derivation in the resultant rewrite system for words corresponding to illegal states. The non-cycle property is shown to equivalent to the inexistence of infinite derivations starting with illegal states in the reverse rewrite system. The non-cycle property can be shown by showing the existence of a well-founded relation defined over the words.

The method cannot be automated completely as it requires heuristics to define the well-founded relation to show the inexistence of cycles. This requirement also implies that the method can only be employed by expert users.

3.1.4 Control-Theoretic Verification Techniques

The methods described so far are, in essence, the modified forms of the methods that are found in the body of work on formal verification. An interesting alternative has been presented in [70] by showing that *non-linear feedback control systems* [71] are analogous to self-stabilizing systems. More specifically, it is shown that the convergence property of self-stabilizing systems is analogous to the *asymptotic global stability* of non-linear feedback systems. Informally, a non-linear feedback system is asymptotically globally stable if it *converges* from any point in state space to a unique *equilibrium point* and the system does not deviate from the equilibrium point once it is reached. A asymptotically stable feedback system, thus, also exhibits, in a certain sense, convergence as well as closure property. This similarity allows the usage of the control-theoretic methods meant for analyzing non-linear feedback systems to verify self-stabilizing systems. Such methods have been modified for verifying self-stabilizing systems in [72, 73]. These contol-theory based methods were further generalized in [74, 75] by modeling self-stabilizing systems as *hybrid systems* [76] and using Lyapunov functions [77] for verifying them. The control-theoretic verification method is of particular interest in the scope of this discussion, because this method is derived from the techniques originally meant for proving the convergence property –*i.e.* asymptotic stability– of the systems with infinite state space. Furthermore, it also becomes apparent, that proving correctness of self-stabilizing systems is rather non- trivial despite the availability of such strong verification techniques. In the following, we provide a brief description of the verification method based on control theory.

The method models self-stabilizing algorithms as discrete hybrid systems. Hybrid systems are feedback control systems which have both continuous *and* discrete dynamics, and have different *modes* of control. A hybrid system evolves according to the differential equation governing a mode as long as the invariant corresponding to that model holds. A change in mode occurs if the invariant is falsified and the system "jumps" to the mode that satisfies the control mode jump condition. A trajectory of a hybrid system, thus, consists of *flows* interspersed by discrete jumps. Mode dynamics in discrete hybrid systems are governed by *difference equations* instead of differential equations. Thus, difference equations determine the behavior of a hybrid system as long as a mode invariant is satisfied.

Global asymptotic stability of hybrid systems is verified via an extension of the "Second Method" of Lyapunov [78]. The stability of hybrid systems is shown by showing the existence of a corresponding *Lyapunov function*. Intuitively, if every step of a system leads to a decrease in the "energy level" of the system, then the system must be able to come to rest irrespective of the initial starting point. A Lyapunov function captures this notion of decrease in energy level of a hybrid system. More specifically, if there exists a function $V(x)$ such that (1) $V(x_e) = 0$ at the equilibrium point x_e in the state space, (2) $V(x) > 0$ for all non-equilibrium points, and (3) $V(x)$ is decreasing along the *all* trajectories, *i. e.* $\forall x : \forall_{1 \leq i \leq m} i$:

$V(f_i(x)) - V(x) < 0$, such that $x[k+1] = f_i(x[k])$ defines the dynamics of the discrete-time system in mode i, then the given discrete-time hybrid system is globally asymptotically stable. Such a function $V(x)$ is referred to as Lyapunov function. The problem of searching a suitable Lyapunov function can be formulated as a *convex optimization* problem with help of *linear matrix inequalities* (LMI) [79], and thereby computed automatically, if the given hybrid system exhibits *affine* dynamics in each mode. Such hybrid systems are called *piecewise affine* hybrid systems.

A given system is modeled as a hybrid system in order to verify whether the system is self-stabilizing or not via a Lyapunov function. As distributed algorithms perform discrete operations in assignment parts, discrete hybrid systems are used to model distributed algorithms. The notion of time in the resultant hybrid system is represented by the selection of guarded command in the original algorithm. Thus, the "time ticks" whenever a guarded command is selected by the underlying scheduler. Every guarded command of every process in the system is represented in the discrete-time hybrid system via (1) a difference equation and (2) a control mode. A control mode of the hybrid system – modeling a distributed algorithm– corresponds to an enabled guarded command of the original system and the assignment statement of that guarded command is reflected in the difference equation of the control mode. A mode switch occurs when an enabled guarded command is executed and the next control mode is selected non-deterministically amongst control modes that represent the resulting system state. An extra mode per process is defined to capture states where none of the guarded commands of a process are enabled in order to ensure that the resultant discrete-time hybrid system has infinite time line. Thus, if a system has n processes and a process P_i has m_i guarded commands, then the resultant system has $\Sigma_1^n m_i + n$ control modes. A Lyapunov function for a hybrid system constructed as explained above can be computed automatically provided that the hybrid system is piecewise affine. The existence of a Lyapunov function for such a hybrid system shows that the original distributed algorithm is self-stabilizing. Note that convergence within finite number of steps follows if there exists a constant $k \in \mathbb{R}$, such that for all possible points x and x', $\|x - x'\| > k$ holds. For example, this constraint follows automatically for the systems with integers as variables.

In order to verify self-stabilizing systems which converge to a closed set of legal states and transit between these states in a *cyclic* manner –also referred to as *orbitally self-stabilizing* systems– an additional technique from control theory has been used. *Poincaré maps* [80] are used to prove that non-linear feedback control systems converge to stable limit cycles. The primary idea behind the method is to identify a hyperplane in the state space and to consider only those points of trajectory where it intersects with the chosen hyperplane. In case there exists an infinite number of intersection points between the hyperplane and trajectories, then stability analysis can be confined to the hyperplane instead of the whole state space, as follows. A non-linear system has a stable limit cycle if, for all trajectories, the infinite sequence of intersection points converges to a unique point on the hyperplane. This limit cycle forms a closed trajectory. The convergence of the sequence of intersection points can be shown via the existence of a Lypanunov function. Thus, if a hybrid system derived from a distributed algorithm has a stable limit cycle then, the original distributed algorithm is orbitally self-stabilizing.

Discussion

The control-theoretic techniques exploit the analogy and a much larger body of results to verify self-stabilizing systems. However, it can be employed to automatically verify only those self-stabilizing systems which can be represented as piecewise affine hybrid systems. Although there exist control-theoretic methods that can handle hybrid systems which are not piecewise affine hybrid systems, but given the inherent non-determinism in distributed systems, the gain brought in by these methods is not much. This owes to the fact that a stronger scheduler leads to an increase in the number of control modes in the resulting hybrid system.

3.2 Compositional Methods for Self-Stabilizing Systems

Verification of self-stabilizing algorithms is not trivial despite the support provided by formal methods. As the size of systems grow larger, it becomes progressively difficult to provide intermediate auxiliary constructs required by the formal methods. Various compositional techniques have been adopted in order to overcome scalability-related challenges encountered while designing large self-stabilizing systems. These compositional methods are discussed in the following.

3.2.1 Asymmetric Compositional Methods

Two *asymmetric* compositions schemes for self-stabilizing algorithms are presented in [81], namely, *selection composition* and *hierarchical composition*. These methods are asymmetric because some restrictions are put on the execution semantics of one of the two component algorithms.

Selection Composition

Selection composition is defined on two self-stabilizing algorithms which are *compatible* with each other. Compatibility of two component algorithms ensures that no two process write to a common variables. The composition uses a boolean vector of size n where n is the number of processes in the system. Each of the component is self-stabilizing with respect to their respective predicates. Let $\mathcal{P}_A$ and $\mathcal{P}_B$ denote the respective safety predicates of the component algorithms and $\mathbf{e}$ denotes the boolean vector. Each guarded command $\mathcal{G}_{\mathcal{A}_{ij}} \to a^A_{ij}$ of component algorithm $\mathbb{A}$ in a process P_i is modified to $\mathcal{G}_{\mathcal{A}_{ij}} \wedge \mathbf{e}_i \to a^A_{ij}$, where $\mathbf{e}_i$ is the i^{th} element of the boolean vector and j is the index of the guarded command in P_i. Similarly, each guarded command of the other component algorithm $\mathbb{B}$ is modified to $\mathcal{G}_{\mathcal{B}_{ij}} \wedge \neg\mathbf{e}_i \to a^B_{ii}$. The composed algorithm consists of the union of the modified guarded commands of both components and the boolean vector $\mathbf{e}$. The composed algorithm is self-stabilizing with respect to predicate $\mathcal{P}_A$ if all elements of $\mathbf{e}$ have truth value *True*. The composed algorithm is self-stabilizing with respect to predicate $\mathcal{P}_B$ if the truth value of all elements of vector $\mathbf{e}$ is *False*. Thus, a Boolean vector $\mathbf{e}$ can be used to modulate the self-stabilization of a composed algorithm.

Hierarchical Composition

Hierarchical composition is defined over two self-stabilizing algorithms as well. However, this composition method requires that one of the components "controls" the other component. A component algorithm $\mathbb{A}$ controls component algorithm $\mathbb{B}$ iff $\mathbb{B}$ uses a variable of $\mathbb{A}$ as *input variable* to execute its guarded commands, however, it does not modify it. Guarded commands of component $\mathbb{B}$ in each process P_i are modified to $\mathcal{G}_{\mathcal{B}_{ij}} \wedge \mathbf{idle}(A_i) \to a^A_{ij}$ where $\mathbf{idle}(A_i)$ denote the *conjunction of negated guards* of component $\mathbb{B}$ in process P_i. The composed algorithm consists of the union of guarded commands of $\mathbb{A}$ and the modified guarded commands of $\mathbb{A}$. Let algorithm $\mathbb{A}$ be self-stabilizing to predicate $\mathcal{P}_A$ and $\mathbb{B}$ converges to $\mathcal{P}_B$ provided $\mathcal{P}_A$ holds true for its input variables and all guards of $\mathbb{A}$ become disabled once it converges to $\mathcal{P}_A$. Then, the composed algorithm converges to predicate $\mathcal{P}_B$ if component $\mathbb{B}$ is *fair* in the sense that actions of $\mathbb{B}$ in process P_i can block execution of actions of $\mathbb{A}$ in any other process.

Detector-Corrector based Composition

A compositional method to design self-stabilizing algorithms from algorithms that may not be self-stabilizing via the use of *coordinators* is presented in [82]. This technique builds upon the method for designing fault tolerant programs using *correctors* and *detectors* [83]. A detector is an algorithm that "detects" whether a given state falsifies a certain predicate. A corrector, in addition to detecting a predicate, sets the system to state to a pre-defined "correct" state if the predicate is not satisfied. The primary idea is to isolate and potentially "block" a component so that it cannot corrupt variables of other component algorithms. The task of isolating and blocking is performed by a *coordinator* which is the linchpin of this composition technique. The composition technique exploits the dependency between the components. It defines a coordinator for each component which continuously monitors the respective component with the help of a relevant detector and in case an anomaly is noticed by the detector, it uses the corrector to correct itself and other components. The structure of graph representing dependencies between various components determines the structure of coordinator employed. Two types of dependencies are used: a *correction dependency* and a *corruption dependency*. A component $\mathbb{A}$ depends on component $\mathbb{B}$ for correction if an action of $\mathbb{A}$ can correct $\mathbb{A}$ if $\mathbb{B}$ is already in a correct state. A component $\mathbb{A}$ can corrupt $\mathbb{B}$ if an action of $\mathbb{A}$ in a state in which $\mathbb{A}$ is incorrect and $\mathbb{B}$ is correct leads to a state where both components are in an incorrect state. Each component consists of its own guarded commands and coordinator guarded commands. A component communicates with its neighboring components via *method calls*, wherein the component communicating with its neighbor invokes a method on the callee.

3.2.2 Symmetric Composition

Unlike the composition schemes discussed before, symmetric composition [84] does not restrict the execution semantics of the component algorithms in the sense that, one of the two component algorithms execution steps are restricted or regulated by its counterpart. Symmetric composition is defined over two *compatible and suffix-closed* self-stabilizing algorithms. Two self-stabilizing algorithms are said to be compatible in this compositional framework if none of the algorithms uses variables belonging to the other algorithm. A distributed algorithm is said to be suffix-closed if, for every execution of the algorithm, any suffix of the execution is also an execution of the algorithm. Let $\mathbb{A}$ and $\mathbb{B}$ be two suffix-closed compatible algorithms self-stabilizing with respect to predicates $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively. Symmetric composition of algorithms $\mathbb{A}$ and $\mathbb{B}$, defined as the union of guarded commands of $\mathbb{A}$ and $\mathbb{B}$, is self-stabilizing with respect to the predicate $\mathcal{P}_A \wedge \mathcal{P}_B$.

3.3 Summary

Composition has been presented as an alternative design methodology to overcome scalability issues encountered while verifying self-stabilizing algorithms. However, techniques found in the literature assume that the component algorithms are self-stabilizing under the same class of schedulers and, hence, leave a large class of self-stabilizing algorithms outside their gamut. More specifically, such techniques cannot be used to compose self-stabilizing algorithms, which are self-stabilizing under different class of schedulers. Additionally, these compositional methods cannot handle algorithms that exhibit weaker forms of convergence. Furthermore, some of the compositional methods cannot be used to compose self-stabilizing algorithms which are not silent. Thus, the compositional techniques available in the literature –in addition to being rather limited in number– cannot be used to compose a considerable number of self-stabilizing algorithms. This raises the question, whether it is possible to overcome these limitations while leaving resultant algorithm with optimal non-determinism. We analyze these aspects of compositional design of self-stabilizing algorithms in the following chapters.

4

Lifting Composition of Self-Stabilizing Algorithms

4.1 Introduction

While designing a self-stabilizing algorithm, the problem specification typically states the safety predicate and the underlying scheduler. Should the algorithm designer decide to take the compositional route – which is true more often than not due to the complexity of proofs – compatibility of schedulers of potential component algorithms become a critical design challenge. The problem can be rather acute given the fact that some algorithms are self-stabilizing under very specific schedulers. This problem can be alternatively formulated to devise a composition scheme that is oblivious to respective schedulers. This is, indeed, a rather strong condition to be forced on a compositional operator. However, the prior knowledge in form of proofs of the self-stabilization properties of component algorithms can be used to define such an operator. This exploitation of self-stabilization proofs in order to effect the compositional transformation of self-stabilizing algorithms forms the kernel of our endeavor in this chapter.

Outline

In this chapter, we investigate whether usual compatibility requirements of schedulers can be transcended during the composition. To that end, we define *lifting composition* of self- stabilizing algorithms. Lifting composition transfers the self-stabilization property of a component algorithm $\mathbb{B}$ such that $\mathbb{B}$ exhibits self-stabilizing behavior in the composed algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ under the scheduler of the other component algorithm $\mathbb{A}$. We further analyze the effect of the relationship between the respective schedulers on the result of lifting composition.

This chapter is structured as follows. In Section 4.2, we briefly recall relevant definitions. Section 4.3 contains the definition of lifting composition and shows that it preserves the self-stabilization property of the component algorithms. Section 4.4 delves on the implication of the relationship between schedulers on the result of lifting composition. Two self-stabilizing algorithms are constructed in Section 4.5 with the help of the new compositional technique. The chapter ends with a summary in Section 4.6.

4.2 System Model

A distributed system consists of a set $\Pi := \{P_1, \cdots, P_n\}$ of n processes communicating with each other via shared memory registers. A process consists of write registers, read registers, local variables and the sub-algorithm it executes. The local variables are the memory registers used by a process for internal computation. Each process P_i has a system wide unique identifier i.

A process P_i executes a sub-algorithm $\mathcal{A}_i$, which is defined as a set of guarded commands. Distributed algorithm $\mathbb{A}$ executed by a distributed system is the union of the sub-algorithms executed by all the processes in Π. The *local state* s_i of a process P_i consists of the valuation of its local variables and write registers.

The *global system state* of a distributed system is a vector $\sigma = \langle s_1, \cdots, s_n \rangle$ whose elements are the local states of all the processes in the system (see Definition 2.15). The set of all possible global system states σ_i of a distributed algorithm is termed as *global system state space* Σ.

Inherent non-determinism of a distributed algorithm is resolved with the help of a scheduler which generates (interleaved) executions of the distributed algorithm. A scheduler is defined as a set of strategies and each strategy corresponds to the sequence in which enabled processes are activated (see Definition 2.30). Additional constraints are imposed on a scheduler in form of fairness restrictions. A weakly fair scheduler ensures that a continuously enabled guarded command is activated infinitely often.

4.3 Lifting Composition

We present a new composition operation, called *lifting composition*, for self-stabilizing algorithms, which preserves the self-stabilization property of the components.

4.3.1 Definitions

Component Algorithms

Lifting composition takes two self-stabilizing algorithms, $\mathbb{A}$ and $\mathbb{B}$, as operands. Component algorithm $\mathbb{A}$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$. $\mathbb{D}_A$ is a weakly-fair scheduler implying that in a maximal execution (see Definition 2.20) a continuously enabled process is invoked infinitely often. Furthermore, in every system state of algorithm $\mathbb{A}$, each process P_i has exactly one enabled guarded command.

Remark 4.1. The assumption that algorithm $\mathbb{A}$ has an enabled guarded command in each process in every system state is not restrictive. An algorithm which has none or more than one enabled guarded command in a process can be transformed to an algorithm which has exactly one enabled guarded command per process without modifying its specification [85]. For example, consider a process P_i which has two guarded commands $\mathcal{G}_{Ai_x}$ and $\mathcal{G}_{Ai_y}$ enabled in some state. If sub-algorithm $\mathcal{A}_i$ is modified by adding $\neg\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Ai_y}$ in place of $\mathcal{G}_{Ai_y}$ then, P_i will never have both guards enabled in same state. Instead, if a process P_i has no guarded command enabled in process then, a guarded command of the form $\bigwedge_{x=1}^{l_i} \neg\mathcal{G}_{Ai_x} \rightarrow$**skip**; can be added to P_i to ensure that a guarded command is enabled in every state.

Component algorithm $\mathbb{B}$ is self-stabilizing with respect to the predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Scheduler $\mathbb{D}_B$ is a serialized scheduler $\mathbb{D}_B$ activates exactly one of the enabled processes in every execution step. Scheduler $\mathbb{D}_A$ is assumed to preserve the closure property of algorithm $\mathbb{B}$: it implies that, under scheduler $\mathbb{D}_A$, any execution of algorithm $\mathbb{B}$ starting in a state satisfying predicate $\mathcal{P}_B$ never reaches a state where predicate $\mathcal{P}_B$ does not hold true. Component algorithms $\mathbb{A}$ and $\mathbb{B}$ have disjoint state spaces, that is, they do not have any shared variables or communication registers.

A prerequisite of the inference that algorithm $\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$ is the fact that algorithm $\mathbb{B}$ always converges to the states which satisfy $\mathcal{P}_B$. Convergence of a distributed algorithm is proven with the help of a well-foundedness argument. It is shown that, irrespective of the initial state, any execution of the algorithm can be projected on to a monotonous sequence of the elements of a well-founded set. More precisely, convergence is proven by showing existence of a suitable *ranking function*.

Definition 4.1 (Ranking Function [86]). *A ranking function $\varDelta$: $\varSigma \rightarrow \varTheta$ maps state space $\varSigma$ of a distributed algorithm to a well-founded set $\varTheta$ such that $\varDelta(\sigma_i) > \varDelta(\sigma_j)$ for any two states σ_i and σ_j if state σ_j is reachable from σ_i via a* single *execution step.*

The existence of such a function $\varDelta$, such that

1) $\varDelta(\sigma_i) > \varDelta(\sigma_j)$ for any execution step $\sigma_i \rightarrow \sigma_j$ in a state σ_i with $\sigma_i \not\models \mathcal{P}$ and
2) $\varDelta(\sigma_i)=\inf(\varTheta)$ if $\sigma_i \models \mathcal{P}$, where $\inf(\varTheta)$ is the minimum element of $\varTheta$,

is used to prove convergence of a distributed algorithm with respect to a predicate $\mathcal{P}$.

Let $\varDelta_B$ be a ranking function used to prove the convergence of algorithm $\mathbb{B}$ with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Thus, the value of function $\varDelta_B$ decreases under scheduler $\mathbb{D}_B$ no matter which process executes.

Remark 4.2. Note that it is possible to design an algorithm whose convergence as well as closure is provable only under a specific scheduler. However, for such an algorithm proof obligations of self-stabilization are usually stronger than otherwise. While the convergence proof obligation would still be existence of a ranking function, the closure proof would require that a distributed algorithm shows well-defined behavior even after it reaches a legal state. The proof artifact used to verify closure in such cases would be stronger than mere state invariants, and therefore, such algorithms are not considered in the scope of this work.

Composition

Algorithm $\mathbb{A}$ consists of n sub-algorithms $\{\mathcal{A}_i \mid 0 \leq i \leq n\}$. Sub-algorithm $\mathcal{A}_i$ is a set of l_i guarded commands $\{\mathcal{G}_{Ai_x} \mid 1 \leq x \leq l_i\}$. A set of n sub-algorithms $\{\mathcal{B}_i \mid 1 \leq i \leq n\}$ constitutes algorithm $\mathbb{B}$. Sub-algorithm $\mathcal{B}_i$ is a set of m_i guarded commands $\{\mathcal{G}_{Bi_x} \mid 1 \leq x \leq m_i\}$. Algorithms $\mathbb{A}$ and $\mathbb{B}$ consists of $\varSigma_{i=1}^{n} l_i$ and $\varSigma_{i=1}^{n} m_i$ guarded commands respectively.

Algorithm $\mathbb{B}$, by virtue of being self-stabilizing under $\mathbb{D}_B$, has a known ranking function $\varDelta_B$.

Definition 4.2 (Lookahead Value δ_{Bix}). *Consider a guarded command $\mathcal{G}_{Bi_x} \rightarrow act_{Bi_x}$ of sub-algorithm $\mathcal{B}_i$. Let v_{Bi_x} be the subset of the variables of the algorithm $\mathbb{B}$ whose values are changed by the assignment part act_{Bi_x} of the guard $\mathcal{G}_{Bi_x}$. Let $\varDelta_{Bi_x}$ be the function obtained from $\varDelta_B$ by replacing variables in v_{Bi_x} by their respective assignment expressions in act_{Bi_x}. δ_{Bix} is defined as*

$$\delta_{Bix} = \varDelta_B - \varDelta_{Bi_x}.$$

For example, consider a guarded command $\mathcal{X} :: (x_1 \leq 1) \rightarrow x_2 := f_1(x_1, x_2)$ and a ranking function $\varDelta = x_1^2 + x_2^2$; $\varDelta_{\mathcal{X}}$ corresponding to the guardeded command $\mathcal{X}$ is equal to $x_1^2 + (f_1(x_1, x_2))^2$.

We are now ready to formally define lifting composition. The lifting composition $\mathbb{A}^{\triangle}\mathbb{B}$ of algorithms $\mathbb{A}$ and $\mathbb{B}$ is defined as follows.

Definition 4.3 (Lifting Composition). *Sub-Algorithm $\mathcal{A}_i{}^{\triangle}\mathcal{B}_i$ run by a process P_i consists of $3 \cdot l_i \cdot m_i + l_i$ guarded commands of the following structure:*

$$\begin{array}{llll}
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge (\delta_{Bix} < 0) \wedge \neg\mathcal{P}_B & \rightarrow & act_{Ai_x}; act_{Bi_y}; & (1) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_B & \rightarrow & act_{Ai_x}; act_{Bi_y}; & (2) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge (\delta_{Bix} \geq 0) \wedge \neg\mathcal{P}_B & \rightarrow & act_{Ai_x}; & (3) \\
\mathcal{G}_{Ai_x} \wedge (\neg\mathcal{G}_{Bi_1} \wedge \cdots \wedge \neg\mathcal{G}_{Bi_{m_i}}) & \rightarrow & act_{Ai_x}; & (4)
\end{array}$$

for all $x \in \{1, \cdots, l_i\}$ and all $y \in \{1, \cdots, m_i\}$. Algorithm $\mathbb{A}^{\triangle}\mathbb{B}$ is the union of all sub-algorithms $\mathcal{A}_i{}^{\triangle}\mathcal{B}_i$

$$\mathbb{A}^{\triangle}\mathbb{B} = \bigcup_{i=1}^{n} \mathcal{A}_i{}^{\triangle}\mathcal{B}_i$$

run by the processes in $\varPi$.

Description

Every process P_i in the composed algorithm $\mathbb{A}\vartriangle\mathbb{B}$ has four types of guarded commands.

Guarded Commands of Type 1 Process P_i has $l_i \cdot m_i$ guarded commands of type 1. A guard of Type 1 is true in a system state if (1) constituent guards $\mathcal{G}_{Ai_x}$ and $\mathcal{G}_{Bi_y}$ are true, (2) δ_{Bi_x} corresponding to guard $\mathcal{G}_{Bi_y}$ is negative, and (3) safety predicate $\mathcal{P}_B$ of algorithm $\mathbb{B}$ does not hold in the state. If a guard of Type 1 is true and it is activated then, assignment statements belonging to parent guarded commands, act_{Ai_x} and act_{Bi_y} are executed. A Type 1 guard allows component algorithm $\mathbb{B}$ to converge towards the states satisfying predicate $\mathcal{P}_B$ without hindering the actions of algorithm $\mathbb{A}$.

Guarded Commands of Type 2 Every process has $l_i \cdot m_i$ guarded commands of Type 2. A Type 2 guard is true in a system state if (1) guards $\mathcal{G}_{Ai_x}$ and $\mathcal{G}_{Bi_y}$ hold true and (2) predicate $\mathcal{P}_B$ holds true in the state. If an enabled guarded command of Type 2 is selected then, assignment statements of parent guarded commands are executed. A Type 2 guarded command facilitates the joint execution of actions of both component algorithms once algorithm $\mathbb{B}$ reaches the states satisfying predicate $\mathcal{P}_B$.

Guarded Commands of Type 3 There are $l_i \cdot m_i$ guarded commands of Type 3 in each process P_i. A Type 3 guard is true in a state if (1) it does not satisfy predicate $\mathcal{P}_B$ (2) parent guards are true, and (3) δ_{Bi_x} is positive. If an enabled guarded command of Type 3 is selected then, assignment statement of guarded command $\mathcal{G}_{Ai_x}$ is executed. A Type 3 guarded command ensures that guards of algorithm $\mathbb{A}$ are not blocked in a process in case actions of algorithm $\mathbb{B}$ lead to an increase in the value of Δ_B.

Guarded Commands of Type 4 There are l_i guarded commands of Type 4 in each process P_i. A Type 4 guarded command is enabled if (1) corresponding parent guard is true and (2) none of the guards of sub-algorithm $\mathcal{B}_i$ are true. If an enabled guarded command of Type 4 is selected then, only the assignment statement of the parent guard act_{Ai_x} is executed. By virtue of a Type 4 guarded command, guards of algorithm $\mathbb{A}$ are enabled even if none of the guards of algorithm $\mathbb{B}$ are enabled.

Remark 4.3. Global information in every process P_i is generally required in order to reason about the sign of δ_{Ai_x} in any state and this information should be available in every execution step. In some cases the topology of the communication graph allows inspection of variables required for the calculation of Δ_B. Alternatively, extra algorithmic support can be added to acquire the global information. In the scope of this chapter we assume that global information is available in each execution.

4.3.2 Preservation of Self-Stabilization

We now show that self-stabilization properties of algorithms $\mathbb{A}$ and $\mathbb{B}$ are preserved in the lifting composition $\mathbb{A}\vartriangle\mathbb{B}$. The proof maps the executions of the composed algorithm $\mathbb{A}\vartriangle\mathbb{B}$ to the set of executions of each component. The proof consists of two parts. In the first part, it is shown that algorithm $\mathbb{A}\vartriangle\mathbb{B}$ can produce all possible executions of algorithm $\mathbb{A}$. In the latter part of the proof, we show that, in all executions of $\mathbb{A}\vartriangle\mathbb{B}$, the convergence property of $\mathbb{B}$ with respect to predicate $\mathcal{P}_B$ is preserved.

We provide a few definitions prior to the proof.

Definition 4.4 (Projection of a State). *Let σ be a global system state of a composed algorithm $\mathbb{A}\vartriangle\mathbb{B}$. Projection $\sigma_{|\mathbb{A}}$ of state σ over component algorithm $\mathbb{A}$ is obtained from σ by removing all the variables belonging to algorithm $\mathbb{B}$*

Definition 4.5 (Projection of an Execution). *Let $\hat{\Xi}_{\mathbb{A}\vartriangle\mathbb{B}} = \langle \cdots, \sigma_i, \sigma_j, \cdots \rangle$ be a maximal execution of a composed algorithm $\mathbb{A}\vartriangle\mathbb{B}$. Projection $\breve{\Xi}_{\mathbb{A}\vartriangle\mathbb{B}|\mathbb{A}}$ of maximal execution $\hat{\Xi}_{\mathbb{A}\vartriangle\mathbb{B}}$ over component algorithm $\mathbb{A}$ is obtained by replacing each global state of $\mathbb{A}\vartriangle\mathbb{B}$ by its projection over algorithm $\mathbb{A}$, i. e.,*

$$\breve{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}|\mathbb{A}} := \langle \cdots, \sigma_{i|\mathbb{A}}, \sigma_{j|\mathbb{A}}, \cdots \rangle.$$

The guards of the algorithm $\mathbb{A}$ are unaffected in the composed algorithm and therefore, the set of enabled guarded commands of $\mathbb{A}$ in the composed algorithm each state is unchanged.

Lemma 4.1. *The projection of any maximal execution of* $\mathbb{A}^{\vartriangle}\mathbb{B}$ *under* $\mathbb{D}_A$ *on* $\mathbb{A}$ *is a maximal execution of* $\mathbb{A}$ *under* $\mathbb{D}_A$.

Proof. Consider a maximal execution $\hat{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}}$ of $\mathbb{A}^{\vartriangle}\mathbb{B}$ under $\mathbb{D}_A$ and let $\sigma_i \rightarrow \sigma_j$ be a execution step in $\hat{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}}$. Let P_x be a process such that guard $\mathcal{G}_{Ax_t}$ is enabled in P_x in global state σ_i. If σ_i satisfies predicate $\mathcal{P}_B$ and a guard $\mathcal{G}_{Bx_w}$ of algorithm $\mathbb{B}$ is enabled, then a Type 2 guard is enabled in process P_x. If σ_i does not satisfy predicate $\mathcal{P}_B$, then, depending on the value of δ_{Bx_w}, a Type 1 or Type 3 guard is enabled. Should there be no guards of algorithm $\mathbb{B}$ enabled in σ_i, a Type 4 guard is enabled in process P_x. Moreover, the guards of $\mathbb{A}$ are not modified in any process P_x. Thus, the set of enabled guards of $\mathbb{A}$ in any process P_x in state σ_i and, therefore, the set of enabled processes in state σ_i is same as it would be in state $\sigma_{i|\mathbb{A}}$ if $\mathbb{A}$ was running alone. Thus, there exists a execution step $\sigma_{i|\mathbb{A}} \rightarrow \sigma_{j|\mathbb{A}}$ of $\mathbb{A}$ under $\mathbb{D}_A$ corresponding to $\sigma_i \rightarrow \sigma_j$, because assignment statements of $\mathbb{A}$ are unchanged in $\mathbb{A}^{\vartriangle}\mathbb{B}$ and $\mathbb{A}$ and $\mathbb{B}$ do not have shared variables. The lemma follows. □

Lemma 4.1 shows that, under scheduler $\mathbb{D}_A$, there exists a maximal execution of $\mathbb{A}$ corresponding to a maximal execution of the composed algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$. Figure 4.1 is an informal depiction of the relationship between the state spaces of algorithms $\mathbb{A}$ and $\mathbb{A}^{\vartriangle}\mathbb{B}$. Each state σ_i of composed algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ can be represented as a tuple $\sigma_i = \langle \sigma_{i|\mathbb{A}}, \sigma_{i|\mathbb{B}} \rangle$. Thus, as a result of the composition, the state space of algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ is a product of state spaces of algorithms $\mathbb{A}$ and $\mathbb{B}$. Note that the projection relation between states of algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ and algorithm $\mathbb{A}$ is a *surjective* relation. Consequently, multiple executions of algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ have a single projection in the state space of algorithm $\mathbb{A}$. For example, consider the execution starting in states $\sigma_{i,1}$, $\sigma_{i,2}$ and $\sigma_{i,3}$ of algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ in Figure 4.1. The projection of these states is state σ_i of algorithm $\mathbb{A}$. The projection of $\langle \sigma_{i,2} \rightarrow \sigma_{j,2} \rightarrow \sigma_{n,2} \rightarrow \sigma_{u,2} \rightarrow \cdots \rangle$ is the execution $\langle \sigma_i \rightarrow \sigma_j \rightarrow \sigma_n \rightarrow \sigma_u \rightarrow \cdots \rangle$ of algorithm $\mathbb{A}$. The projections of executions starting in states $\sigma_{i,1}$ and $\sigma_{i,3}$ are also $\langle \sigma_i \rightarrow \sigma_j \rightarrow \sigma_n \rightarrow \sigma_u \rightarrow \cdots \rangle$. The maximality of the algorithm $\mathbb{A}$ in the composed algorithm also implies preservation of predicates over the executions of $\mathbb{A}$.

Lemma 4.2. *If all maximal executions of algorithm* $\mathbb{A}$ *under scheduler* $\mathbb{D}_A$ *satisfy a predicate* $\mathcal{P}$*, then the projection of every execution of* $\mathbb{A}^{\vartriangle}\mathbb{B}$ *under scheduler* $\mathbb{D}_A$ *on* $\mathbb{A}$ *also satisfies predicate* $\mathcal{P}$.

Proof. The projection of every maximal execution of $\mathbb{A}^{\vartriangle}\mathbb{B}$ under scheduler $\mathbb{D}_A$ over algorithm $\mathbb{A}$ is a maximal execution of $\mathbb{A}$ under $\mathbb{D}_A$ as well (from Lemma 4.1). Let $\breve{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}|\mathbb{A}}$ be the projection of a maximal execution of $\mathbb{A}^{\vartriangle}\mathbb{B}$ over $\mathbb{A}$ such that $\breve{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}|\mathbb{A}}$ does not satisfy predicate $\mathcal{P}$. Thus, $\breve{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}|\mathbb{A}}$ is a maximal execution of $\mathbb{A}$ under $\mathbb{D}_A$ and $\breve{\Xi}_{\mathbb{A}^{\vartriangle}\mathbb{B}|\mathbb{A}}$ does not satisfy predicate $\mathcal{P}$. However, it is contrary to the assumption in the if-clause. This completes the lemma. □

Lifting composition preserves all the properties of algorithm $\mathbb{A}$, and if executions of $\mathbb{A}$ are closed with respect to some predicate, then the projection of any execution of $\mathbb{A}^{\vartriangle}\mathbb{B}$ is closed with respect to that predicate as well. Corollary 4.1 follows from Lemma 4.2 by substituting predicate $\mathcal{P}$ with the safety predicate $\mathcal{P}_A$ of algorithm $\mathbb{A}$.

Corollary 4.1. *If all maximal executions of* $\mathbb{A}$ *which start in a state satisfying predicate* $\mathcal{P}_A$ *do not enter a state that does not satisfy predicate* $\mathcal{P}_A$*, then the projection of all maximal executions of* $\mathbb{A}^{\vartriangle}\mathbb{B}$ *which start in a state whose projection on* $\mathbb{A}$ *satisfies predicate* $\mathcal{P}_A$ *does not enter a state whose projection on* $\mathbb{A}$ *does not satisfy* $\mathcal{P}_A$.

With the help of Lemmata 4.1 and 4.2, we next show the convergence of algorithm $\mathbb{A}$ towards predicate $\mathcal{P}_A$ is preserved by lifting composition.

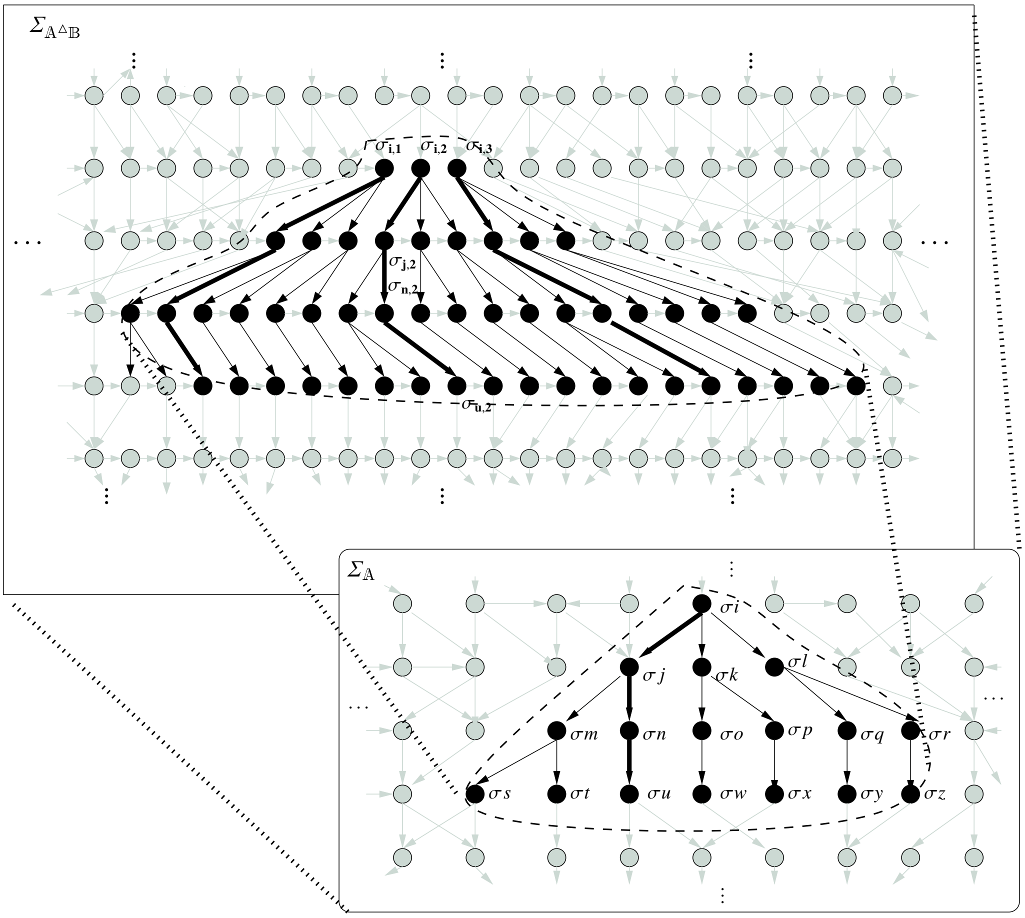

Fig. 4.1: Projection of Algorithm $\mathbb{A}\triangle\mathbb{B}$ over Algorithm $\mathbb{A}$

Theorem 4.1. *If all maximal executions of algorithm $\mathbb{A}$ converge to states where predicate $\mathcal{P}_A$ holds under scheduler $\mathbb{D}_A$, then all maximal executions of algorithm $\mathbb{A}\triangle\mathbb{B}$ under $\mathbb{D}_A$ also converge to the states whose projection on algorithm $\mathbb{A}$ satisfies predicate $\mathcal{P}_A$.*

Proof. Lifting composition does not change the result of any execution of algorithm $\mathbb{A}$ (from Lemma 4.1). Thus, if a maximal execution of $\mathbb{A}$ reaches a state satisfying $\mathcal{P}_A$ then projection of a maximal execution of $\mathbb{A}\triangle\mathbb{B}$ on $\mathbb{A}$ reaches a state satisfying $\mathcal{P}_A$ as well(Corollary 4.1).

Because the set of enabled guarded commands in every state in $\mathbb{A}\triangle\mathbb{B}$ remains the same (from Lemma 4.1), scheduler $\mathbb{D}_A$ cannot increase the number of execution steps required to reach a state whose projection on $\mathbb{A}$ satisfies predicate $\mathcal{P}_A$. Thus, the number of execution steps required by $\mathbb{A}\triangle\mathbb{B}$ to reach a state whose projection on $\mathbb{A}$ satisfies $\mathcal{P}_A$ under $\mathbb{D}_A$ is equal to the number required by $\mathbb{A}$ to reach a state which satisfies $\mathcal{P}_A$. □

As shown above, lifting composition simply "pastes" the executions of algorithm $\mathbb{A}$ in the state space of the composed algorithm $\mathbb{A}\triangle\mathbb{B}$. Consequently, lifting composition allows algorithm $\mathbb{A}$ to exhibit all the properties that it exhibits executing alone under scheduler $\mathbb{D}_A$.

We now focus on the behavior of algorithm $\mathbb{B}$ in the composed algorithm $\mathbb{A}\triangle\mathbb{B}$. It is shown that the embedding of ranking function Δ_B of algorithm $\mathbb{B}$ in algorithm $\mathbb{A}\triangle\mathbb{B}$ ensures that the self-stabilization property of $\mathbb{B}$ is preserved provided that scheduler $\mathbb{D}_A$ is weakly-fair.We first show that lifting composition preserves the maximality of algorithm $\mathbb{B}$.

Lemma 4.3. *The projection of every maximal execution of the composed algorithm* $\mathbb{A}\triangle\mathbb{B}$ *under a weakly-fair scheduler* $\mathbb{D}_A$ *on algorithm* $\mathbb{B}$ *is a maximal execution of* $\mathbb{B}$ *under scheduler* $\mathbb{D}_B$*.*

Proof. Algorithm $\mathbb{B}$ has a convergence property with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$ which is proven with help of ranking function $\varDelta_B$. The proof is organized in three parts: 1) we first show that the actions of algorithm $\mathbb{B}$ are not disabled due to composition in the composed algorithm, 2) the actions of $\mathbb{B}$ are unaltered and, 3) progress of $\mathbb{B}$ is unhindered.
Non-existence of a deadlock. The convergence property implies that in every state of $\mathbb{B}$ which does not satisfy $\mathcal{P}_B$ there is *least one* process with an enabled guard such that the execution of an enabled guarded command leads to a decrease in the value of $\varDelta_B$. Thus, in every state σ_i of the maximal execution $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ which does not satisfy $\mathcal{P}_B$, there exists at least one process P_x such that the Boolean expression $\mathcal{G}_{Bx_\kappa} \wedge (\delta_{Bix} < 0)$ evaluates to true.
Exclusivity of actions of $\mathbb{B}$. Consider an execution step $\sigma_i \rightarrow \sigma_j$ in $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ such that $\sigma_{i|\mathbb{B}}$ is not equal to $\sigma_{j|\mathbb{B}}$. Then, $\sigma_i \rightarrow \sigma_j$ involves an execution of a guarded command of $\mathbb{B}$, because assignment statements of $\mathbb{A}$ and $\mathbb{B}$ are unchanged during composition.
Maximality of projection. Let $\varepsilon = \langle \cdots, \sigma_\iota, \sigma_\kappa \rangle$ be a prefix of a weakly fair maximal execution $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ of $\mathbb{A}\triangle\mathbb{B}$ under $\mathbb{D}_A$ such that $\mathcal{P}_B$ does not hold true and no guarded command of $\mathbb{B}$ is executed in the states following σ_κ. There is at least one process P_x with an active guarded command $\mathcal{G}_{Bx_i}$ such that $\delta_B < 0$ in every state of ε. Let $\rho_\kappa = \{P_{\kappa_1}, \cdots, P_{\kappa_\theta}\}$ be the set of processes which have an enabled guarded command $\mathcal{G}_{B\kappa_t}$ such that $\delta_B < 0$ in global state σ_κ. $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ is maximal and algorithm $\mathbb{A}$ has an enabled guarded command in every state therefore ε has an infinite suffix in $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$. The suffix of ε is infinite and does not contain any execution step belonging to the processes in ρ_κ. Execution of guarded commands of algorithm $\mathbb{A}$ in processes belonging to $\varPi \backslash \rho_\kappa$ may change the truth values of individual guards of $\mathbb{A}$ in processes in ρ_κ. Because algorithm $\mathbb{A}$ has an enabled guarded command in every process and in every state, the execution of the guarded commands in processes belonging to $\varPi \backslash \rho_\kappa$ cannot change the enabledness of the processes in ρ_κ. The maximal execution $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ does not contain any execution steps belonging to processes in ρ_κ after state σ_κ although each process in ρ_κ is enabled in all the subsequent states. However, scheduler $\mathbb{D}_A$ is a weakly-fair scheduler and ensures that in any execution, a continuously enabled process is activated infinitely often. Therefore, such a sequence of states cannot be a maximal execution of algorithm $\mathbb{A}\triangle\mathbb{B}$ under scheduler $\mathbb{D}_A$. We now consider maximal states where states satisfy predicate $\mathcal{P}_B$.
Let $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ be a maximal execution of a composed algorithm $\mathbb{A}\triangle\mathbb{B}$. Let $\varepsilon_{\mathcal{P}_B}$ be the prefix of $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ such that all the states in $\varepsilon_{\mathcal{P}_B}$ satisfy predicate $\mathcal{P}_B$ and no guarded command of algorithm $\mathbb{B}$ is executed in all subsequent states. Algorithm $\mathbb{B}$ can exhibit two kinds of behavior in the states belonging to $\varepsilon_{\mathcal{P}_B}$:

Case 1: If algorithm $\mathbb{B}$ has none of its guarded command enabled in states satisfying predicate $\mathcal{P}_B$, then the projection of $\hat{\Xi}_{\mathbb{A}\triangle\mathbb{B}}$ on algorithm $\mathbb{B}$ is maximal as none of the guarded commands is enabled in the states belonging to $\varepsilon_{\mathcal{P}_B}$.

Case 2: If algorithm $\mathbb{B}$ has enabled guarded commands in the states satisfying predicate $\mathcal{P}_B$ then, as described above, the suffix of $\varepsilon_{\mathcal{P}_B}$ is constituted of states where at least one process P_x has an enabled guard $\mathcal{G}_{Bx_\iota}$ of algorithm $\mathbb{B}$. Process P_x is never activated in the suffix of $\varepsilon_{\mathcal{P}_B}$. However, this is not possible because scheduler $\mathbb{D}_A$ is weakly-fair thereby invoking every continuously enabled process infinitely often.

The exclusivity of actions of algorithm $\mathbb{B}$ ensures that if a process is activated in a state, then the transition would be between the same pair of states as that of algorithm $\mathbb{B}$ prior to composition. Recall the premise that, in all maximal executions of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$, the value of $\varDelta_B$ decreases as long as predicate $\mathcal{P}_B$ does not hold; the lemma follows. □

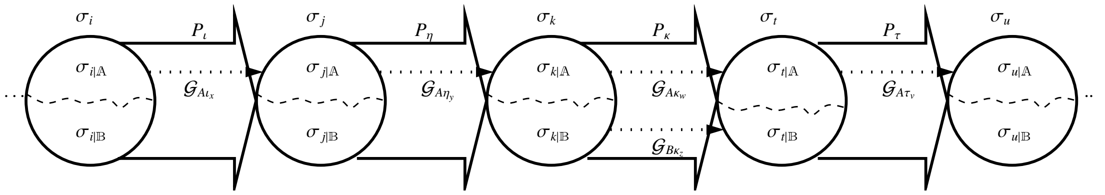

Fig. 4.2: Filtering of Execution Steps of Algorithm $\mathbb{B}$ in $\mathbb{A}^{\vartriangle}\mathbb{B}$

Lemma 4.3 shows that the guards of $\mathbb{A}^{\vartriangle}\mathbb{B}$ do not hinder algorithm $\mathbb{B}$ from taking an action. Indeed, a guarded command of algorithm $\mathbb{B}$ is executed only if it ensures that ranking function Δ_B decreases. Thus, lifting composition "shields" guards of algorithm $\mathbb{B}$ from scheduler $\mathbb{D}_A$. Although scheduler $\mathbb{D}_A$ can postpone the execution of actions of algorithm $\mathbb{B}$, weak fairness and the conjunction of guards in algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ guarantees that actions of algorithm $\mathbb{B}$ cannot be delayed indefinitely. An equivalent maximal execution of algorithm $\mathbb{B}$ can be, thus, constructed by "filtering" the projection of maximal execution of $\mathbb{A}^{\vartriangle}\mathbb{B}$ on $\mathbb{B}$. For example, consider the fragment of maximal execution of $\mathbb{A}^{\vartriangle}\mathbb{B}$ shown in Figure 4.2. Process P_ι is selected by scheduler $\mathbb{D}_A$ in state σ_i and guard $\mathcal{G}_{A\iota_x}$ pertaining to algorithm $\mathbb{A}$ is executed resulting in new state σ_j. Similarly, process P_η is selected in σ_j and as a result of execution of guard $\mathcal{G}_{A\eta_y}$ the system reaches state σ_k. Note that projections of states σ_i, σ_j and σ_k on algorithm $\mathbb{B}$ are equal as no guarded command of $\mathbb{B}$ is executed between the three states. Eventually process P_κ is selected by scheduler $\mathbb{D}_A$ in state σ_k and guarded commands belonging to both component algorithms are executed. The projection of the fragment of execution on algorithm $\mathbb{B}$ is $\langle\sigma_{i|\mathbb{B}}, \sigma_{j|\mathbb{B}}, \sigma_{k|\mathbb{B}}, \sigma_{t|\mathbb{B}}, \sigma_{u|\mathbb{B}}\rangle$. The fragment of equivalent maximal execution of $\mathbb{B}$ under $\mathbb{D}_B$ is $\langle\sigma_{i|\mathbb{B}}, \sigma_{t|\mathbb{B}}\rangle$; it is obtained by removing those projected states which are equal.

We give the definitions of an execution fragment and a round before showing that algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ preserves the convergence of algorithm$\mathbb{B}$ under scheduler $\mathbb{D}_B$ as well.

Definition 4.6 (Execution Fragment). *Let $\Xi = \langle\sigma_1, \sigma_2, \cdots, \sigma_i, \cdots, \sigma_j, \cdots\rangle$ be an execution of a distributed algorithm. A fragment of execution Ξ ε_{ij} is a subsequence of Ξ that starts in state σ_i and ends in state σ_j.*

Definition 4.7 (Execution Round). *A round in an execution is the shortest fragment ε such that each process executes an execution step in ε.*

For example, consider an execution $\Xi = \langle\sigma_1 \underset{P_1}{\longrightarrow} \sigma_2 \underset{P_2}{\longrightarrow} \sigma_3 \underset{P_1}{\longrightarrow} \sigma_4 \underset{P_3}{\longrightarrow} \sigma_5 \underset{P_2}{\longrightarrow} \sigma_6 \underset{P_3}{\longrightarrow} \cdots\rangle$ of a distributed system with three constituent processes. The first execution round of this execution is $\varepsilon = \langle\sigma_1 \underset{P_1}{\longrightarrow} \sigma_2 \underset{P_2}{\longrightarrow} \sigma_3 \underset{P_1}{\longrightarrow} \sigma_4 \underset{P_3}{\longrightarrow} \sigma_5\rangle$.

Theorem 4.2. *If all maximal executions of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$ converge to the predicate $\mathcal{P}_B$, then the projection of any maximal execution of the composed algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ on algorithm $\mathbb{B}$ under a weakly-fair scheduler $\mathbb{D}_A$ converges to predicate $\mathcal{P}_B$.*

Proof. The projection of a maximal execution of the composed algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ on algorithm $\mathbb{B}$ is a maximal execution of $\mathbb{B}$ under $\mathbb{D}_B$ (from Lemma 4.3). Thus, algorithm $\mathbb{B}$ takes a step in a state that does not satisfy predicate $\mathcal{P}_B$ only if a decrease in the value of ranking function Δ_B is guaranteed. The progress of algorithm $\mathbb{B}$ towards a state satisfying predicate $\mathcal{P}_B$ can only be hindered by not enabling a process which has a guard active such that $\delta_B < 0$. However, this goes contrary to the assumption that scheduler $\mathbb{D}_A$ is weakly-fair.

Algorithm $\mathbb{B}$ gets a chance to execute *at least* one of its enabled guarded command in every round else the weak fairness constraint would be violated. The projection of algorithm $\mathbb{A}^{\vartriangle}\mathbb{B}$ over algorithm

$\mathbb{B}$ reaches a state satisfying predicate $\mathcal{P}_B$ in finite number of steps, because Δ_B is monotonous and is defined over a well-founded domain. Well-foundedness of the domain of ranking function Δ_B implies that an infinite sequence consisting of the elements of the domain does not exist, and any non-empty subset of elements has a minimal element. Hence, algorithm $\mathbb{B}$ reaches a state satisfying predicate $\mathcal{P}_B$ in not more than $O(n) \cdot C(n)$ rounds where n is the number of processes in the system and $C(n)$ is the number of rounds required by $\mathbb{B}$ to converge to $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. □

Theorem 4.2 shows that the presence of ranking function Δ_B of algorithm $\mathbb{B}$ in guards of algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ prevents scheduler $\mathbb{D}_A$ from destroying progress of algorithm $\mathbb{B}$ towards predicate $\mathcal{P}_B$. The effect of lifting composition on the state spaces of algorithms $\mathbb{A}{\vartriangle}\mathbb{B}$ and $\mathbb{B}$ is depicted informally in Figure 4.3. Multiple states of the composed algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ have projection on a state of algorithm $\mathbb{B}$. Note that although the projection relation between states of $\mathbb{A}{\vartriangle}\mathbb{B}$ and $\mathbb{B}$ is surjective but not bijective, and the projection relation between executions of the two algorithms under scheduler $\mathbb{D}_A$ is *neither surjective nor injective*. There can be multiple executions of the composed algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ corresponding to a single execution of algorithm $\mathbb{B}$. Consider executions of algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ starting in states $\sigma_{i,1}$, $\sigma_{i,2}$ and $\sigma_{i,3}$. The projection of all the three states is σ_i in the state space of algorithm $\mathbb{B}$. Consequently, the projection of the execution $\langle\ \sigma_{i,2} \ \rightarrow\ \sigma_{j,2} \ \rightarrow\ \sigma_{n,2} \ \cdots\ \rangle$ is $\langle\ \sigma_i \ \rightarrow\ \sigma_j \ \rightarrow\ \sigma_n \ \cdots\ \rangle$. However, not all executions of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_A$ have corresponding executions in the state space of the composed algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$. As convergence of algorithm $\mathbb{B}$ as such is guaranteed only under scheduler $\mathbb{D}_B$, a different scheduler may lead to an execution which never reaches a state satisfying predicate $\mathcal{P}_B$. For example, consider the state space of algorithm $\mathbb{B}$ as shown in Figure 4.3. Scheduler $\mathbb{D}_A$ can produce an execution $\langle\ \sigma_i \ \rightarrow\ \sigma_j \ \rightarrow\ \sigma_n \ \rightarrow\ \sigma_u \ \rightarrow\ \sigma_j \ \cdots\ \rangle$ and prevent algorithm $\mathbb{B}$ from reaching a state that satisfies predicate $\mathcal{P}_B$ without violating the fairness constraint. However, such loops are not possible in the state space of composed algorithm, because the embedding of ranking function of algorithm $\mathbb{B}$ disables such counterproductive transitions; therefore, some of the executions have no corresponding execution in the state space of $\mathbb{A}{\vartriangle}\mathbb{B}$. Consider an execution $\langle\ \cdots\ \sigma_i \ \rightarrow\ \sigma_j \ \cdots\ \rangle$ of $\mathbb{B}$ under scheduler $\mathbb{D}_B$. Let process P_x execute an enabled guarded command in σ_i to reach state σ_j, and let processes P_v and P_w be enabled in σ_j. Assume that in all strategies of scheduler $\mathbb{D}_A$ process P_v is selected immediately after process P_x. In such a case, algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ will never have an execution where process P_w executes its guarded command in σ_j. Therefore, although every projected execution of algorithm $\mathbb{B}$ in $\mathbb{A}{\vartriangle}\mathbb{B}$ under scheduler $\mathbb{D}_A$ has an equivalent execution under scheduler $\mathbb{D}_B$, not all executions of $\mathbb{B}$ under scheduler $\mathbb{D}_B$ may have corresponding executions in the state space of algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$.

Theorem 4.3. *Algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under a weakly-fair scheduler $\mathbb{D}_A$.*

Proof. Convergence of $\mathbb{A}{\vartriangle}\mathbb{B}$ follows from Theorems 4.1 and 4.2. Closure follows from Lemmata 4.2 and 4.3 and the assumption that predicate $\mathcal{P}_B$ is closed under scheduler $\mathbb{D}_A$. □

4.4 The Role of Schedulers in Lifting Composition

We have shown that the result of lifting composition is a self-stabilizing algorithm as well. We now discuss the impact of the relationship between the schedulers of component algorithms $\mathbb{A}$ and $\mathbb{B}$ on the schedulers which preserve the self-stabilization properties in the composed algorithm $\mathbb{A}{\vartriangle}\mathbb{B}$.

Corollary 4.2. *Lifting composition preserves the self-stabilization property of algorithm $\mathbb{B}$ under schedulers for which algorithm $\mathbb{A}$ is self-stabilizing with respect to $\mathcal{P}_A$ provided $\mathcal{P}_B$ is closed under the scheduler of algorithm $\mathbb{A}$.*

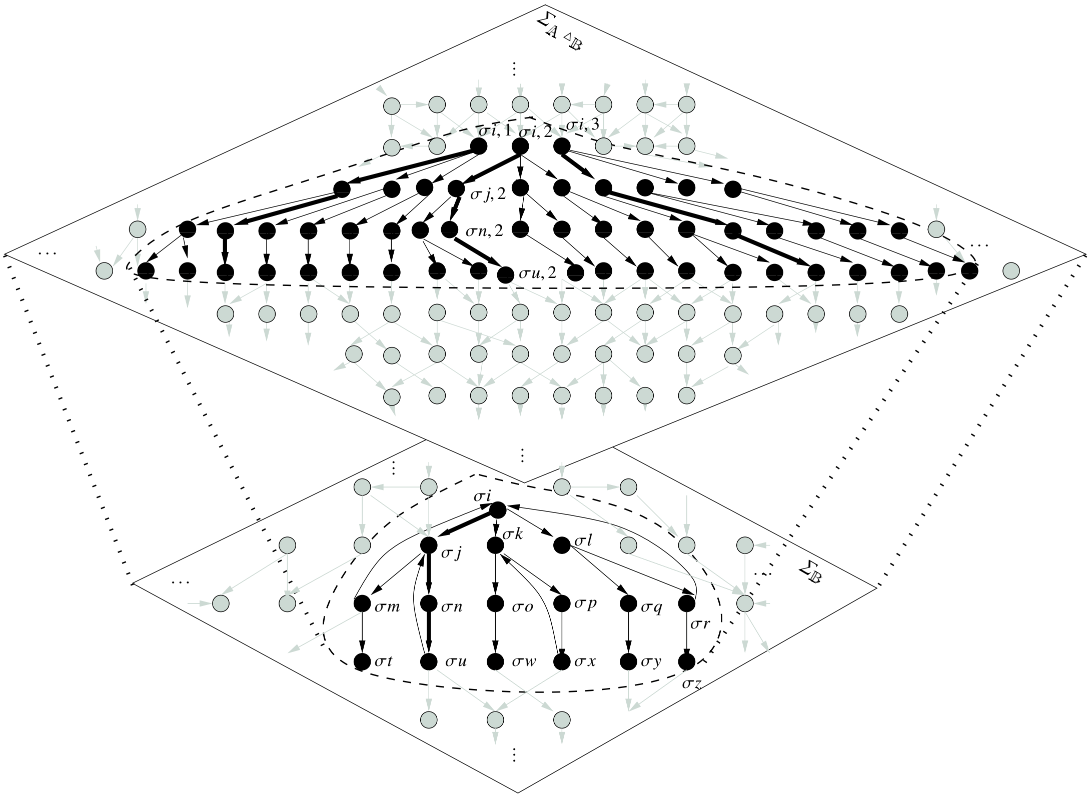

Fig. 4.3: Projection of Algorithm $\mathbb{A}\triangle\mathbb{B}$ over Algorithm $\mathbb{B}$

The critical aspect of lifting composition is exploiting the knowledge of the ranking function of algorithm $\mathbb{B}$. Ranking function of algorithm $\mathbb{B}$ encodes the information about the progress that $\mathbb{B}$ makes towards the set of states represented by predicate $\mathcal{P}_B$. Essentially, a transition of algorithm $\mathbb{B}$ in the composed algorithm $\mathbb{A}\triangle\mathbb{B}$ is enabled *only* when it is guaranteed that it will lead to progress. A scheduler not admitted by algorithm $\mathbb{B}$ can invoke a process such that execution of its enabled guarded command is counterproductive to the convergence of $\mathbb{B}$, and thereby, might result in an execution which does not have states satisfying predicate $\mathcal{P}_B$. However, the guards of algorithm $\mathbb{B}$ are "protected" by the ranking function Δ_B. Thus, the ranking function Δ_B of algorithm $\mathbb{B}$ act as a "filter" of actions of $\mathbb{B}$ in the composed $\mathbb{A}\triangle\mathbb{B}$. In any state an assignment statement of algorithm $\mathbb{B}$ is executed in algorithm $\mathbb{A}\triangle\mathbb{B}$ only if a favorable process is selected by scheduler $\mathbb{D}_A$ of algorithm $\mathbb{A}$. Thus, for an external observer algorithm $\mathbb{B}$ is self-stabilizing despite the fact that the underlying scheduler is $\mathbb{D}_A$ instead of $\mathbb{D}_B$. The transfer of the self-stabilizing property of under scheduler $\mathbb{D}_B$, however, may come at the cost of increased convergence time. For instance, assume execution of an action act_{Bi} of algorithm $\mathbb{B}$ in process P_x after invocation of process P_y is necessary to ensure progress towards predicate $\mathcal{P}_B$. Let scheduler $\mathbb{D}_A$ be the scheduler of algorithm $\mathbb{A}$ which does not ensure this requirement. Therefore, algorithm $\mathbb{B}$ is not guaranteed to make progress under scheduler $\mathbb{D}_A$. However, the presence of ranking function Δ_B "shields" algorithm $\mathbb{B}$ from counter-productive process invocation. As explained above the execution of the assignment statements of algorithm $\mathbb{B}$ has to be postponed at times and thus, takes longer than it would have under scheduler $\mathbb{D}_B$.

Compositional laws for self-stabilizing algorithms usually require that both components $\mathbb{C}_1$ and $\mathbb{C}_2$ show self-stabilizing behavior under schedulers admitted by each of the components. In essence, schedulers admitted by such compositional schemes are those strategies ϱ_i which are members of both schedulers $\mathbb{D}_{C1}$ *and* $\mathbb{D}_{C2}$. An important consequence of this property is that it is necessary that there exists a set of strategies $\aleph := \{\varrho_i \mid \varrho_i \in \mathbb{D}_{C1} \wedge \varrho_i \in \mathbb{D}_{C2}\}$ common to $\mathbb{D}_{C1}$ and $\mathbb{D}_{C2}$, that is, $\mathbb{D}_{C1} \cap \mathbb{D}_{C2} \neq \emptyset$. However, lifting composition does not have any such restriction. Lifting compositon "lifts" the self-stabilization property of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$ to scheduler $\mathbb{D}_A$ without interfering with the properties of algorithm $\mathbb{A}$.

Theorem 4.4. *Algorithm* $\mathbb{A}^{\triangle}\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_A \wedge \mathcal{P}_B$ *under scheduler* $\mathbb{D}_A$ *even if* $\mathbb{D}_A \cap \mathbb{D}_B = \emptyset$.

Proof. Closure and convergence of algorithm $\mathbb{A}$ in $\mathbb{A}^{\triangle}\mathbb{B}$ follows from Theorem 4.1.
Scheduler $\mathbb{D}_A$ has no strategy under which algorithm $\mathbb{B}$ converges to its safety predicate $\mathcal{P}_B$. Nevertheless, each strategy of scheduler $\mathbb{D}_A$ must select each process infinitely often in any maximal execution because scheduler $\mathbb{D}_A$ is weakly fair. The theorem follows from the observation that guards of algorithm $\mathbb{B}$ in the composed algorithm $\mathbb{A}^{\triangle}\mathbb{B}$ are enabled only if it leads to decrease in ranking function Δ_B, and the assumption that algorithm $\mathbb{A}$ has exactly one enabled guarded command per process in every state. □

The transfer of the self-stabilizing property of $\mathbb{B}$ via lifting composition leads to an interesting result in case scheduler $\mathbb{D}_A$ contains scheduler $\mathbb{D}_B$. This means that all the strategies of $\mathbb{D}_B$ are members of $\mathbb{D}_A$ and vice versa is not true, that is $\mathbb{D}_A \cap \mathbb{D}_B = \mathbb{D}_B$. In such a scenario, lifting composition elevates the self-stabilizing property of scheduler $\mathbb{B}$ to a class of schedulers larger than its own. The algorithm $\mathbb{D}_B$ exhibits the self-stabilizing behavior – in addition to strategies of scheduler $\mathbb{D}_B$ – also under process invocation sequences of $\mathbb{D}_A$.

Theorem 4.5. *Lifting composition* $\mathbb{A}^{\triangle}\mathbb{B}$ *enlarges the set of strategies under which* $\mathbb{B}$ *self-stabilizes to predicate* $\mathcal{P}_B$.

Proof. If a strategy belongs to both schedulers $\mathbb{D}_A$ and $\mathbb{D}_B$, then the convergence of algorithm $\mathbb{B}$ in composed algorithm $\mathbb{A}^{\triangle}\mathbb{B}$ follows directly.
If a strategy belongs exclusively to scheduler $\mathbb{D}_A$, then the convergence of algorithm $\mathbb{B}$ follows from Theorem 4.4, since in all such strategies, $\mathbb{D}_A$ must activate each continuously enabled process infinitely often. Thus, if all strategies of scheduler $\mathbb{D}_B$ also belong to $\mathbb{D}_A$ and *vice versa* is not true, then lifting composition enlarges the set of strategies under which $\mathbb{B}$ is self-stabilizing. □

The above property can also be used for iterative design of self-stabilizing algorithms. Large self-stabilizing algorithms can constructed if (1) there is an order defined over the schedulers of the components, that is $\mathbb{D}_i \subset \mathbb{D}_j \subset \mathbb{D}_k$, and (2) the each components has a ranking function under its respective scheduler.

Calibrating Lifting Composition

The property of transcending schedulers distinguishes lifting composition from other compositional techniques proposed for self-stabilizing algorithms. As discussed in Chapter 3 most of the compositional techniques inherently assume that component algorithms are self-stabilizing under the *same* scheduler. Beauquier *et al.* [87] first addressed the problem of transferring self-stabilization across the schedulers of the components and presented, what the authors termed as, "cross-over composition" as a solution. However, cross-over composition heavily relies on the "strong" component algorithm to transcend the schedulers. We recall two critical properties of cross-over composition (as stated in [87]) in order to calibrate the result of lifting composition.

Theorem 4.6 (Self-Stabilization Preservation in Cross-Over Composition [87]). *Let* $\mathbb{A}\diamond\mathbb{B}$ *be the cross-over composition between algorithm* $\mathbb{A}$ *and a* fair *algorithm* $\mathbb{B}$. *If algorithm* $\mathbb{A}$ *is self-stabilizing with respect to* $\mathcal{P}_{SP}$ *under scheduler* $\mathbb{D}$ *and algorithm* $\mathbb{B}$ *is self-stabilizing with respect to* $\mathcal{P}_{SR}$ *and* fair *under* $\mathbb{D}$ *then* $\mathbb{A}\diamond\mathbb{B}$ *is self-stabilizing with respect to* $\mathcal{P}_{SP}\wedge\mathcal{P}_{SR}$ *under* $\mathbb{D}$.

Guards in cross-over composition are formed by simple conjunction of guards belonging to the component algorithms and do not contain any extra condition; guards of "strong" algorithm $\mathbb{B}$ are allowed to execute if none of the guards of algorithm $\mathbb{A}$ are true in a process. Thus, cross-over composition requires algorithm $\mathbb{B}$ to be fair lest some actions of algorithm $\mathbb{A}$ may never be executed. Additionally, both component algorithms must be self-stabilizing under same scheduler otherwise scheduler $\mathbb{D}$ can hinder the convergence of algorithm $\mathbb{A}$. For instance, if algorithm $\mathbb{A}$ is self-stabilizing only under a round-robin scheduler –which selects every processor once in every execution round– and algorithm $\mathbb{B}$ is self-stabilizing under any weakly fair scheduler, then self-stabilization of algorithm $\mathbb{B}$ in the composed algorithm $\mathbb{A}\diamond\mathbb{B}$ cannot be guaranteed. In such cases, cross-over composition requires algorithm $\mathbb{B}$ to have stronger properties such as exemplified by the following theorem.

Theorem 4.7 (Self-stabilization from *k*-fairness to *k*-boundedness [87]). *If algorithm* $\mathbb{A}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{SP}$ *under a k- bounded scheduler and algorithm* $\mathbb{B}$ *is a k-*fair *self-stabilizing algorithm with respect to predicate* $\mathcal{P}_{SR}$, *then cross-over composition* $\mathbb{A}\diamond\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{SP}\wedge\mathcal{P}_{SR}$ *under any unfair scheduler.*

A process can execute at most *k* steps between any two execution steps of another process in any execution of a *k*-fair algorithm; a *k*-bounded scheduler ensures that till an enabled process is selected to execute a step, any other process can execute at most *k* execution steps. Essentially, the transfer of the self-stabilization property of algorithm$\mathbb{A}$ hinges on the ability of algorithm $\mathbb{B}$ to produce *k*-fair executions under an unfair scheduler and the assertion that a *k*-fair execution is *k*-bounded as well. Thus, the task of filtering the execution is passed on to the component algorithm. In contrast to cross-over composition, lifting composition uses the ranking function of a component algorithm to produce conforming executions. Thus, in case the self-stabilizing property of an algorithm needs to be transferred to another scheduler via composition, other component algorithm is not required to have strong properties thereby making the proof obligations for showing correctness of the composed algorithm simpler.

Lifting Composition and Weak Convergence

The ability of lifting composition to enlarge the set of strategies (Theorem 4.5) can be used to compose weakly stabilizing algorithms with self-stabilizing algorithms. Recall that, unlike a self-stabilizing algorithm, a weakly-stabilizing algorithm exhibits weak convergence; every state in weakly-stabilizing system has an execution that reaches a state satisfying the safety predicate. Because self-stabilization is strictly stronger than weak-stabilization, proof obligations for showing correctness of weakly-stabilizing systems are relatively less strict. Weak convergence is proven by showing the existence of a weaker form of ranking function as defined below.

Definition 4.8 (Weak Ranking Function Δ_{weak} **[53]).** *A weak ranking function* Δ_w *maps the state space* Σ *of a distributed algorithm to a well-founded set* Θ *such that for every state* $\sigma_i \in \Sigma$ *there exists an execution step* $\sigma_i \rightarrow \sigma_j$ *where* $\Delta_w(\sigma_i) > \Delta_w(\sigma_j)$ *holds.*

A weak ranking function, unlike a ranking function, is required to decrease for *at least one* execution step emanating in any state. Alternatively, in every state of a weakly-stabilizing algorithm, the execution of at least one enabled guarded command is "beneficial for the progress towards the safety predicate". Thus, if one can define a scheduler that is constrained to select only such beneficial transition, then weakly-stabilizing algorithm can be transformed into a self-stabilizing algorithm. A "favorable" notion of strong fairness is defined by Gouda [53]; subsequently it is been shown that, under such a "strong scheduler," a weakly-stabilizing algorithm turns into a self-stabilizing algorithm.

Definition 4.9 (Gouda Strong Fairness [53]). *An execution Ξ is strongly fair iff for every state transition $\sigma_i \rightarrow \sigma_j$, if state σ_i appears infinitely often in Ξ, then transition $\sigma_i \rightarrow \sigma_j$ appears infinitely often in Ξ.*

Gouda's notion of strong fairness stipulates that if a transition is *possible* infinitely often then *it is taken* infinitely often as well. A weakly-stabilizing algorithm exhibits convergence under Gouda strong fairness as stated by following theorem.

Theorem 4.8 (From Weak Stabilization to Self-Stabilization [53]). *If algorithm $\mathbb{W}$ has a finite number of states and $\mathbb{W}$ is weakly stabilizing with respect to predicate $\mathcal{P}_W$, then $\mathbb{W}$ is self-stabilizing with respect to $\mathcal{P}_W$ under Gouda Strong fairness.*

A Gouda scheduler is constrained to take every transition emanating from a state that appears infinitely often in any execution; the theorem follows from the premise that every state has at least one execution that has states satisfying predicate $\mathcal{P}_W$ in its suffix. Essentially, a Gouda scheduler encapsulates best-case behavior of a weakly-stabilizing algorithm.

While it is known that two weakly-stabilizing algorithms can be composed to produce another weakly-stabilizing algorithm [53], composition of a weakly-stabilizing algorithm with a self-stabilizing algorithm does not lead to transfer of weak convergence under the scheduler of the self-stabilizing component. However, lifting composition can be used to transform a weakly-stabilizing algorithm into a self-stabilizing algorithm.

The proof of weak-stabilization includes a weak ranking function which decreases for at least one transition in each state [53]. Recall that a scheduler is said to be stronger than another scheduler if it is subjected to weaker constraints while selecting a strategy. Moreover –as described earlier– a weakly-stabilizing algorithm is a self-stabilizing algorithm under a Gouda strongly fair scheduler. However, a Gouda strongly fair scheduler $\mathbb{D}_W$ is *strictly weaker* than the classical strongly-fair scheduler [88]. This implies that all the strategies of the scheduler $\mathbb{D}_W$ also belong to a strongly-fair scheduler $\mathbb{D}_S$ and *vice versa* is not true, that is, $\mathbb{D}_W \subset \mathbb{D}_S$. It is also known that a classical strongly-fair scheduler is weaker than a weakly-fair scheduler $\mathbb{D}_{WF}$ [41]. Thus, a Gouda strongly fair scheduler is strictly weaker than a weakly fair-scheduler as well, that is, $\mathbb{D}_W \subset \mathbb{D}_{WF}$. The fact that there is a ranking function which encapsulates the notion of progress for a weakly-stabilizing algorithm can be exploited by lifting composition. Corollaries 4.4 and 4.3 follow from Theorem 4.5 by instantiating $\mathbb{D}_B$ with $\mathbb{D}_W$, and $\mathbb{D}_A$ with $\mathbb{D}_S$ and $\mathbb{D}_{WF}$ respectively. In the resultant composed algorithm $\mathbb{S}^{\vartriangle}\mathbb{W}$, the weakly-stabilizing algorithm $\mathbb{W}$ is self-stabilizing under the scheduler $\mathbb{D}_S$ or $\mathbb{D}_{WF}$, respectively, of the self-stabilizing algorithm $\mathbb{S}$.

Corollary 4.3. *The lifting composition $\mathbb{S}^{\vartriangle}\mathbb{W}$ of a weakly-stabilizing algorithm $\mathbb{W}$ with a self-stabilizing algorithm $\mathbb{S}$ which is self-stabilizing under a strongly-fair scheduler leads to self-stabilization of weakly-stabilizing algorithm $\mathbb{W}$ under the strongly-fair scheduler of self-stabilizing algorithm $\mathbb{S}$.*

Corollary 4.4. *The lifting composition $\mathbb{S}^{\vartriangle}\mathbb{W}$ of a weakly-stabilizing algorithm $\mathbb{W}$ with an algorithm $\mathbb{S}$ which is self-stabilizing under a weakly-fair scheduler leads to self- stabilization of weakly-stabilizing algorithm $\mathbb{W}$ under the weakly-fair scheduler of self- stabilizing algorithm $\mathbb{S}$.*

4.5 Examples

We now apply lifting composition on two pairs of self-stabilizing algorithms although the component algorithms have different schedulers. Additionally, it is shown that the resultant algorithms are self-stabilizing despite the "incompatibility" of component schedulers. Before we illustrate the use of lifting composition, the component algorithms are briefly analyzed to aid comprehension.

Self-Stabilizing Maximum Algorithm

The self-stabilizing maximum algorithm $\mathbb{SSMax}$ computes the maximum value among the local states of the processes which run the algorithm. The local state of each process P_i in $\mathbb{SSMax}$ is represented by an integer x_i. Processes communicate via shared memory registers. Each process can communicate with every other process in the system, therefore, the communication topology is represented by a fully connected graph. Figure 4.4 shows the sub-algorithm run by each process P_i. The macro **max** calculates the maximum value amongst the local states of the neighbors of process P_i.

```
process $P_i$
{
    var int  $x_i$; \ * local state * \
    $\mathcal{G}_{A1}$ :: $[]\mathbf{max}(x_j|\forall x_j \in \Pi \backslash \{P_i\}) > x_i$    $\rightarrow$    $x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \backslash \{P_i\})$;
    $\mathcal{G}_{A2}$ :: $[]\mathbf{max}(x_j|\forall x_j \in \Pi \backslash \{P_i\}) \leq x_i$    $\rightarrow$    $x_i := x_i$;
}
```

Fig. 4.4: Sub-Algorithm $\mathcal{SSMax}_i$

Each process P_i continuously compares its local state with the result of the macro **max**; if x_i is smaller than result of **max**, then P_i assigns it to x_i (Guarded Command $\mathcal{G}_{A1}$). Let $\sigma_f = \langle x_1^f, x_2^f, \cdots, x_n^f \rangle$ be some global state of the system and $\mathcal{P}_{SMax}$ be a predicate defined over the global states of algorithm $\mathbb{SSMax}$; any global state σ_e belonging to an execution of $\mathbb{SSMax}$ starting in state σ_f satisfies predicate $\mathcal{P}_{SMax}$ if the local states of all processes in σ_e are equal to the maximum value of the local states in σ_f, that is, $\forall_{i \in \{1, \cdots, n\}} : x_i^e = \mathbf{max}(x_1^f, \cdots, x_n^f)$.

The following theorem encapsulates the correctness of algorithm $\mathbb{SSMax}$.

Theorem 4.9. *Algorithm* $\mathbb{SSMax}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{SMax}$ *under any weakly-fair scheduler* $\mathbb{D}_W$.

Proof Sketch. Note that each process has exactly one enabled guarded command in every global state. Whenever a process executes a guarded command, it updates its local state to the maximum value in the system and if it has the maximum value, then it does not modify its local state. Consider a subset π_{arbit} of Π such that none of the processes in π_{arbit} has maximum value. Further assume that the processes in $\Pi \backslash \pi_{arbit}$ have the maximum value. A scheduler can delay the convergence of $\mathbb{SSMax}$ towards the states satisfying $\mathcal{P}_{SMax}$ only by not selecting a process belonging to π_{arbit}. However, such a scheduler would violate weak fairness as all the processes are continuously enabled. The theorem follows from the inference that a weakly fair scheduler must select each process infinitely often.

Self-Stabilizing Bi-Stabilizer Algorithm

The self-stabilizing bi-stabilizer algorithm $\mathbb{SSBiSt}$ is defined over a set of processes Π which has two classes of processes, *ODD* and *EVEN*. Process P_i belongs to class *ODD* if its identifier i is an odd integer; P_i belongs to class *EVEN* if i is an even integer. The local state of each process is represented by a Boolean variable y_i. Processes communicate with each other using shared memory registers and communication graph of the system is a fully connected graph.

Figure 4.5 shows the sub-algorithm $\mathcal{SSBiST}_{i_o}$ run by the processes in class *ODD* while the processes belonging to class *EVEN* implement the sub-algorithm $\mathcal{SSBiST}_{i_e}$ is shown in Figure 4.6. Every *ODD* process P_{i_o} repeatedly reads the local states of its neighbors, and if the local states of all of its neighbors as well as its own local state are *true*, then it assigns *true* (represented by 1) to x_{i_o}, else y_{i_o} is assigned *false* (represented by 0). An *EVEN* process P_{i_e} assigns its local state *false* if its local states and the local

```
process $P_i$  \ * ODD * \
{
  var bool  $y_i$;
  $\mathcal{G}_{B1}$ :: [] $\bigwedge_{j\in\{1,\cdots,n\}} y_j \rightarrow y_i := 1$;
  $\mathcal{G}_{B2}$ :: []$\neg \bigwedge_{j\in\{1,\cdots,n\}} y_j \rightarrow y_i := 0$;
}
```

Fig. 4.5: Sub-Algorithm $\mathcal{SSBiSt}_{i_o}$

```
process $P_i$  \ * EVEN * \
{
  var bool  $y_i$;
  $\mathcal{G}_{B1}$ :: [] $\bigwedge_{j\in\{1,\cdots,n\}} y_j \rightarrow y_i := 0$;
  $\mathcal{G}_{B2}$ :: []$\neg \bigwedge_{j\in\{1,\cdots,n\}} y_j \rightarrow y_i := 1$;
}
```

Fig. 4.6: Sub-Algorithm $\mathcal{SSBiSt}_{i_e}$

states of all of its neighbors are *true*, else it assigns *true* to y_{i_e}. Algorithm $\mathbb{SSBiSt}$ leads the system to a "bivalent configuration," that is, the local states of all the *ODD* processes are **false** and the local states of all the *EVEN* processes are **true**.

Let $\mathcal{P}_{BiSt} \equiv (\bigwedge_{i=0}^{i\leq\lfloor n-1/2\rfloor}(y_{2\cdot i+1} = false)) \wedge (\bigwedge_{j=1}^{j\leq\lfloor n/2\rfloor}(y_{2\cdot j} = true))$ be predicate which is true in a state where all *ODD* processes have local states equal to *false* and all *EVEN* processes have local states equal to *true*. Reachability of states satisfying predicate $\mathcal{P}_{BiSt}$ is established by the following theorem.

Theorem 4.10. *Algorithm* $\mathbb{SSBiSt}$ *has at least one execution that reaches a state satisfying predicate* $\mathcal{P}_{BiSt}$

Proof Sketch. Consider a binary string s of size n representing the global state of a system implementing algorithm $\mathbb{SSBiSt}$. An *ODD* process sets its local state to *false* in a state where s has at least one element with value *false*. Likewise, an *EVEN* process P_{i_e} assigns y_{i_e} the truth value *true* if s has at least one element with value *false*. Note that if s has at least one odd element with value *false* then the value of this element does not change in any following state. If s has an odd element with value *true* and there exists at least one element with value *false* then the action of the *ODD* element with local state *true* will switch it to *false*. An action of an *EVEN* process in a state where all the elements of s are *true* leads to an assignment of *false* to the local state of the *EVEN* process. The theorem follows from the observation that a local state of an *ODD* process remains unmodified once it has truth value *false*.

Theorem 4.10 implies that algorithm $\mathbb{SSBiSt}$ *can* converge towards the states satisfying predicate $\mathcal{P}_{BiSt}$. However, the convergence property of $\mathbb{SSBiSt}$ is contingent on the constraints imposed on the underlying scheduler. More specifically, if the assumed scheduler is allowed to utilize "enough strategies", algorithm $\mathbb{SSBiSt}$ may never reach states satisfying predicate $\mathcal{P}_{BiSt}$.

Theorem 4.11. *Algorithm* $\mathbb{SSBiSt}$ *does not converge to states satisfying predicate* $\mathcal{P}_{BiSt}$ *under a weakly fair scheduler* $\mathbb{D}_W$.

Proof. Let ξ ($\xi \subset \Sigma$) be a subset of system states of $\mathbb{SSBiSt}$ such that each global state σ ($\sigma \in \xi$) has at least one *EVEN* process with local state *false* and all *ODD* processes with local states *true*. We construct an execution Ξ (and the respective strategy ϱ of scheduler $\mathbb{D}_W$) –starting in a state σ_i belonging to set ξ– that does not reach a state satisfying predicate $\mathcal{P}_{BiSt}$. Let $\iota = \{i_1, i_2, \cdots, i_k\}$ be the set of identifiers of *EVEN* processes whose local states are *false* in σ_i. Scheduler $\mathbb{D}_W$ selects one of the processes with identifiers in ι and, as a result, the local state of the selected process changes to *true*. In the resultant state scheduler $\mathbb{D}_W$ again selects one of the processes of set ι; $\mathbb{D}_W$ selects processes from ι till all of them have their local states equal to *true*. Let σ_{2^n-1} be the state where local states of all the processes in the system are *true*. $\mathbb{D}_W$ sequentially selects all the *ODD* process in state σ_{2^n-1} without selecting any *ODD* process twice. As all the processes in the system have local states equal to *true*, system remains in state σ_{2^n-1}. $\mathbb{D}_W$ next selects one of the *EVEN* processes and system reaches a state σ_{2^n-2} where exactly one (*EVEN*) process has *false* as local state. Let $P_{i_{ol}}$ be the *EVEN* process with local state *false* in σ_{2^n-2}. Scheduler $\mathbb{D}_W$ selects all *EVEN* processes except $P_{i_{ol}}$ in state σ_{2^n-2}; system state remains unchanged because $P_{i_{ol}}$ has *false* as local state. After all other *EVEN* processes are selected, $\mathbb{D}_W$ selects process $P_{i_{ol}}$ and, consequentially, the system again reaches state σ_{2^n-1}. Scheduler $\mathbb{D}_W$ now selects processes in the same order in which they were selected when system was previously in state σ_{2^n-1}. This cycle is repeated *ad infinitum* and as a result, the system oscillates between states σ_{2^n-1} and σ_{2^n-2}. Note that each process has exactly one enabled guarded command in every global state and, hence, Ξ is an infinite execution where every process remains enabled continuously. Scheduler $\mathbb{D}_W$ does not violate weak fairness because every process is selected infinitely often. The theorem then follows. □

While Theorem 4.11 shows that self-stabilization of algorithm $\mathbb{SSBiSt}$ is impossible under a weakly fair scheduler, Theorem 4.10 indicates that the impossibility can be circumvented by selecting a favorable scheduler. The proof of Theorem 4.11 provides an instance of additional constraint which can be put on a favorable scheduler while outlining a strategy which prevents $\mathbb{SSBiSt}$ from converging to the set of legal states. More specifically, an underlying scheduler can be restrained by precluding the selection of an *EVEN* process in any execution step if the scheduler has exclusively selected *EVEN* processes in the preceding $\lfloor n/2 \rfloor - 1$ execution steps. We now define an instance of such a restricted scheduler and then show that algorithm $\mathbb{SSBiSt}$ is self-stabilizing under this scheduler.

Definition 4.10 (Scheduler $\mathbb{D}_{BiSt}$). *Scheduler $\mathbb{D}_{BiSt}$ selects an enabled process in each step while fulfilling the following conditions:*

1) an infinitely often enabled process is selected infinitely often,
2) two EVEN or two ODD processes are never selected in consecutive execution steps,
3) an EVEN process is never selected in a state if it is the only process with local state **false**,
4) an ODD process is never selected in a state where all processes have local states equal to **true**.

Theorem 4.12. *Algorithm $\mathbb{SSBiSt}$ is self-stabilizing with respect to predicate $\mathcal{P}_{BiSt}$ under scheduler $\mathbb{D}_{BiSt}$.*

Proof. We define function

$$\Delta_{BiSt} = K_1 \cdot (\lceil n/2 \rceil - \textstyle\sum_{j \in ODD}(1 - y_j)) + K_2 \cdot (\lfloor n/2 \rfloor - \sum_{k \in EVEN} y_k)) + K_3 \cdot \prod_{\forall i \in \Pi} y_i$$

such that Δ_{BiSt} maps the global states of the system to the set of integers. K_1,K_2 and K_3 are positive integers such that $K_3 > K_2$. The minimum value of Δ_{BiSt} is 0 and it attains this value in the state where all *ODD* processes have *false* as local states and all *EVEN* processes have *true* as local states. Note that such a global state satisfies the safety predicate $\mathcal{P}_{BiSt}$. Consider a state σ_u such that it does not satisfy $\mathcal{P}_{BiSt}$ and at least one process has local state *false*. If an *ODD* process is selected in σ_u, then (1) the selected process sets its local state to *false* and, (2) the value of Δ_{BiSt} is decreased because the term $\sum_{\forall j \in ODD}(1 - y_j)$ increases by 1. The value of Δ_{BiSt} decreases even if the scheduler selects an *EVEN* process in σ_u because, in such a case the selected process sets its local state to **true** and –as a result– the term $\sum_{\forall : k \in EVEN} y_k$ increases by 1. Consider a state σ_v where all processes except a *EVEN* process have local states equal to *true*. In such a state scheduler $\mathbb{D}_{BiSt}$ cannot select the *EVEN* process with local state *false* (due to condition 3 in Definition 4.10). The execution of any guarded command in state σ_v leads to a decrease in Δ_{BiSt} as an *ODD* process would set its local state to *false* if it is selected. No *ODD* process can be selected by $\mathbb{D}_{BiSt}$ in state where all local states are *true* (due to condition 4 in Definition 4.10). Nonetheless, an action of *EVEN* process leads to a decreases in the value of Δ_{BiSt} because an *EVEN* process assigns *false* to its local state and $K_2 < K_3$. The theorem follows. □

Lifting Composition of Algorithms $\mathbb{SSMax}$ *and* $\mathbb{SSBiSt}$

Algorithm $\mathbb{SSBiSt}$ requires much stronger constraints on the scheduler to converge to the states satisfying $\mathcal{P}_{BiSt}$ than weak fairness. Thus, any compositional operation on $\mathbb{SSBiSt}$ must guarantee that the scheduling requirements are met. A trivial solution would be to compose $\mathbb{SSBiSt}$ with an algorithm that stabilizes under $\mathbb{D}_{BiSt}$ or a scheduler which is strictly weaker than $\mathbb{D}_{BiSt}$. However, if the aim is to compose $\mathbb{SSBiSt}$ with $\mathbb{SSMax}$, –an algorithm that stabilizes under a stronger scheduler $\mathbb{D}_W$– then the application of lifting composition becomes a viable solution.

Figures 4.8 and 4.7 show the composed sub-algorithms. All *ODD* processes run sub-algorithm $\mathcal{SSMax}_{i_o}{}^{\triangle}\mathcal{SSBiSt}_{i_o}$; sub-algorithm $\mathcal{SSMax}_{i_e}{}^{\triangle}\mathcal{SSBiSt}_{i_e}$ is run by *EVEN* processes. Every guard of the composed algorithm $\mathbb{SSMax}^{\triangle}\mathbb{SSBiSt}$ is obtained by the conjunction of the respective guards of algorithms $\mathbb{SSBiSt}$ and $\mathbb{SSMax}$ along with two other terms which evaluate the current state of $\mathbb{SSBiSt}$. A function Δ_{Bij} is derived from ranking function Δ_{BiSt} by replacing the variable modified by a guarded command $\mathcal{G}_{B_i}$ in process P_j with the value assigned to the variable in the assignment part of $\mathcal{G}_{B_i}$. For example, Δ_{B1i} is defined as follows

$$\Delta_{B1i} = K_1 \cdot (\lceil n/2 \rceil - \textstyle\sum_{k \in ODD - \{P_j\}} (1 - y_k)) + K_2 \cdot (\lfloor n/2 \rfloor - \sum_{l \in EVEN} y_l)) + K_3 \cdot \prod_{u \in \Pi \setminus \{P_j\}} y_u$$

for an *ODD* process and it is obtained by replacing y_j with 1 in Δ_{BiSt}; similarly, Δ_{B1i} for an *EVEN* process is

$$\Delta_{B1i} = K_1 \cdot (\lceil n/2 \rceil - \textstyle\sum_{k \in ODD} (1 - y_k)) + K_2 \cdot (\lfloor n/2 \rfloor - \sum_{l \in EVEN \setminus \{P_j\}} y_l)).$$

```
process $P_i$
{
  var bool $y_i$;
  var int $x_i$;
  macro $\mathcal{P}_{BiSt} \equiv (\bigwedge_{i=0}^{i \leq \lfloor n-1/2 \rfloor} (y_{2 \cdot i+1} = 0)) \wedge (\bigwedge_{j=1}^{j \leq \lfloor n/2 \rfloor} (y_{2 \cdot j} = 1))$
  macro $\Delta_{\mathcal{P}_{BiSt}} \equiv K_1 \cdot (\lceil n/2 \rceil - \sum_{\forall k \in ODD} (1 - y_k)) + K_2 \cdot (\lfloor n/2 \rfloor - \sum_{\forall l \in EVEN} y_l))$
                                                   $+K_3 \cdot \prod_{\forall u \in \Pi} y_u$
  $\mathcal{G}_{A^{\triangle}B1}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 1;$
                                                   $x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B2}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B2i} < \Delta_{BiSt}) \rightarrow y_i := 0;$
                                                   $x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B3}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 1; x_i := x_i;$
  $\mathcal{G}_{A^{\triangle}B4}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 0; x_i := x_i;$
  $\mathcal{G}_{A^{\triangle}B5}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 1;$
                                   $x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B6}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 0;$
                                   $x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B7}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 1; x_i := x_i;$
  $\mathcal{G}_{A^{\triangle}B8}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 0; x_i := x_i;$
  $\mathcal{G}_{A^{\triangle}B9}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B10}$ :: []$\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B2i} > \Delta_{BiSt}) \rightarrow x_i :=$ max$(x_j | \forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\triangle}B11}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i := x_i;$
  $\mathcal{G}_{A^{\triangle}B12}$ :: []$\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i := x_i;$
}
```

Fig. 4.7: Sub-algorithm $\mathcal{SSMax}_{i_o}{}^{\triangle}\mathcal{SSBiSt}_{i_o}$

```
process $P_i$
{
  var bool $y_i$;
  var int $x_i$;
  macro $\mathcal{P}_{BiSt} \equiv (\bigwedge_{i=0}^{i\leq\lfloor n-1/2\rfloor}(y_{2\cdot i+1} = 0)) \wedge (\bigwedge_{j=1}^{j\leq\lfloor n/2\rfloor}(y_{2\cdot j} = 1))$
  macro $\Delta_{BiSt} \equiv K_1 \cdot (\lceil n/2\rceil - \sum_{\forall k\in ODD}(1 - y_k)) + K_2 \cdot (\lfloor n/2\rfloor - \sum_{\forall l\in EVEN} y_l))$
                                                        $+K_3 \cdot \prod_{\forall u\in\Pi} y_u$;
  $\mathcal{G}_{A^{\vartriangle}B1} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 0;$
                                                        $x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B2} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B2i} < \Delta_{BiSt}) \rightarrow y_i := 1;$
                                                        $x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B3} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 0; x_i := x_i;$
  $\mathcal{G}_{A^{\vartriangle}B4} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} < \Delta_{BiSt}) \rightarrow y_i := 1; x_i := x_i;$
  $\mathcal{G}_{A^{\vartriangle}B5} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 0;$
                                      $x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B6} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 1;$
                                      $x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B7} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 0; x_i := x_i;$
  $\mathcal{G}_{A^{\vartriangle}B8} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \mathcal{P}_{BiSt} \rightarrow y_i := 1; x_i := x_i;$
  $\mathcal{G}_{A^{\vartriangle}B9} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B10} :: [\!]\mathcal{G}_{A1} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B2i} > \Delta_{BiSt}) \rightarrow x_i := \mathbf{max}(x_j|\forall x_j \in \Pi \setminus \{P_i\});$
  $\mathcal{G}_{A^{\vartriangle}B11} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B1} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i := x_i;$
  $\mathcal{G}_{A^{\vartriangle}B12} :: [\!]\mathcal{G}_{A2} \wedge \mathcal{G}_{B2} \wedge \neg\mathcal{P}_{BiSt} \wedge (\Delta_{B1i} > \Delta_{BiSt}) \rightarrow x_i := x_i;$
}
```

Fig. 4.8: Sub-algorithm $\mathcal{SSMax}_{i_e}{}^{\vartriangle}\mathcal{SSBiSt}_{i_e}$

Theorem 4.13. *Algorithm* $\mathbb{\$\$Max}^{\vartriangle}\mathbb{\$\$Bi\$t}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{S\,Max} \wedge \mathcal{P}_{BiSt}$ *under any weakly-fair scheduler* $\mathbb{D}_W$.

The presence of ranking function Δ_{BiSt} in the guards of $\mathbb{\$\$Max}^{\vartriangle}\mathbb{\$\$Bi\$t}$ prevents $\mathbb{D}_W$ from generating an execution whose projection on $\mathbb{\$\$Bi\$t}$ never converges to $\mathcal{P}_{BiSt}$. Figure 4.9 shows how actions of $\mathbb{\$\$Bi\$t}$ are shielded in a trace of an execution of the composed algorithm. Let processes P_ι and P_η be *ODD* processes and processes P_κ and P_τ be *EVEN* processes. Assume that the projection of global state σ_i on $\mathbb{\$\$Bi\$t}$ is σ_{2^n-1} where the local states of all processes are *true*. Algorithm $\mathbb{\$\$Bi\$t}$ does not execute its guarded command until scheduler $\mathbb{D}_W$ selects an *EVEN* process P_κ; no guarded command of $\mathbb{\$\$Bi\$t}$ is executed in state σ_t as scheduler $\mathbb{D}_W$ again selects an *EVEN* process P_τ.

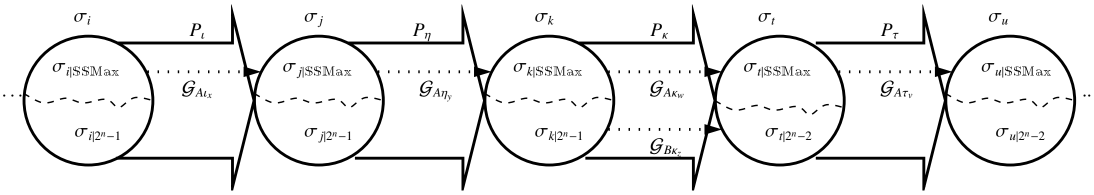

Fig. 4.9: Filtering of Execution Steps of $\mathbb{\$\$Bi\$t}$ in $\mathbb{\$\$Max}^{\vartriangle}\mathbb{\$\$Bi\$t}$

The composed algorithm $\mathbb{SSMax}^{\triangle}\mathbb{SSBiSt}$ does not have the guarded commands of Type 4 because algorithm $\mathbb{SSBiSt}$ has exactly one enabled guarded command in every state and in every process. Hence $\neg\mathcal{G}_{B1} \wedge \neg\mathcal{G}_{B2}$ is always false.

Remark 4.4. Note that, although the composed algorithm looks complicated compared to the component algorithms, the increase in size is due to the conjunction of the guarded commands. However, due to the relative simplicity of the rules of combining guards of the component algorithms, it is not difficult to automatize the process of generating composed algorithms.

Self-Stabilizing Equalizer Algorithm

Algorithm $\mathbb{SSEqual}$ is defined over a completely connected graph of processes. We assume that the process identifiers are totally ordered. Figure 4.10 shows the sub-algorithm implemented by each process in the system. The local state of each process in $\mathbb{SSEqual}$ is represented by a positive integer x_i. Every process compares its local state with the local states of its neighbors and, if there exists a neighbor with different local state, then the process copies the local state of the neighbor. In case there are multiple neighbors with different local states, then a process copies the local state of the neighbor whose identifier is greater than its own and smallest among the processes with different local states.

```
process $P_i$
{
  var int $x_i$;
  $\mathcal{G}_{C1}$ :: $[] x_i \neq x_{(i+1)\mathbf{mod}\ n} \rightarrow x_i := x_{(i+1)\mathbf{mod}\ n}$;
  $\mathcal{G}_{C2}$ :: $[] (x_i = x_{(i+1)\mathbf{mod}\ n}) \wedge (x_i \neq x_{(i+2)\mathbf{mod}\ n}) \rightarrow x_i := x_{(i+2)\mathbf{mod}\ n}$;
  ⋮
  $\mathcal{G}_{Ck}$ :: $[] (x_i = x_{(i+1)\mathbf{mod}\ n}) \wedge (x_i = x_{(i+2)\mathbf{mod}\ n}) \cdots \wedge (x_i \neq x_{(i+k)\mathbf{mod}\ n}) \rightarrow x_i := x_{(i+k)\mathbf{mod}\ n}$;
  ⋮
  $\mathcal{G}_{Cn-1}$ :: $[] (x_i = x_{(i+1)\mathbf{mod}\ n}) \wedge (x_i = x_{(i+2)\mathbf{mod}\ n}) \wedge \cdots \wedge (x_i \neq x_{(i+n-1)\mathbf{mod}\ n}) \rightarrow$
                                                  $x_i := x_{(i+n-1)\mathbf{mod}\ n}$;
}
```

Fig. 4.10: Sub-Algorithm $\mathcal{SSEqual}_i$

Let $\mathcal{P}_{eq}$ be a predicate defined over the states of algorithm $\mathbb{SSEqual}$ that holds true if local states of all the processes are equal, that is, $\mathcal{P}_{eq} = \bigwedge_{\forall i,j \in \Pi}(x_i = x_j)$. Predicate $\mathcal{P}_{eq}$ is closed under the execution of algorithm $\mathbb{SSEqual}$. However, convergence of algorithm $\mathbb{SSEqual}$ towards predicate $\mathcal{P}_{eq}$ requires rather constrained scheduling strategies. For example, consider an execution of $\mathbb{SSEqual}$ –shown in Figure 4.11– under a scheduler $\mathbb{D}_{RR}$ that selects one of the enabled processes in the round-robin fashion. The system consists of four processes and starts in state $\sigma_\alpha = \langle b, b, c, d \rangle$, where b, c and, d are arbitrary positive integers. The bold letter next to a node in Figure 4.11 denotes the local state of the process represented by the node. All processes have enabled guarded commands in σ_α. Scheduler $\mathbb{D}_{RR}$ begins with process P_2 in state σ_α. Scheduler $\mathbb{D}_{RR}$ selects the constituent processes in the subsequent states repeatedly in the following order: $\langle P_3, P_4, P_1, P_2 \rangle$. Consequently $\mathbb{SSEqual}$ again reaches σ_α after three execution rounds. As a result of this strategy $\mathbb{SSEqual}$ never reaches a state where predicate $\mathcal{P}_{eq}$ holds. Note that this strategy is weakly fair because each process is selected infinitely often. An edge in the communication graph of system is said to be "balanced" if processes connected by the edge have the same local state. We now define a scheduler which helps algorithm $\mathbb{SSEqual}$ to stabilize.

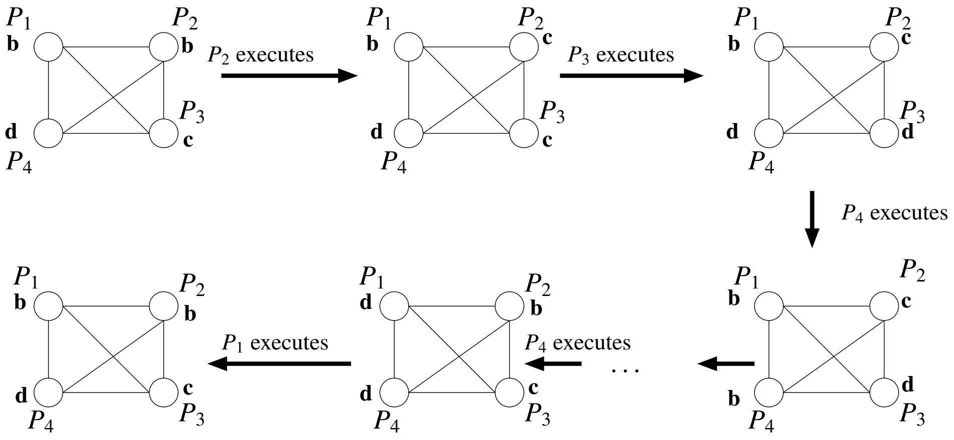

Fig. 4.11: Trace of a Diverging Execution of $\mathbb{\$\$Equal}$

Definition 4.11. *Scheduler $\mathbb{D}_{SEq}$ selects an enabled process in every execution step while fulfilling the following constraints:*

1) an infinitely often enabled process is selected infinitely often.
2) in any state select a process whose execution leads to an increase in the number of balanced edges in the communication graph.

Indeed, as the proof of the following theorem shows, scheduler $\mathbb{D}_{SEq}$ is a very benign scheduler.

Theorem 4.14. *Algorithm $\mathbb{\$\$Equal}$ is self-stabilizing with respect to predicate $\mathcal{P}_{eq}$ under scheduler $\mathbb{D}_{SEq}$.*

Proof. Consider a function $\varDelta_{Eq}= \frac{n\cdot(n-1)}{2} - \sum_{\forall i,j \in \Pi} \delta(x_i - x_j)$ defined over states of $\mathbb{\$\$Equal}$; $\delta(x)$ is 1 if $x = 0$ and 0 otherwise. In a state where all processes have different local states, then the action of any process leads to a decrease in the value of ranking function $\varDelta_{Eq}$ because term $\sum_{\forall i,j \in \Pi} \delta(x_i - x_j)$ increases from 0 to 1. Let σ_e be a global state where the local state of every process belongs to a set of $\frac{n}{2}$ different integers and each integer is assigned to exactly two processes. Any action in state σ_e decreases the value of $\varDelta_{Eq}$ by 1 because any action increases the number of "balanced" edges by 2 and increases the number of "unbalanced" edges by 1. Similarly, in any state where integer values are equally distributed over the processes, the execution of any enabled guarded command decreases the value of $\varDelta_{eq}$ by 1.
Let global σ_a be a state of algorithm $\mathbb{\$\$Equal}$ such that c processes have same local state l_i and the number of processes having local state different from l_i is smaller than c. If any process having local state l_i executes its action in global state σ_a then the number of processes having same local states would decrease and the value of $\varDelta_{Eq}$ would increase. However, such an execution step is not possible under scheduler $\mathbb{D}_{SEq}$ because the step leads to a decrease in the number of balanced edges in the graph (by definition). Therefore, in state σ_a, scheduler $\mathbb{D}_{SEq}$ selects only those processes which copy l_i as their local state; such an execution step leads to a decrease in $\varDelta_{Eq}$. It can be proven in an analogous manner that, in any state, $\mathbb{D}_{SEq}$ allows a process, with local state l_y to copy an integer l_x only if the number of processes having l_x as local state is larger than the number of processes having l_y as local state. The theorem follows from the observation that the minimum value of ranking function $\varDelta_{Eq}$ is 0 and, it is attained in the global state where all the processes have same local state. □

Lifting Composition of Algorithms $\mathbb{SSBiSt}$ *and* $\mathbb{SSEqual}$

The composition of algorithms $\mathbb{SSBiSt}$ and $\mathbb{SSEqual}$ is not trivial because both algorithms have very stringent scheduling requirements for convergence. Furthermore, due to rather specific constraints on schedulers, a strategy applicable to one algorithm can hamper convergence of the other. For example, consider a strategy ϱ_{BiSt} of $\mathbb{D}_{BiSt}$ which selects processes in a round-robin manner starting with an *EVEN* process. While algorithm $\mathbb{SSBiSt}$ converges to predicate $\mathcal{P}_{BiSt}$ under ϱ_{BiSt}, algorithm $\mathbb{SSEqual}$ never reaches the states satisying $\mathcal{P}_{eq}$ if ϱ_{BiSt} is used. Hence, lifting composition is a viable alternative as it disables divergence-inducing guards of algorithm $\mathbb{SSEqual}$. Figures 4.12 and 4.13 show the resultant sub-algorithms.

Theorem 4.15. *Algorithm* $\mathbb{SSBiSt}\vartriangle\mathbb{SSEqual}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{BiSt} \wedge \mathcal{P}_{eq}$ *under scheduler* $\mathbb{D}_{BiSt}$.

Any process in $\mathbb{SSBiSt}\vartriangle\mathbb{SSEqual}$ –provided it is selected by scheduler $\mathbb{D}_{BiSt}$– does not execute an action of $\mathbb{SSEqual}$ if it leads to a decrease in the number of balanced edges.

Scheduler $\mathbb{D}_{BiSt}$ does not violate weak fairness because it is constrained to select an infinitely often enabled process infinitely often. Additionally, $\mathbb{SSBiSt}$ ensures that all processes are enabled in every state.

4.6 Summary

We defined a new compositional operator, namely lifting composition, for self-stabilizing algorithms. Lifting composition exploits the existence of a known ranking function of a self-stabilizing algorithm (and the implicit knowledge "coded into it") to transform its property via composition. We showed how lifting composition preserves the self-stabilization property of a self-stabilizing algorithm under the scheduler of its counterpart. The effect of the relationship between respective schedulers of component algorithms on the scheduler of the composed algorithm was also analyzed. In particular, we showed how a weak-stabilizing algorithm can be transformed to into a self-stabilizing algorithm via lifting composition. Application of the composition method was illustrated by composing the self-stabilizing algorithms with incompatible scheduling constraints.

It was, however, assumed that the topology of the system permits such an online evaluation of the ranking function in each process. This assumption constricts the usage of lifting composition to the systems with a fully connected topology. Additional algorithmic machinery is, therefore, required to support the continuous evaluation of the ranking function in a system with arbitrary communication graphs. Since our ultimate goal is to have the complete system to be self-stabilizing, a lower layer supporting evaluation of a ranking function must also be self-stabilizing. We describe a modular approach to construct such a self-stabilizing lower layer in the next chapter.

```
process $P_i$
{
  var bool $y_i$;
  var int $x_i$;
  macro $\mathcal{P}_{eq} \equiv \bigwedge_{\forall k,l \in \Pi}(x_k = x_l)$
  macro $\Delta_{Eq} \equiv \frac{n\cdot(n-1)}{2} - \sum_{\forall:k,l \in \Pi} \delta(x_k - x_l)$
  $\mathcal{G}_{B\triangle C1} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 1; x_i := x_{(i+1)\mathbf{mod}\ n};$
   $\vdots$
  $\mathcal{G}_{B\triangle Cn-1} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 1; x_i := x_{(i+n-1)\mathbf{mod}\ n};$
  $\mathcal{G}_{B\triangle Cn} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 0; x_i := x_{(i+1)\mathbf{mod}\ n};$
   $\vdots$
  $\mathcal{G}_{B\triangle C2n-2} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 0; x_i := x_{(i+n-1)\mathbf{mod}\ n};$
  $\mathcal{G}_{B\triangle C2n-1} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 1; x_i := x_{(i+1)\mathbf{mod}\ n};$
   $\vdots$
  $\mathcal{G}_{B\triangle C3n-3} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 1; x_i := x_{(i+n-1)\mathbf{mod}\ n};$
  $\mathcal{G}_{B\triangle C3n-2} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 0; x_i := x_{(i+1)\mathbf{mod}\ n};$
   $\vdots$
  $\mathcal{G}_{B\triangle C4n-4} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 0; x_i := x_{(i+n-1)\mathbf{mod}\ n};$
  $\mathcal{G}_{B\triangle C4n-3} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 1;$
   $\vdots$
  $\mathcal{G}_{B\triangle C5n-5} :: [\,]\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 1;$
  $\mathcal{G}_{B\triangle C5n-4} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 0;$
   $\vdots$
  $\mathcal{G}_{B\triangle C6n-6} :: [\,]\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 0;$
  $\mathcal{G}_{B\triangle C6n-5} :: [\,]\mathcal{G}_{B1} \wedge \neg\mathcal{G}_{C1} \wedge \neg\mathcal{G}_{C2} \wedge \cdots \wedge \neg\mathcal{G}_{Cn-1}; \rightarrow y_i := 1;$
  $\mathcal{G}_{B\triangle C6n-4} :: [\,]\mathcal{G}_{B2} \wedge \neg\mathcal{G}_{C1} \wedge \neg\mathcal{G}_{C2} \wedge \cdots \wedge \neg\mathcal{G}_{Cn-1}; \rightarrow y_i := 0;$
}
```

Fig. 4.12: Sub-algorithm $\mathcal{SSBiSt}_{i_o}{}^{\triangle}\mathcal{SSEqual}_{i_o}$

```
process $P_i$
{
  var bool $y_i$;
  var int $x_i$;
  macro $\mathcal{P}_{eq} \equiv \bigwedge_{\forall k,l \in \Pi}(x_k = x_l)$
  macro $\Delta_{Eq} \equiv \frac{n\cdot(n-1)}{2} - \sum_{\forall:k,l\in\Pi}\delta(x_k - x_l)$
  $\mathcal{G}_{B\triangle C1} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 0; x_i := x_{(i+1)\mathbf{mod}\,n}$;
  $\vdots$
  $\mathcal{G}_{B\triangle Cn-1} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 0; x_i := x_{(i+n-1)\mathbf{mod}\,n}$;
  $\mathcal{G}_{B\triangle Cn} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 1; x_i := x_{(i+1)\mathbf{mod}\,n}$;
  $\vdots$
  $\mathcal{G}_{B\triangle C2n-2} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} < \Delta_{Eq}) \rightarrow y_i := 1; x_i := x_{(i+n-1)\mathbf{mod}\,n}$;
  $\mathcal{G}_{B\triangle C2n-1} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 0; x_i := x_{(i+1)\mathbf{mod}\,n}$;
  $\vdots$
  $\mathcal{G}_{B\triangle C3n-3} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 0; x_i := x_{(i+n-1)\mathbf{mod}\,n}$;
  $\mathcal{G}_{B\triangle C3n-2} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 1; x_i := x_{(i+1)\mathbf{mod}\,n}$;
  $\vdots$
  $\mathcal{G}_{B\triangle C4n-4} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \mathcal{P}_{eq} \rightarrow y_i := 1; x_i := x_{(i+n-1)\mathbf{mod}\,n}$;
  $\mathcal{G}_{B\triangle C4n-3} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 0$;
  $\vdots$
  $\mathcal{G}_{B\triangle C5n-5} :: []\mathcal{G}_{B1} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 0$;
  $\mathcal{G}_{B\triangle C5n-4} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{C1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 1$;
  $\vdots$
  $\mathcal{G}_{B\triangle C6n-6} :: []\mathcal{G}_{B2} \wedge \mathcal{G}_{Cn-1} \wedge \neg\mathcal{P}_{eq} \wedge (\Delta_{C1i} > \Delta_{Eq}) \rightarrow y_i := 1$;
  $\mathcal{G}_{B\triangle C6n-5} :: []\mathcal{G}_{B1} \wedge \neg\mathcal{G}_{C1} \wedge \neg\mathcal{G}_{C2} \wedge \cdots \wedge \neg\mathcal{G}_{Cn-1}; \rightarrow y_i := 0$;
  $\mathcal{G}_{B\triangle C6n-4} :: []\mathcal{G}_{B2} \wedge \neg\mathcal{G}_{C1} \wedge \neg\mathcal{G}_{C2} \wedge \cdots \wedge \neg\mathcal{G}_{Cn-1}; \rightarrow y_i := 1$;
}
```

Fig. 4.13: Sub-algorithm $\mathcal{SSBiSt}_{i_e}{}^{\triangle}\mathcal{SSEqual}_{i_e}$

5

Scheduler Transformation of Self-Stabilizing Algorithms

5.1 Introduction

A critical part of designing a self-stabilizing algorithm is the proof that the algorithm converges to the behavior outlined in its specification. As discussed earlier, such proofs are not easy to draw and the automatic methods to do so do not scale well enough. The proofs of self-stabilization also depend on the underlying scheduler and the fairness assumption [28]; the increased generality of a scheduler –embodying scheduling strategies and fairness assumptions– makes convergence proofs progressively complicated. Scheduler assumptions remain a crucial part of the proof even if a compositional method is used to design a self-stabilizing algorithm. Indeed, an incompatible scheduler can render one of the component algorithms divergent.

In order to get around the complexity of proofs due to the underlying schedulers, Gouda and Haddix [85] suggested the use of a so-called "alternator" to preserve the self-stabilization property under a distributed scheduler and, in turn, spurred investigation into such transformers. These transformers, however, require that the original algorithm must be self-stabilizing under all weakly-fair schedulers. While the transformation of the self-stabilization property from weakly-fair sequential schedulers to distributed schedulers is well-studied, methods required for self-stabilizing algorithms which exhibit convergence under very restrictive schedulers, such as weakly-stabilizing algorithms [53], have not been investigated extensively. As Devismes *et al.* [88] showed, a weakly-stabilizing algorithm can at best exhibit probabilistic convergence under a distributed randomized scheduler. Nonetheless, there is a need for a method to transform *probable* convergence to *guaranteed* convergence.

The crux of the challenge is to identify and enable –in every state of a system executing the algorithm– processes whose actions are "beneficial" to the overall convergence of the algorithm. It has been recently shown that during the design phase of a distributed algorithm, the results of verification can be used to transform the algorithm such that, the amount of knowledge a process has, determines whether its actions are enabled or not [89]. This raises the question whether a similar approach can be used to design a transformer for self-stabilizing algorithms under very restrictive schedulers.

Outline. We suggest the usage of a ranking function, returned as a by-product of a convergence proof of a distributed algorithm under a specific (and restrictive) scheduler, to transform a self-stabilizing algorithm. We present a method to transfer the self-stabilization property of a distributed algorithm proven under a very restrictive scheduler to any *weakly-fair scheduler*. The transformation embodies a *progress monitor* [90] which tracks the progress of a self-stabilizing algorithm towards its correct behavior under any generic scheduler. We also provide a method to increase inherent concurrency in the transformed system by exploiting the very structure of the ranking function.

The chapter is structured as follows. Section 5.2 provides an overview of the related work. The transformation is explained in Section 5.3 together with the proofs. An optimization method to increase concurrency in the transformed system is presented in Section 5.3.3. We analyze the transformation technique from a rather abstract perspective in Section 5.4. The chapter concludes with a summary in Section 5.5.

5.2 Related Work

The difficulty of designing a self-stabilizing algorithm from scratch has led to the development of various methods to transform algorithms that are not self-stabilizing in the first place into respective self-stabilizing algorithms. A transformer that employs a *supervisor* process to reset a global system state has been proposed in [91]. The supervisor process periodically requests a global snapshot and resets the system to a pre-defined initial state in case the snapshot violates some (safety) predicate. This methods assumes the existence of a distinguished process in the system. Awerbuch and Varghese [92] presented a transformer that converts a synchronous distributed algorithm into an asynchronous self-stabilizing algorithm via a *resynchronizer*. A resynchronizer simulates a synchronous algorithm under an asynchronous environment in a self-stabilizing manner. The underlying principle of the resynchronizer-based transformer is to check the output of each process after T rounds, where T is the time complexity of the algorithm, and restart the algorithm if any inconsistency is detected during the checking phase. It has been shown in [93, 94] that for some problems, it is sufficient to check the inconsistency *locally*. A *local stabilizer* that transforms *online* synchronous distributed algorithms into respective self-stabilizing algorithms is presented in [95]. Each process implementing a local stabilizer maintains a data structure termed as *pyramid*; a pyramid contains d values, where d is the diameter of the system, such that i^{th} entry represents the local states of the processes within i hops in $T-i$ rounds. An inconsistent system state is detected if pyramids of two neighboring processes do not match or if a process is not able to reconstruct its current local state using relevant entries of the pyramids of its neighbors. In case an inconsistency is detected, then the system is repaired via pyramids of non-faulty processes. Beauquier *et al.* presented a set of transient fault detectors for various families of tasks [96]; a transient fault detector ensures that eventually, a transient fault is detected by at least one process in the system. Such a transient fault detector can be composed with a self-stabilizing reset algorithm [97] to transform a distributed algorithm into a self-stabilizing algorithm.

A lock-based transformer is presented in [98, 99] to transfer the self-stabilization property of an algorithm from a sequential scheduler to a distributed scheduler. The transformer ensures that a process can execute its guarded command only if it gets the lock. The conflict among multiple processes competing for the lock is resolved on the basis of timestamps sent along with the request for the lock. This transformer, however, may not preserve the self-stabilization property if convergence requires a fair scheduler. Self-stabilizing solutions to the classical problem of Dining Philosphers [100] have been proposed to transfer the self-stabilization property of an algorithm proven under a weakly fair scheduler to a distributed scheduler [101, 102, 103]. A self-stabilizing algorithm to implement strong fairness under a weakly-fair scheduler is presented in [104]. This algorithm emulates the behavior under a strongly fair scheduler by ensuring that whenever a process executes its guarded command, it does so exclusively in its 2-neighborhood. A *maximal* algorithm to emulate strong fairness is presented [105] where maximality implies that the algorithm is able to produce *all possible* strongly-fair scheduling strategies. However, this algorithm is not self-stabilizing.

5.3 Transformation of Self-Stabilizing Algorithms

We now present a method to transform an algorithm that is self-stabilizing under a given scheduler to an algorithm that is self-stabilizing under any weakly-fair scheduler. The underlying idea is presented before the constituents of the transformer are explained. We then show the correctness of the transformation method.

Overall Idea

The convergence property of a self-stabilizing algorithm is particularly prone to the scheduling assumptions. It has been discussed earlier that an algorithm –that has been shown to be self-stabilizing otherwise– does not converge to the set of legal states if the constraints on the scheduler are relaxed. In particular, a malevolent scheduler can regularly select an enabled process which sends outdated or wrong information to its neighbors and, thus, prevent an algorithm from reaching the set of states satisfying the safety predicate. The non-determinism inherent in the algorithm allows the scheduler to mold such diverging executions.

In order to produce the specified behavior even in presence of an adversarial scheduler, intrinsic non-determinism of an algorithm must be reduced. A way to reduce unnecessary non-determinism could be to limit the set of enabled processes to those processes whose guarded commands help the algorithm to converge to the safety predicate. The set of enabled process in each execution step can be limited by ensuring that no guards in the processes which "harm" convergence are enabled. However, it implies that such processes should "know" when their actions are counterproductive for overall correctness of the system. Assume that each process P_i is equipped with a variable go_i such that go_i is true whenever an enabled guarded command of P_i is helpful for convergence and false if it is not. As the algorithm is supposed to be self-stabilizing and can have an arbitrary initial state, the value of go_i in all processes with detrimental guarded commands can be true and the value of go_i in the processes with favorable guarded commands can be false. It is, thus, in general impossible for a process to know by itself whether its actions are favorable or not.

An alternative would be to "instrument" each process with extra information so that it can correctly decide when to disable guarded commands. Convergence is a global property and, therefore, the enhancement of a process' knowledge must carry some form of global information. A global snapshot at any instant is the maximum information that a process can get and it is sufficient in this scenario. However, a process must also have some extra knowledge to act upon the information in the form of a global snapshot.

Since the algorithm –which we aim to transform– is self-stabilizing under a specific scheduler, we have a safety predicate as part of the specification. Recall that safety predicate holds true only for the legal states; a process can decide whether it is in a legal state or not provided that the guards are strengthened with the safety predicate. Nonetheless, safety predicate can only distinguish between a legal and an erroneous system state. Hence, a process with enhanced guarded commands cannot differentiate two erroneous states –an ability which is critical while the system is converging towards the set of legal states. The knowledge of a global system state and the safety predicate is not enough to decide whether a process should execute its guarded command or not. Every process in the system must be able to *gauge the severity of every erroneous state*. The requirement can also be interpreted as the ability to measure the progress of the system towards the set of legal states.

A well-foundedness argument is used to show the convergence property of a self-stabilizing algorithm. As the algorithm under consideration is also self-stabilizing –though under a specific scheduler– we are provided with a ranking function which encapsulates the well-foundedness argument. A ranking function assigns each global system state a value in a well-founded domain. Such a mapping has an interesting property: for every execution Ξ emanating from a state σ_i such that Ξ has a suffix in the set of legal states, the values assigned by the ranking function to the states that appear in Ξ after global state σ_i are lower than the value assigned to global state σ_i. A ranking function can, therefore, be used to distinguish between states while a system is converging to the set of legal states. Alternatively, a ranking function –supplied as part of the convergence proof– can be used to track the progress of the system towards the safety predicate.

The transformation essentially implements a “progress monitor” of self-stabilizing algorithm $\mathbb{A}$ which tracks the progress of algorithm $\mathbb{A}$ with respect to its convergence towards the states satisfying a safety predicate $\mathcal{P}_A$. The progress monitor uses knowledge of both predicate $\mathcal{P}_A$ and the ranking function Δ_A. Since every process must vet its action before executing them, the progress monitor is implemented in a distributive fashion. The transformation ensures that the actions of algorithm $\mathbb{A}$ are enabled only if they guarantee progress towards the set of safe states.

Continuous evaluation of the ranking function is the core of the transformation and, as discussed above, it requires the latest global snapshot to function correctly. Since we do not assume a fully connected communication infrastructure, functioning of such a ranking function-based progress monitor necessitates a global coordination mechanism to support it. The supporting coordination mechanism should also be self-stabilizing by itself because the variables belonging to the coordination algorithm can have arbitrary values in the initial state.

There are two alternative methods to implement the coordination mechanism in a self-stabilizing manner: 1) wave based algorithms [106] or 2) mutual exclusion algorithms. A wave-based algorithm – synonymously referred to as Propagation of Information with Feedback (PIF) algorithm– works in three phases: 1) broadcast, 2) feedback, and 3) cleaning. The initiator process starting the wave sends some information in the network by changing its state to broadcast. A node changes its state to broadcast if one of its neighbor is in broadcast phase; if a node cannot broadcast because all of its neighbor already have received the information being sent, it sets its state to feedback. The feedback wave reaches the initiator process once all of its neighbors are in feedback phase. A process in feedback phase sets its state to cleaning phase if all of its neighbors are in feedback phase or in cleaning phase. The initiator process can start a new broadcast once it reaches the cleaning phase. Should we use a wave-based algorithm for global coordination, algorithm $\mathbb{A}$ would use waves to gather and distribute information in the following manner. Each process, first, broadcasts its local state to all the processes in the system and waits for feedback. After this first phase of information gathering is over, the process evaluates the enhanced guards, (possibly) executes them and, sends its updated local state. However, this method of implementation mandates that each process should run its own wave algorithm. Thus, at any time instant, the system might have multiple waves traveling in different directions. Coordinating multiple waves would require extra resources at each process because each process must maintain its position in multiple spanning trees. Additionally, due to the presence of multiple waves at any time instant, it would be difficult to guarantee that a process has the latest global state when it executes a guarded command of algorithm $\mathbb{A}$. The issues remain even if only one instance of wave algorithm is run by the system. A distinguished process is required if a single instance of the wave algorithm is used for the global coordination. Note that, with a single instance of the wave algorithm, the coordination mechanism resembles a self-stabilizing synchronizer [107]. Although memory requirements decrease considerably as a result of a single wave in the system, ensuring a latest global snapshot remains difficult. In this case, the special process floods the system with the latest global snapshot and sends the directive to execute enabled guarded commands to other processes after it receives confirmation of the receipt of the snapshot. However, the distinguished process cannot control the order of execution of guarded commands in the system once it floods the system with the permission to execute guarded commands. Assume that two non-neighbor processes P_i and P_j have enabled guarded commands after the permission to execute is sent by the special process. One of the two processes would evaluate the ranking function based on an outdated global snapshot –irrespective of the order in which they execute the enabled guarded commands– because the changes made by the process executing first would be visible to the other process only when the special process starts the next wave.

A mutual exclusion algorithm guarantees that, in any execution step, exactly one process accesses its critical section. Typically, the critical section of a process refers to the actions which access a resource shared across the distributed system. In contrast to a wave algorithm-based progress monitor, a progress monitor implemented with the help of a mutual exclusion algorithm can ensure that only one process executes the guarded command of the algorithm to be transformed at any time instant. However, a mutual exclusion-based progress monitor has to overcome the following limitations: 1) self-stabilizing mutual exclusion algorithms require a spanning tree with a distinguished process to work correctly [94, 27] and 2) mutual exclusion algorithms provide no support for snapshot collection. As we neither assume the presence of such a distinguished processor serving as a root nor a spanning tree, a self-stabilizing spanning tree algorithm is required for correct execution of the self-stabilizing mutual exclusion algorithm. Token-based mutual exclusion algorithms [108] use a system-wide unique token to rotate permission to execute a critical section among the processes. A self-stabilizing token based mutual exclusion algorithm can be adapted to support snapshot collection. We now describe the various constituents of the transformer.

Architecture of Transformer

Figure 5.1 shows the components and the layered architecture of the transformation along with the flow of information between the constituent algorithms. The constituent algorithms communicate with each other by reading and writing the variables shown in Figure 5.1. More specifically, variable $token_i$ is modified by the mutual exclusion layer and read by the modified algorithm. Variable $snapshot_i$ is modified by both mutual exclusion layer and the modified algorithm. A self-stabilizing spanning tree algorithm forms the lowest layer of the transformer. The spanning tree algorithm is composed with a self-stabilizing mutual exclusion algorithm via *fair composition*. Fair composition of algorithms $\mathbb{F}_{\mathbb{i}}$ and $\mathbb{F}_{\mathbb{j}}$ is the union of the set of guarded commands of $\mathbb{F}_{\mathbb{i}}$ and $\mathbb{F}_{\mathbb{j}}$; guarded commands of each component algorithm are executed in a fair fashion in the composed algorithm, that is, each process executes guarded commands of each algorithm infinitely often [27]. Fair composition uses the notion of a "master" and a "slave" algorithm; the slave algorithm reads the variables modified by the master algorithm assuming that its counterpart has stabilized. The slave algorithm stabilizes after the variables modified by the master algorithm satisfy the safety predicate of the master algorithm.

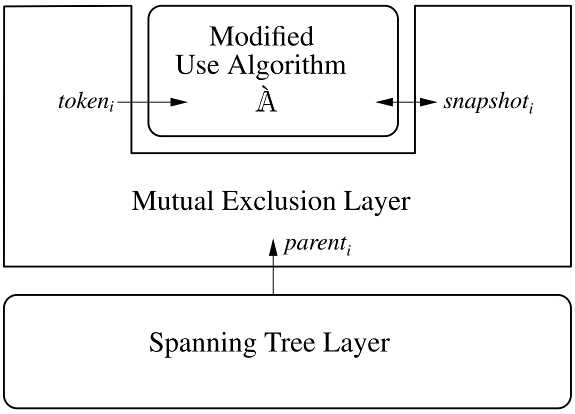

Fig. 5.1: Layered View of the Transformation

The self-stabilizing spanning tree algorithm functions as the master algorithm and, the mutual exclusion algorithm acts as the slave algorithm in the transformer. The self-stabilizing mutual exclusion algorithm and the spanning tree algorithm communicate via $parent_i$ variables; $parent_i$ variable points to the parent of a process P_i in the spanning tree and it is modified by the spanning tree algorithm during the course of algorithm's execution. The mutual exclusion algorithm uses variable $parent_i$ to circulate a token around the system. The modified algorithm forms the critical section of the mutual exclusion algorithm and the permission to execute is granted via the *token*$_i$ variable. The mutual exclusion algorithm is modified to collect a global snapshot as well and this information is passed on to the modified algorithm with the help of a so-called *snapshot*$_i$ variable. We are now ready to give the complete definition of the scheduler-oblivious transformation of a self- stabilizing algorithm.

5.3.1 Definition

Let algorithm $\mathbb{A}$ be self-stabilizing with respect to a predicate $\mathcal{P}_A$ under a specific scheduler $\mathbb{D}_A$. The convergence property of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ has been proven with the help of a ranking function Δ_A. Algorithm $\mathbb{A}$ consists of m_i guarded commands $\mathcal{G}_{A_{ix}}$, $1 \leq x \leq m_i$ in each process $P_i \in \Pi$. Recall that transforming an algorithm so that it has exactly one enabled guarded command per process and per state is not difficult. Under these assumptions, the scheduler-oblivious transformation of algorithm $\mathbb{A}$ –synonymously referred to as use algorithm– is defined as follows.

The transformation is carried out in three steps. In the first step, use algorithm $\mathbb{A}$ is modified by strengthening its guards. The modified algorithm $\mathbb{A}$ is then composed with the self-stabilizing mutual exclusion algorithm of [109, pp. 24–27] so that it forms the critical section of the mutual exclusion algorithm. The self-stabilizing mutex algorithm of [109] is also modified in order to be able to collect the global snapshot. In the last step, the modified self-stabilizing mutual exclusion algorithm is composed with the self-stabilizing spanning tree algorithm of [94]. We describe the algorithms implementing each layer in the following.

Spanning Tree Layer

Figure 5.2 shows the self-stabilizing spanning tree algorithm due to [94]. Each sub-algorithm of the self-stabilizing spanning tree algorithm uses three variables: *dis*$_i$, *parent*$_i$ and *root*$_i$ to construct a rooted spanning tree. The *root*$_i$ variable of process P_i contains the identifier of the root node of the spanning tree to which P_i belongs. The shortest distance between the root node and a process P_i is stored in variable *dis*$_i$. The variable *parent*$_i$ points to the parent of process P_i in the spanning tree. Every process P_i repeatedly reads the state of its neighbors and updates its local state if its neighborhood does not satisfy certain predicates. The system is said to be in the *correct global state* if the local state of *each* process P_i satisfies predicate $\mathcal{P}_{\text{span}_i}$, that is:

$$[(i \;=\; root_i) \;\textbf{OR}\; (P_i \text{ is a node in a tree})] \;\textbf{AND}\; (root_i \;\leq\; \min_{j \in \text{neighbor}(i)}(root_j)$$

where neighbor(i) returns the set of the neighbors of process P_i. Process P_i considers itself root if 1) the values of the variables *root*$_i$ and *parent*$_i$ are equal to its own identifier and 2) the value of *dis*$_i$ is equal to 0. Process P_i is said to be a node in a tree if 1) the value of *root*$_i$ is less than its own ID, 2) the value of variable *parent*$_i$ belongs to the set returned by neighbor(i), 3) the value of *root*$_i$ is equal to that of $root_{parent_i}$, and 4) the difference between the values of the variables *dis*$_i$ and dis_{parent_i} is exactly 1. Predicate $\mathcal{P}_{\text{forest}}$ holds if 1) no cycle exists and 2) the graph defined by the *parent*$_i$ pointers of the processes in the system is a forest. Predicate $\mathcal{P}_{\text{forest}}$ is true if predicate $\mathcal{P}_{\text{forest}_i}$:

$$(P_i \;=\; root_i) \;\textbf{OR}\; [(P_i \text{ is node in a tree}) \;\textbf{AND}\; (root_i \;\leq\; \min_{j \in \text{neighbor}(i)}(root_j)]$$

holds true at each process P_i. Note that predicate $\mathcal{P}_{\text{span}_i}$ implies predicate $\mathcal{P}_{\text{forest}_i}$, vice versa, however, is not true. If predicate $\mathcal{P}_{\text{forest}_i}$ does not hold at a process P_i, then it marks itself as the root by assigning the variables *parent*$_i$ and *root*$_i$ its own ID and setting variable *dis*$_i$ to 0. Process P_i requests permission to join the spanning tree of a neighbor if 1) $\mathcal{P}_{\text{forst}_i}$ holds true, 2) $\mathcal{P}_{\text{span}_i}$ does not hold, and 3) the identifier of the root of the spanning tree of the neighbor is lower than that of its own. A process sends a request only if it is not waiting for the reply of any previously sent request. Such a request to join a spanning tree is routed through the members of the spanning tree to the root. The requesting node joins the tree only when it receives a “grant” to do so from the neighbor through which it initiated the request.

Each process has four variables –$request_i$, $source_i$, $next_i$ and $direction_i$– to handle the requests and the grants to join a spanning tree. The variable $request_i$ contains the identifier of the node that has sent request to join the tree to which P_i belongs. Variable $source_i$ contains the identifier of the neighboring process from which process P_i copied the value of $request_i$. Process P_i assigns its own ID to the variables $request_i$ and $source_i$ if it tries to join a spanning tree. The variable $next_i$ is the ID of the node through which process P_i tries to forward a request. The variable $direction_i$ has two potential values – *grant* and *ask*– if variable $direction_i$ is equal to *ask* then it implies that the process with ID equal to $request_i$ wishes to join the tree to which process P_i belongs; if the request to do so is granted, then variable $direction_i$ contains the value *grant*. Process P_i participates in forwarding requests and permissions to join to the tree to which it belongs only if predicate $\mathcal{P}_{\text{span}_i}$ holds. If predicate $\mathcal{P}_{\text{span}_i}$ holds and there exists a neighbor process P_j with the variables $direction_j$ and $next_j$ being equal to *ask* and i respectively, then P_i forwards the request of P_j by assigning j, j, $parent_i$, and *ask* to the variables $request_i$, $source_i$, $next_i$, and $direction_i$ respectively. A request from a child in the spanning tree is also handled in the similar way. Process P_i does not propagate any request until a previously forwarded request is granted. Process P_i allows another process to join its tree by setting $direction_i$ to *grant* if process P_i is root and predicate $\mathcal{P}_{span_i}$ holds true. A non-root process P_i forwards a grant from its parent in a similar fashion if 1) variable $next_i$ is equal to $parent_i$, 2) variable $\text{request}_{parent_i}$ is equal to variable $request_i$, 3) variable $source_{parent_i}$ is i, and 4) predicate $\mathcal{P}_{span_i}$ holds. Process P_i joins a spanning tree if 1) predicate $\mathcal{P}_{forest_i}$ holds, 2) predicate $\mathcal{P}_{span_i}$ does not hold, 3) variable $direction_i$ is *ask*, 4) variable $next_i$ is j, and 5) variable $direction_j$ and $request_j$ are *grant* and i respectively. While joining a tree process P_i sets 1) variable $parent_i$ to j, 2) variable $root_i$ to $root_j$, 3) variable $distance_i$ to $distance_j + 1$, and 4) variable $direction_i$ to *null*.

The algorithm ensures that multiple spanning trees in a forest merge because processes continuously compare the ID of their root with those of their neighbors, and once a process P_i discovers that one of its neighbors has a root with lower ID, then it sends a request to join the tree to which its neighbor belongs; in this fashion, eventually, all the processes in the previous tree of process P_i join the tree with lower root ID.

The algorithm also eliminates any false roots; an identifier f is said to be a false root if there exist a process P_i such that $root_i$ is equal to f and there exists no process with identifier equal to f. Consider a false root f_m such that f_m is smaller than ID of all the processes in the system. Since there exists no node with ID f_m, no request to join tree of f_m will be replied back with *grant*. Also, as no process P_i has variable $parent_i$ equal to f_m, one of the process P_j along the branch of f_m tree will set itself as root. A false root f_m is purged out of the system as all the processes along the branches of f_m reset their root variables. Any cycles in the graph are also removed when a process in the cycle checks the truth values of the predicates $\mathcal{P}_{\text{span}_i}$ and $\mathcal{P}_{\text{forest}_i}$. For example, consider the graph shown in Figure 5.3. The system contains six processes and the values of $root_i$ and dis_i are shown next to the ID of each process; dashed arrows point to the parent of each process. The system is not in a correct state –a spanning tree consisting of all nodes does not exist – as there exist a tree and a cycle of four processes. Predicate $\mathcal{P}_{\text{forest}_i}$ does not hold in processes P_3 and P_4. Therefore, process P_3 sets itself as root; in the next step, process P_3 realizes that process P_1 has an ID smaller than its own, and thus, joins the tree of process P_1. Since predicate $\mathcal{P}_{\text{forest}_4}$ does not hold, process P_4 sets itself as root. As a result of the action of process P_4, predicate $\mathcal{P}_{\text{forest}_6}$ does not hold at process P_6 anymore and it also sets itself as root. Process P_4 joins the tree of process P_1 by sending request through process P_2. Process P_5 sets itself as root because predicate $\mathcal{P}_{\text{forest}_5}$ is not true after the actions of processes P_3 and P_6. Consequently, processes P_5 and P_6 join the tree rooted at process P_1 and the system, subsequently, reaches a correct global state.

Remark 5.1. Although there are other instances of spanning tree algorithms in the literature, however most of these self-stabilizing algorithms require a distinct node to build a spanning tree. Should one of

```
process $P_i$
{
  localvar $root_i, request_i, source_i, parent_i, distance_i, next_i, direction_i$;
  macro $\mathrm{requestsent}_i \equiv (\exists l \in \mathrm{neighbor}(i) \mid root_l = \max_{m \in \mathrm{neighbor}(i)}(root_m) > root_i)$
                  $\wedge (request_i = source_i = i)$
                  $\wedge (next_i = l) \wedge (direction_i = ask)$
  macro $\mathrm{req}_i \equiv ((\exists j \in \mathrm{neighbor}(i)) \wedge (parent_j = j \neq i)$
                  $\wedge (request_j = source_j = request_i = source_i = j = root_j)$
                  $\wedge (next_j = i) \wedge (direction_j = ask) \wedge (next_i = parent_i))$
                  $\vee ((\exists j \in \mathrm{neighbor}(i)) \wedge (parent_j = i)$
                  $\wedge (null \neq request_j = request_i \neq j)$
                  $\wedge (source_i = j)) \wedge (next_i = parent_i)$
                  $\wedge (next_j = i) \wedge (direction_j = ask))$
  macro $\mathrm{req}_i' \equiv \mathrm{req}_i \vee (request_i = next_i = source_i = direction_i = null)$
  [] $\neg\mathcal{P}_{\mathrm{forst}_i}$                                  $\rightarrow root_i := i,$
                                                 $parent_i := i,$
                                                 $distance_i := 0;$

  [] $\mathcal{P}_{\mathrm{forst}_i} \wedge$
    $(\exists j \in \mathrm{neighbor}(i) \mid root_j = \max_{k \in \mathrm{neighbor}(i)}(root_k))$
    $\wedge \neg\mathrm{requestsent}_i$                      $\rightarrow request_i := resource_i := i,$
                                                 $next_i := j, direction_i := ask;$
  [] $\mathcal{P}_{span_i} \wedge \neg\mathrm{req}_i'$     $\rightarrow request_i := source_i := next_i := direction := null;$
  [] $\mathcal{P}_{span_i} \wedge \mathrm{req}_i' \wedge \neg\mathrm{req}_i$
    $\wedge (\exists j \in \mathrm{neighbor}(i) \mid (direction_j = ask) \wedge (next_j = i)$
    $\wedge (request_j = j = root_j = source_j)$
    $\wedge (source_{parent_i} \neq i))$              $\rightarrow request_i := source_i := j,$
                                       $next_i := parent_j, direction_i := ask;$
  [] $\mathcal{P}_{span_i} \wedge \mathrm{req}_i' \wedge \neg\mathrm{req}_i$
    $\wedge (\exists j \in \mathrm{neighbor}(i) \mid (parent_j = i) \wedge (next_j = i)$
    $\wedge (direction_j = ask) \wedge (request_j \neq null)$
    $\wedge (request_j \neq j) \wedge (source_{parent_i} \neq i))$  $\rightarrow request_i := request_j, source_i := j,$
                                       $next_i := parent_i, direction_i := null;$
  [] $\mathcal{P}_{span_i} \wedge \mathrm{req}_i \wedge (P_i$ is a root$) \wedge (direction_i = ask) \rightarrow direction_i := grant;$
  [] $\mathcal{P}_{span_i} \wedge \mathrm{req}_i \wedge (next_i = parent_i = j) \wedge (direction_j = grant)$
    $\wedge (direction_i = ask) \wedge (request_i = request_j)$
    $\wedge (source_j = i)$                                $\rightarrow direction_i := grant;$
```

Fig. 5.2: Self-Stabilizing Spanning Tree Algorithm of [94]

these spanning tree algorithm be used in the transformer, a self-stabilizing leader election algorithm becomes mandatory for the transformer. The algorithm due to [94] –unlike other self-stabilizing spanning tree algorithms– selects a distinguished process while constructing a spanning tree, and is, therefore, used in the transformer.

Mutual Exclusion Layer

The mutual exclusion layer is implemented with the help of the self-stabilizing mutual exclusion algorithm due to Dolev [109, pp. 24]. Figures 5.4 and 5.5 show the self-stabilizing mutual exclusion algorithm of [109]. Process P_0 is the root process. The self- stabilizing mutual exclusion algorithm

```
  [] $\mathcal{P}_{\mathrm{forst}_i} \wedge \neg\mathcal{P}_{span_i} \wedge (direction_i = ask) \wedge (j \in \text{neighbor(i)})$
     $\wedge\, (request_i = request_j = source_i = root_i = i)$
     $\wedge\, (source_j = i) \wedge (direction_j = grant)$
     $\wedge\, (next_i = j) \wedge (root_j > root_i)$             $\rightarrow parent_i := j,$
                                                      $distance_i := distance_j + 1,$
                                                      $root_i := root_j,$
                                                      $request_i := source_i :=$
                                                      $next_i :=$
                                                      $direction_i := null;$
}
```

Fig. 5.2: Self-Stabilizing Spanning Tree Algorithm of [94] (Continued)

ensures that in any global state only one process can access its critical section. The access to critical section is regulated with the help of a "token" and a process can enter its critical section if and only if it has the token. In order to ensure that no process waits indefinitely for access to the critical section, the token is circulated among all the processes. The self-stabilizing algorithm circulates the token over the spanning tree constructed by the spanning tree layer. Since we require snapshots for the correct functioning of the transformation, the structure of the communication register used by the token circulation algorithm is modified. The communication register –shown in Figure 5.6– comprises of three parts: the variables used by the spanning tree (left compartment), the token part (middle compartment) and the global snapshot (right compartment). The leftmost part of a communication register $\mathbf{r}_{ij}$ is used by the spanning tree algorithm to construct and maintain a spanning tree; apart from the copies of the variables pertaining to the position of a process in a spanning tree, it contains the copies of the variables used for forwarding and granting the requests to join a spanning tree. The middle part of $\mathbf{r}_{ij}$ is used by the mutual exclusion algorithm for token circulation; the token part of communication register is an integer which indicates whether a process can enter the critical section or not. The communication register $\mathbf{r}_{ij}$ is extended by appending a compartment containing the process P_i's copy of a global snapshot. The global snapshot part is a vector of all those variables of the use algorithm $\mathbb{A}$ which are required to compute the value of ranking function $\varDelta_A$ in any global system state. For instance, if a local variable of a process P_i, x_i ($\forall P_i \in \varPi$), is used to compute the value of $\varDelta_A$, then each process P_i writes a copy of x_i in the vector representing the global snapshot.

The mutual exclusion algorithm is, essentially, composed of the two types of sub-algorithms. Actions of a process are dictated by its position in the spanning tree built by the lower layer; the root process implements a distinct sub-algorithm, all the other process, modulo their neighborhood, implement the second sub-algorithm. Each process uses an integer $token_i$ to control the access to its critical section. Additionally, each process P_i defines a total order on the write registers read by its children. Let $\lambda_i = \{\mathbf{r}_{iu}, \mathbf{r}_{iv} \cdots, \mathbf{r}_{iz}\}$ be a total ordering on the write registers of a non-root process P_i which, in turn, induces a total ordering on the set of its children processes. A non-root process P_i compares the value of $token_i$ with that of its parent process: if it is not equal to the token value of its parents, then P_i accesses its critical section. P_i assigns the token value of its parent to the variable $token_i$ after leaving the critical section. In order to pass the token to its descendants, process P_i writes its new token value to the communication register ($\mathbf{r}_{iu}$) meant for its first child in the spanning tree. A process returns the token by writing the new token value in the write communication register meant for its parent. Process P_i passes the token to the next child only after it gets the token back from its first child; that is, P_i copies the content of $\mathbf{r}_{ui}$ to the register $\mathbf{r}_{iv}$ and, P_v gets the token when $\mathbf{r}_{ui}$, $\mathbf{r}_{iu}$ and, $\mathbf{r}_{iv}$ are equal to $token_i$. P_i repeatedly passes the token amongst its children by copying the value of token written by a child P_x to the write register $\mathbf{r}_{iy}$ meant for the next process P_y in λ_i. A non-leaf process passes the token back to its parent after it gets back the token from its last child. The ordering λ_i of the write registers of a process

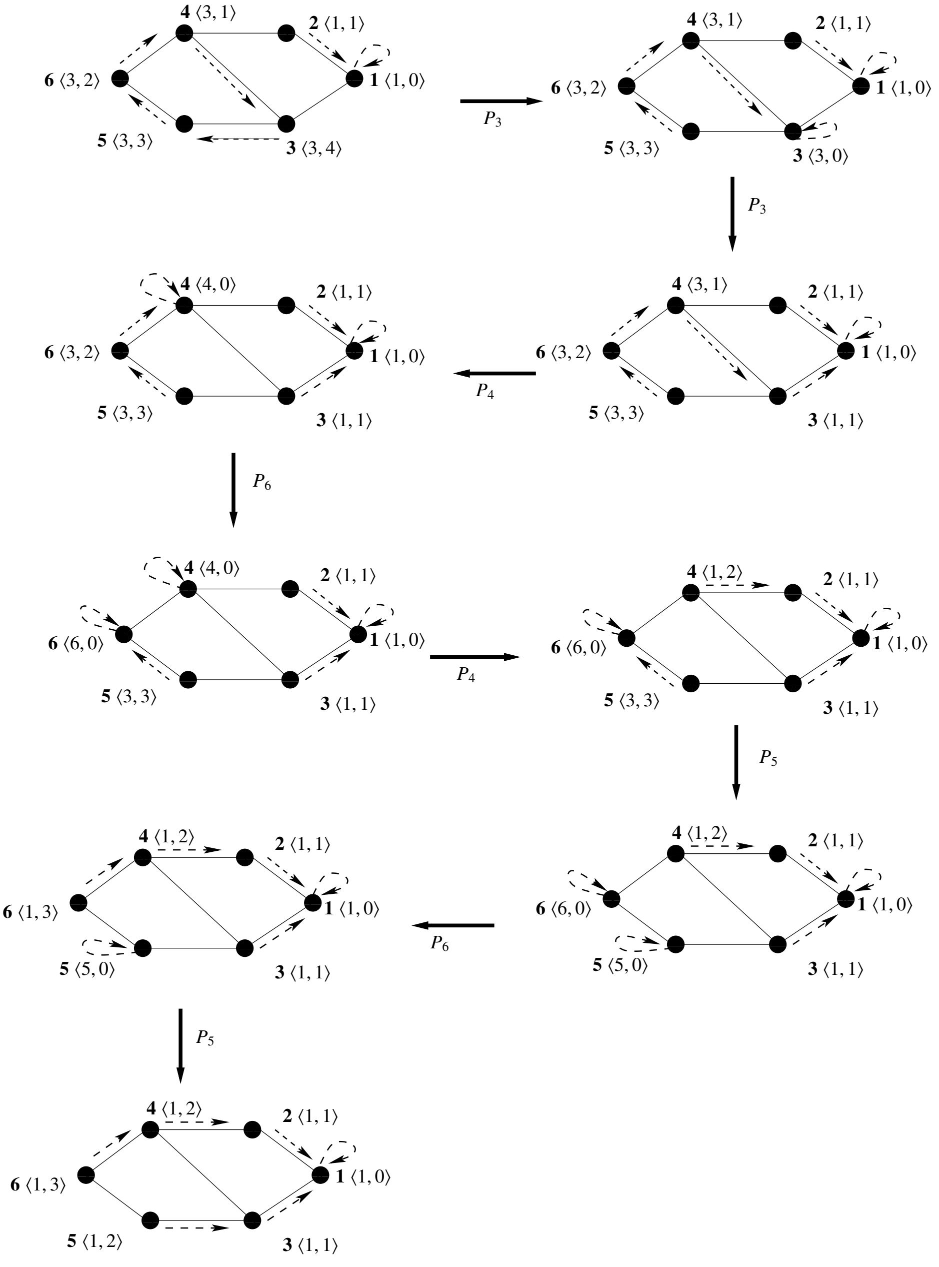

Fig. 5.3: Removal of Cycles by the Spanning Tree Algorithm of [94]

dictates the sequence in which its children get the token. Let $\lambda_0 = \langle \mathbf{r}_{0i}, \cdots, \mathbf{r}_{0m} \rangle$ be the ordering of the write communication registers of a process P_0 designated as the root. Process P_0 accesses its critical section if the value of $token_0$ is equal to the content of $\mathbf{r}_{m0}$, the write register of its last child P_m. The root process P_0 updates its token by incrementing it modulo $4n - 5$, where n is the total number of processes in the system, after it leaves its critical section. It passes on the token to other processes in the tree in the fashion similar to non-root processes.

```
**process** $P_0$
{
    **while**(*true*) **do**
        **if**($\text{token}_0 = \mathbf{r}_{m0}$) **do**
            ⟨*critical section*⟩
            $\mathbf{r}_{0i} := \text{token}_0 := (\text{token}_0 + 1 \ \mathbf{mod}\ (4 \cdot n - 5))$
        **od**
        **for**($x := i$ to $m$) **do**
            $\mathbf{r}_{0(x+1)} := \mathbf{r}_{x0}$
        **od**
    **od**
}
```

Fig. 5.4: Self-Stabilizing Mutual Exclusion Algorithm of [109] (Root Process)

```
**process** $P_i$ ($i \neq 0$)
{
    **while**(*true*) **do**
        **if**($\text{token}_i \neq \mathbf{r}_{\text{parent}_i i}$) **do**
            ⟨*critical section*⟩
            $\mathbf{r}_{iu} := \text{token}_i := \mathbf{r}_{\text{parent}_i i}$
        **od**
        **for**($x := u$ to $z$) **do**
            $\mathbf{r}_{i(x+1)} := \mathbf{r}_{xi}$
        **od**
    **od**
}
```

Fig. 5.5: Self-Stabilizing Mutual Exclusion Algorithm of [109] (Non-root Process)

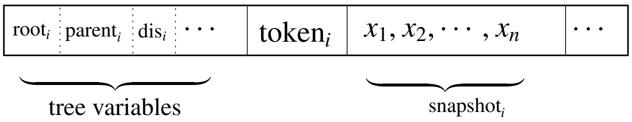

Fig. 5.6: Structure of Communication Register $\mathbf{r}_{ij}$ Used by the Token

Figure 5.7 shows the traversal path of the token in an example graph with seven processes. In addition to showing the parent-child relationship between adjacent processes, the dashed arrows mark the edges of the spanning tree; P_0 is designated as the root process. The traversal path of the token is drawn with the help of the gray arrowed curve. Note that the token traversal in the spanning tree follows *Euler tour* [110]. The token visits each tree edge twice in its traversal path. A leaf node is visited once whereas a non-leaf node is visited $c + 1$ times, where c denotes the number of children of the non-leaf node. Process P_0 passes the token first to process P_1, which, after accessing its critical section, passes the token further to its first child P_3. The token is returned back to P_1 by P_3 and is subsequently passed on to P_5. Process P_1 routes the token back to the root after it gets the token back from P_5. The token is then passed on to P_2 which, after accessing its critical section, passes onto its first child P_6. Process P_0 gets back the token after it is circulated in the subtree of P_2. Note that although each non-leaf process gets token multiple times, it accesses critical section only when its get the token from its parent.

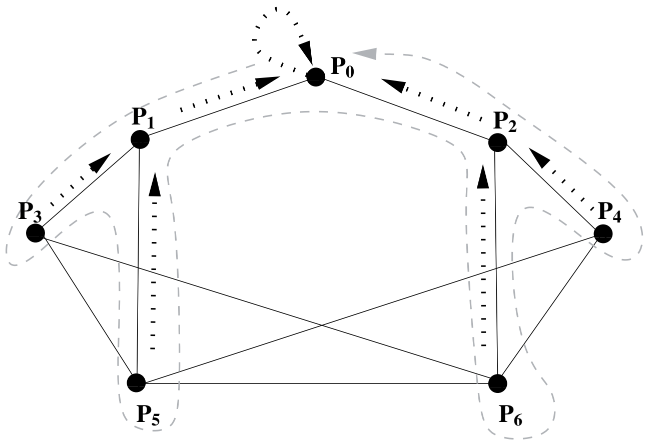

Fig. 5.7: Token Circulation in a Graph

Modification of Use Algorithm $\mathbb{A}$

The actions of use algorithm are "embedded" in the mutual exclusion algorithm. The guarded commands of the use algorithm $\mathbb{A}$ in a process P_i can be executed only if process P_i possesses the token. Thus, in essence, the algorithm $\mathbb{A}$ constitutes the "critical section" of the self-stabilizing mutual exclusion algorithm. The set of guarded commands of the use algorithm is also modified during the transformation. Figure 5.8 shows the transformed use algorithm (hereafter synonymously referred to as $\grave{\mathbb{A}}$).

```
process $P_i$
{
 localvar  $x_i$;
  macro $paint\_token \equiv \text{snapshot}_i.x_i^i := x_i$;
  macro $have\_token \equiv ((\text{token}_i \neq \text{token}_{\text{parent}_i}) \wedge (\text{parent}_i \neq i) \wedge (\text{root}_i \neq i))$
                    $\vee((\text{token}_i == \text{token}_{\text{rightchild}}) \wedge (\text{parent}_i == i) \wedge (\text{root}_i == i))$;
  if($have\_token$)
  $[]\grave{\mathcal{G}}_{i_j} :: \neg\mathcal{G}_{i_1} \wedge \cdots \wedge \mathcal{G}_{i_j} \wedge \cdots \wedge \neg\mathcal{G}_{i_n} \wedge decrease(\Delta_A) \wedge \neg\mathcal{P}_A \rightarrow \text{act}_{ij}; paint\_token;$   (1)
     ⋮
  $[]\grave{\mathcal{G}}_{i_p} :: \neg\mathcal{G}_{i_1} \wedge \cdots \wedge \mathcal{G}_{i_j} \wedge \cdots \wedge \neg\mathcal{G}_{i_n} \wedge \mathcal{P}_A \rightarrow \text{act}_{ij}; paint\_token;$   (2)
     ⋮
  $[]\grave{\mathcal{G}}_{i_q} :: \neg\mathcal{G}_{i_1} \wedge \cdots \wedge \mathcal{G}_{i_j} \wedge \cdots \wedge \neg\mathcal{G}_{i_n} \wedge \neg decrease(\Delta_A) \wedge \neg\mathcal{P}_A \rightarrow \textbf{skip}; paint\_token;$ (3)
     ⋮
  $[]\grave{\mathcal{G}}_{i_r} :: \neg\mathcal{G}_{i_1} \wedge \cdots \wedge \neg\mathcal{G}_{i_n} \rightarrow \textbf{skip}; paint\_token;$   (4)
  endif
}
```

Fig. 5.8: Modified Use Algorithm $\grave{\mathbb{A}}$

Although no new variables are added to the transformed algorithm, it accesses the variables belonging to the lower layers algorithms. In particular, $\grave{\mathbb{A}}$ includes variables *parent*$_i$, *root*$_i$ and, *token*$_i$ in macro *have_token*. Algorithm $\grave{\mathbb{A}}$ evaluates its modified guards only when *have_token* is true. As is evident from the name of the macro, *have_token* holds true when P_i possesses the token. The *paint_token* macro is used to update the global snapshot by appending its current local state. Note that we have used the method of [85] to ensure that each process has exactly one enabled guarded command in every global state. Algorithm $\grave{\mathbb{A}}$ consists of four classes of guarded commands. The guards of the transformed use algorithm are derived from the guards of use algorithm $\mathbb{A}$ in the following manner.

Guards of Type 1. A Type 1 guard $\grave{\mathcal{G}}_{i_j}$ is true only if 1) process P_i has the token, 2) the original guard $\mathcal{G}_{i_j}$ is enabled, 3) safety predicate $\mathcal{P}_A$ does not hold, and 4) the execution of guard $\mathcal{G}_{i_j}$ leads to a decrease in the value of ranking function $\varDelta_A$. The term *decrease*($\varDelta_A$) functions as a "look-ahead" operator of algorithm $\mathbb{A}$. It is obtained by computing the sign of difference between $\varDelta_A$ and $\varDelta_{A_{ij}}$. The value of $\varDelta_{A_{ij}}$ is obtained from ranking function $\varDelta_A$ by replacing variables in $\varDelta_A$ by their respective assignment expressions in act$_{ij}$. If guard $\grave{\mathcal{G}}_{i_j}$ is true, then process P_i executes the action of the original guard $\mathcal{G}_{i_j}$ and copies the new value of its local variable x_i to the global snapshot via the *paint_token* macro. The transformed algorithm has a guarded command of Type 1 corresponding to each guard of use algorithm $\mathbb{A}$, and therefore, every process P_i has m_i guarded commands of type 1.

Guards of Type 2. Each process P_i has m_i guarded commands of Type 2. A type 2 guards is true if 1) P_i holds the token, 2) safety predicate $\mathcal{P}_A$ holds, and 3) the corresponding guard $\mathcal{G}_{i_j}$ of use algorithm $\mathbb{A}$ is true. The assignment part of the original guarded command act$_{ij}$ is executed and snapshot is updated if guard $\grave{\mathcal{G}}_{i_j}$ is true.

Guards of Type 3. A guarded command of Type 3 is evaluated if P_i has the token. It is true if 1) the corresponding guard of $\mathbb{A}$ is true, 2) the safety predicate $\mathcal{P}_A$ does not hold and, 3) the assignment part of $\mathcal{G}_{i_j}$ does not lead to a decrease in the value of ranking function $\varDelta_A$. Process P_i takes a void step and writes its current local state to the global snapshot.

Guards of Type 4. Each process has one guarded command of Type 4 and it is enabled if none of the original guards are true. Process P_i copies its local variable to the global snapshot in case the Type 4 guard is true.

Each sub-algorithm $\grave{\mathcal{A}}_i$ of the transformed algorithm consists of $3 \cdot m_i + 1$ guarded commands. Actions on the assignment side of the guarded commands of $\mathbb{A}$ are unchanged in the transformed algorithm. Irrespective of the truth value of the modified guards, a process P_i writes the new value of its local variable in the snapshot before passing on the token.

5.3.2 Preservation of Self-Stabilization

We now show that the transformation of the use algorithm $\mathbb{A}$ preserves its self-stabilization property with respect to a predicate $\mathcal{P}_A$ under any weakly fair scheduler. Some definitions are in order before we proceed with the proof. In the following, the phrase "modified use algorithm $\grave{\mathbb{A}}$" refers to use algorithm with strengthened guards, and the phrase "transformed algorithm $\mathscr{T}(\mathbb{A})$" refers to the modified use algorithm along with the lower layer algorithms. The proof essentially consists of two parts: 1) in the first part it is shown that –within finite number of execution rounds– algorithm $\mathscr{T}(\mathbb{A})$ reaches a global state where the process possessing token has the correct global snapshot, 2) subsequently we prove that, following the global state where possession of the token implies the correct global snapshot, projection of an execution of algorithm $\mathscr{T}(\mathbb{A})$ over algorithm $\mathbb{A}$ is an execution of algorithm $\mathbb{A}$.

Definition 5.1 (Projection over Algorithm $\mathbb{A}$). *Let $\hat{\Xi} = \langle \cdots, \sigma_i, \cdots, \sigma_j, \cdots \rangle$ be a maximal execution of a transformed algorithm $\mathscr{T}(\mathbb{A})$. The projection $\tilde{\Xi}_{\mathbb{A}}$ of a maximal execution of $\mathscr{T}(\mathbb{A})$ over* *use algorithm $\mathbb{A}$ is obtained by removing the variables belonging to algorithm $\mathbb{A}$ from every global state σ_i appearing in $\hat{\Xi}$ in which algorithm $\mathbb{A}$ executed an enabled guarded command, that is, $\tilde{\Xi}_{\mathbb{A}} := \langle \cdots, \sigma_{i|var(\mathbb{A})}, \sigma_{j|var(\mathbb{A})}, \cdots \rangle$ such that an enabled guarded command of $\mathbb{A}$ is executed in σ_i and σ_j.*

Definition 5.2 (Projection over Process P_i). *The projection $\tilde{\Xi}_{P_i}$ of a maximal execution $\hat{\Xi}$ of a transformed algorithm $\mathscr{T}(\mathbb{A})$ over a process P_i is obtained by removing all the variables except the variables belonging to process P_i from every state σ_i appearing in $\hat{\Xi}$.*

Definition 5.3 (Correct Global Snapshot). *Let $\sigma_{ix|var(A)}$ be the projection of the local state of process P_x on the use algorithm $\mathbb{A}$ obtained by removing variables not belonging to the use algorithm $\mathbb{A}$ from σ_{ix}. The local copy of the global snapshot at process P_y, obtained by inspecting the token, is termed as* correct local global snapshot *if it contains the current values of $\sigma_{ix|var(\mathbb{A})}$ for all $P_x \in \Pi \setminus \{P_y\}$.*

Definition 5.4 (1-Step Consistent Global Snapshot). *Let $\tilde{\Xi}_{\mathbb{A}} = \langle \sigma_{1|var(\mathbb{A})}, \sigma_{2|var(\mathbb{A})}, \cdots \rangle$ be the projection of maximal execution of a transformed algorithm $\mathscr{T}(\mathbb{A})$ over the use algorithm $\mathbb{A}$. The local copy of global snapshot at a process P_y is said to be 1-step consistent if for every process $P_x \in \Pi \setminus \{P_y\}$ the local copy of the local state of P_x is $\sigma_{kx|var(\mathbb{A})}$ such that σ_k is the global system state in which the process P_x executed an action of the use algorithm $\mathbb{A}$ and σ_i is the global system state where P_y executed an action of $\mathbb{A}$ most recently and k is the largest index satisfying the condition $i > k$.*

The notion of 1-step consistent global snapshot captures the scenarios where a process' copy of the global snapshot might be "outdated" because of an execution step of some other process in the system. We have argued earlier that the correctness of a global snapshot is critical for the correctness of the transformation. 1-Step consistency of a global snapshot, *prima facie*, appears to be counter-productive in the light of this argument. Nonetheless, a 1-step consistent snapshot forms part of the legal global system state of the transformed algorithm $\mathscr{T}(\mathbb{A})$. This is due to the application of the mutual exclusion algorithm to coordinate the execution of the guarded commands of the modified use algorithm $\grave{\mathbb{A}}$. Since correct global information is sent around with the token, a process' copy of snapshot can become outdated after it passes on the token. However –as we prove later– in any legal state of the transformed algorithm $\mathscr{T}(\mathbb{A})$, the difference between a process' copy of snapshot and the current global state is restricted to a single execution step of the modified use algorithm by any other process. In this sense the notion of 1-step consistency of global snapshot as defined above is weaker than the notion of snapshot correctness.

Let global state σ_{span} be the global system state where predicate $\mathcal{P}_{\mathrm{span}_i}$ holds at every process $P_i \in \Pi$.

Theorem 5.1 (based on [94]). *Irrespective of the initial state, the transformed algorithm $\mathscr{T}(\mathbb{A})$ reaches the state σ_{span} within a bounded number of execution steps.*

Although the mutual exclusion algorithm and modified use algorithm $\grave{\mathbb{A}}$ read the variables written by the spanning tree algorithm, the upper layer algorithms do not interfere with the execution of the spanning tree algorithm. Intra-process fairness guarantees that the actions of the spanning tree algorithm are executed infinitely often. As explained earlier, each process checks the consistency of its immediate neighborhood and, eventually, false identifiers are purged out of the system. A tree spans the system when each process satisfies $\mathcal{P}_{\mathrm{span}_i}$. The process selected as the root after the spanning tree algorithm has stabilized is denoted as P_{root} in the sequel.

Let $\mathcal{P}_{\mathrm{mutex}}$ be a predicate which holds true if only one process in the system can access its critical section and let σ_{mutex} be a global state satisfying predicate $\mathcal{P}_{\mathrm{mutex}}$.

Theorem 5.2 (based on [109]). *Every execution of the transformed algorithm $\mathscr{T}(\mathbb{A})$ reaches the state σ_{mutex} within $O(n^2)$ rounds of reaching state σ_{span}.*

Note that token traversal in a spanning tree of n processes is similar to token traversal in a ring of $2 \cdot n - 2$ processes. Also, recall that $\mathcal{P}_{\text{mutex}}$ holds in a system if all the token variables in the system have equal value or exactly one pair of non-root neighboring processes have different token values. There is always at least one process with a token in any state of the mutual exclusion algorithm. Assume that a system starts in a state where multiple processes can access their respective critical section. Any action of the mutual exclusion algorithm in this scenario does not increase the number of process which can access their critical sections because, the process that uses privilege to access its critical section loses the token in the following state. Eventually, the token value of every process changes and, thus, the token value of the process P_{root} gets incremented. Since the system has only n process, there exists a number $\tilde{n}, \tilde{n} < 4 \cdot n - 4$, such that no token is equal to $\tilde{n}$. Process P_{root} eventually assigns $\tilde{n}$ to its token variable because P_{root} gets incremented at least once in $4 \cdot n - 4$ rounds. The convergence of the mutual exclusion algorithm follows from the observation that P_{root} does not get token until all other processes assign $\tilde{n}$ to their tokens.

As a result of stabilization of spanning tree and mutual exclusion layers (Theorem 5.2) one of the processes in the system assumes role of the root process and coordinates the token circulation following the state σ_{mutex}. Let σ_{mutroot} be the global state in which the process P_{root} gets the token for the first time following the state σ_{mutex}.

Lemma 5.1. *In any execution of the transformed algorithm $\mathscr{T}(A)$, the root process P_{root} has the correct local global snapshot within $O(n)$ rounds of reaching the state $\sigma_{mutroot}$.*

Proof. The mutual exclusion algorithm let the token circulate on the spanning tree and the token traverses the tree in depth-first manner. It defines a Euler tour, thus a virtual ring, over the spanning tree and the virtual ring has $2n - 2$ virtual nodes.

The root process P_{root} gets the token and thereby chance to access its critical section at least once every $4n - 4$ rounds irrespective of the fact whether the mutual exclusion algorithm has stabilized or not.

The global snapshot that P_{root} gets along with token in σ_{mutroot} might be incorrect and thus P_{root} may execute a guarded command that is enabled with the help of incorrect global snapshot. However, the assignment part of each of the guarded commands writes the current values of the local variables belonging to use algorithm $\mathbb{A}$ to the global snapshot part of the token. Let σ_j be the global state after P_{root} executes its guarded command. Thus, when P_{root} passes on the token to its first child (defined by the ordering of outgoing edges in P_{root}) the global snapshot has the current local state $\sigma_{j_{\text{root}}|\text{var}(A)}$ of P_{root}.

Let σ_k be a global state after σ_j such that a non-root process P_z gets the token from its parent process. It can be observed that – irrespective of the correctness of the global snapshot which is passed on to process P_z along with the token – the token has correct local state $\sigma_{l_z|\text{var}(A)}$ of P_z, where σ_l is the resultant state after the execution of guarded commands of use algorithm $\mathbb{A}$ in which token is passed on to the descendants of P_z. A process accesses its critical section (and thus executes algorithm $\mathbb{A}$) only when it gets the token from its parent process although it gets token more than once while routing it through its sub-tree and back to its parent. This implies that once any process passes on the token to its descendants, projection of its local state on use algorithm does not change. Let $P_{\text{root}} \rightarrow P_x \rightarrow P_y \rightarrow P_z$ be the path traversed by the token in the spanning tree before it reaches process P_z. The argument above can be used to infer that process P_z gets the current values of local variables of processes P_{root}, P_x and P_z (belonging to algorithm $\mathbb{A}$) when it receives the token from its parent. This argument can be further extended to infer that every process gets the current values of variables belonging to algorithm $\mathbb{A}$ of processes that possessed the token before and appends its local state prior to passing the token to its successors.

Every process gets a chance to access its critical section once in $4n - 4$ rounds after the mutual exclusion algorithm has stabilized. Let σ_m be the global state in which process P_{root} gets the token for the first time after σ_{mutroot}. All the other processes append their current local states to the global snapshot between σ_{mutroot} and σ_m and do not change them thereafter. Thus, when P_{root} gets the token in σ_m, it has the current value of every $\sigma_{mx|\text{var}(A)}$ $(\forall x \in \{1, \cdots, n\} \setminus \{\text{root}\})$. □

Figures 5.9 through 5.13 illustrate the state sequence of an example system while the root process gathers the correct global snapshot. The series of figures show a partial view of a system. The thick edges represent the edges which form the spanning tree. Process P_0 acts as the root of the spanning tree. Processes P_3 and P_5 have subtrees which are omitted in the state sequence. The dashed arrows depicts the direction of token traversal. An asterisk symbol next to a process ID indicates that the process has the token and thereby the privilege to execute its critical section. The status of the local copy of a process' snapshot is shown with help of a label at each node in the graph. The label "BAD" indicates that the local copy of the snapshot contains incorrect values; a process is labelled with "UPDATE" if it writes it current local state to the snapshot. The sequence starts in a state where P_0 (the root) gets the token exclusively for the first time. It writes the current value of its local state to the snapshot part of the token and sends the token to its first child P_1 (Figure 5.9).

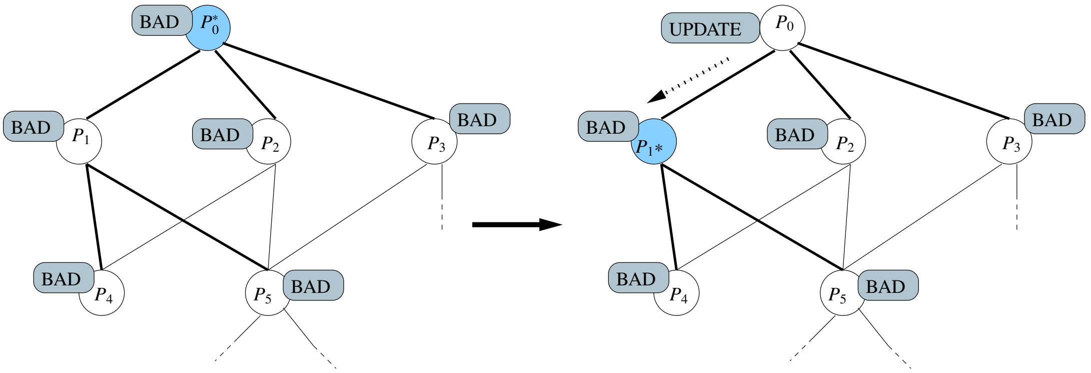

Fig. 5.9: Snapshot Sequence: Process P_0 Updates Snapshot Token

Process P_1 updates the snapshot compartment by writing its current local state before passing on the token to its child P_4 (Figure 5.10). The token is next routed to process P_5 and, the snapshot part of the token contains the latest local states of process P_0, P_1 and, P_4. Process P_5 use the privilege to update the snapshot.

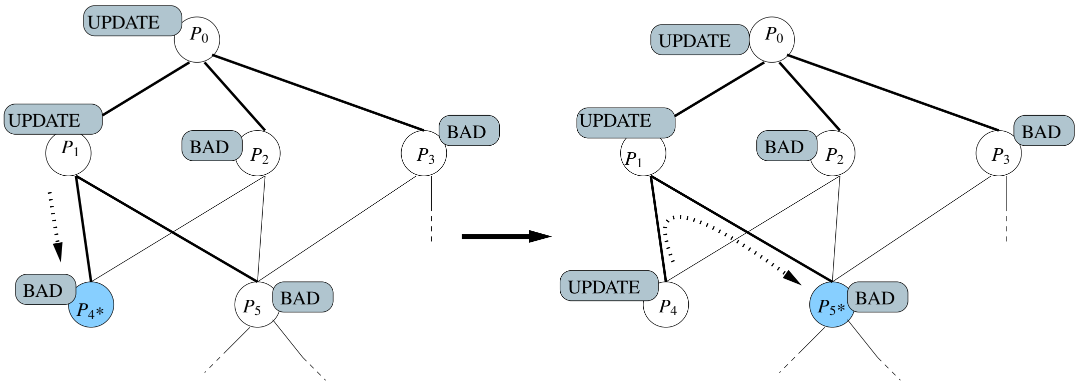

Fig. 5.10: Snapshot Sequence: Processes P_1 and P_4 Update Snapshot

Process P_5 passes the token to its descendants in its subtree (see Figure 5.11). A process in the subtree of P_5 writes to the snapshot when it gets the token and passes it to its children. In this process, the snapshot part collects the current state of all the processes in the subtree of P_5. P_5 –after circulating the token in its subtree– gets it back and passes on to its parent; during these execution steps processes in the subtree get the latest local states of all the processes which possessed the token before them.

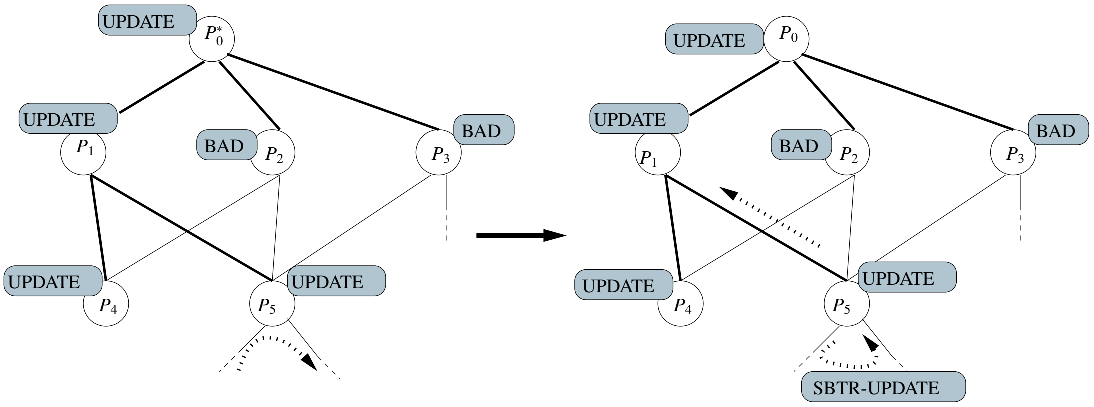

Fig. 5.11: Snapshot Sequence: The Token Circulates in the Subtree of Process P_5

Process P_1 returns the token with updated snapshot to the root process P_0 (Figure 5.12). Since all the processes in the subtree of P_1 have their token value equal to that of P_0, the root process passes on the token to P_2. Process P_2 returns the token back to the root process after updating its state and the snapshot because it does not have any children in the spanning tree. Process P_0 next sends the token to P_3; Process P_3 executes the modified use algorithm and updates snapshot with the latest value of local variables of $\grave{\mathbb{A}}$. Despite possessing the token multiple times, the variables belonging to $\grave{\mathbb{A}}$ are changed only once by the processes which possessed token prior to process P_3. Although the snapshot has the current local states of the processes P_0 through P_2, the snapshot received might not be correct. This is because the snapshot may not contain the current local states of the descendants of process P_3.

Process P_3 sends the token to its descendants and the token is circulated in the subtree (Figure 5.13). Each process executes guarded commands of modified use algorithm and writes the new local state to the snapshot before passing token to its successor. The token is returned by process P_3 after it gets it back from its last child. The root process P_0 gets the token along with the snapshot that contains the current values of the local variables of use algorithm. The root process has the correct snapshot in last configuration of Figure 5.13 because no process executes an action of À after it has received and updated the token obtained from its token.

Let σ_{corsnp} be the global state in which process P_{root} gets the token with the current global snapshot for first time in an execution. Although the root process has the correct snapshot in state σ_{corsnp}, local copies of the snapshot in the other processes in the system might be – and generally are– inconsistent.

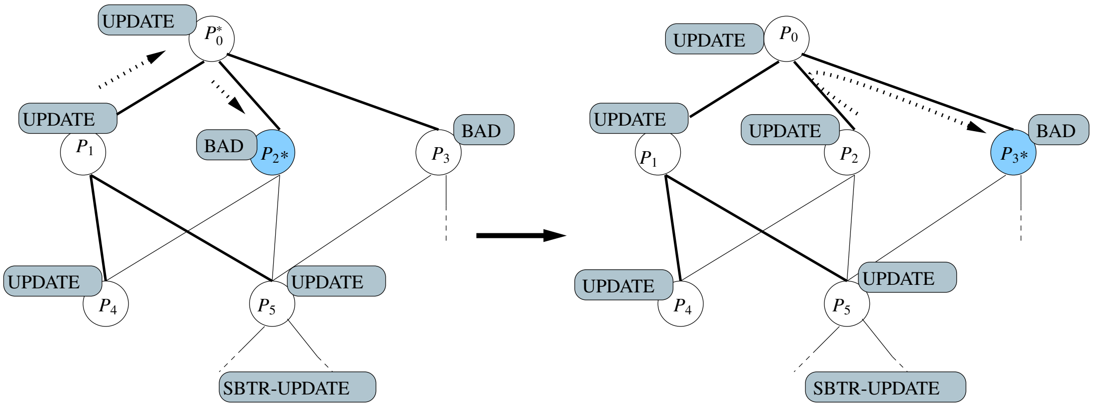

Fig. 5.12: Snapshot Sequence: Token Circulation Among Children of Process P_0

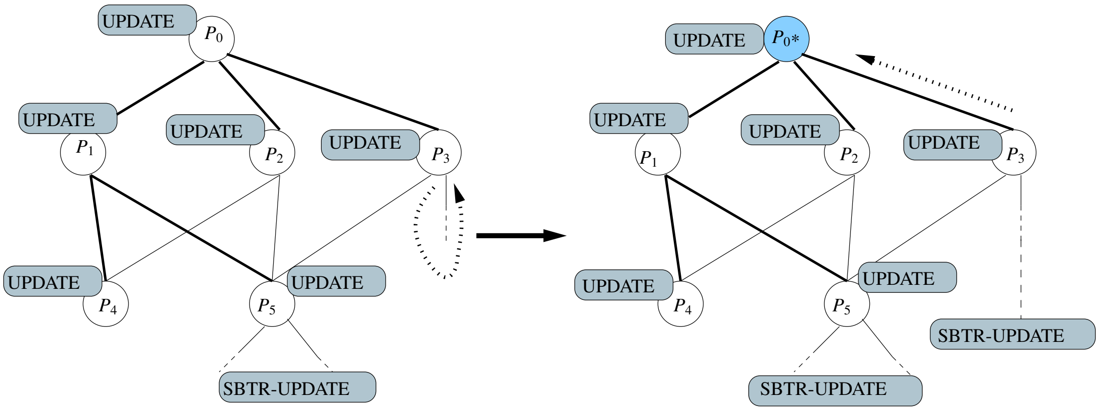

Fig. 5.13: Snapshot Sequence: Token Circulation in the Subtree of Process P_3

The difference between the local copy of the snapshot of a non-root process and the current global state cannot be quantified in σ_{corsnp} because a non-root process gets only a partially updated snapshot in any state between σ_{mutex} and σ_{corsnp}. Lemma below shows that the difference, however, remains bounded following the state σ_{corsnp}.

Lemma 5.2. *In any execution of the transformed algorithm $\mathscr{T}(\mathbb{A})$, every process P_i in the system has at least a 1-step consistent global snapshot within $O(n)$ rounds of reaching the global state σ_{corsnp}.*

Proof. Every process gets a chance to execute its critical section once in $4n - 4$ rounds after the mutual exclusion algorithm stabilizes. Also the root process P_{root} gets the correct global snapshot when it gets the token exclusively for second time in state σ_{corsnp} (Lemma 5.1). Let σ_i be the global state in which process P_{root} has the correct global snapshot. Irrespective of the truth value of the guards of the modified use algorithm $\grave{\mathbb{A}}$, P_{root}, appends its current local state $\sigma_{j_{\mathrm{root}}|\mathrm{var}(\mathbb{A})}$ to the token before passing it on. Let process P_α be the first child of process P_{root} in the spanning tree. Process P_α gets the token right after P_{root}. As a result of token possession, process P_α might change its local state. Let σ_α be the resultant global state. This makes the P_{root} copy of local state of P_α outdated.

Let $\mathit{\Xi}_{|\sigma_{mutex}}$ be a suffix of a maximal execution $\hat{\mathit{\Xi}}$ of the transformed algorithm $\mathscr{T}(\mathbb{A})$ starting in the state σ_{mutex}. The maximal execution $\mathit{\Xi}_{|\sigma_{mutex}}$ cannot have two global states σ_κ and σ_j such that P_α accesses its critical section in σ_κ and σ_j and, the root process P_{root} does not get token in any state between σ_κ and σ_j. This is because process P_α can access its critical only when $token_\alpha$ is not equal to $token_{\mathrm{root}}$ and P_α sets $token_\alpha$ equal to $token_{\mathrm{root}}$ when it accesses its critical section. Thus, in the global state σ_α, P_{root}'s copy of the local state of P_α has the value which corresponds to the global state σ_ι –where σ_ι is the state which resulted from the execution of $\grave{\mathscr{A}}_\alpha$ at P_α– and ι is the largest index such that $\iota < i$.

Let $P_{\mathrm{root}} \to P_\alpha \cdots \to P_\varsigma \to P_\beta$ be the sequence in which the token traverses the spanning tree after state σ_i. Let σ_β be the global state resulting from execution of a critical section of process P_β. As a result of accessing its critical section (thereby possibly executing a guarded command of the modified algorithm $\grave{\mathbb{A}}$), process P_β might change its local state. This action will make the copies of local state of P_β outdated in process that possessed token before P_β. However, as we argued above there cannot be a suffix of maximal execution of algorithm $\mathscr{T}(\mathbb{A})$ with two states σ_ϖ and σ_β where P_β accessed its critical section and none of the processes in the set $\{P_{\mathrm{root}}, \cdots, P_\varsigma\}$ accesses its respective critical section. Thus, each process in $\{P_{\mathrm{root}}, \cdots, P_\varsigma\}$ would have $\sigma_{\varpi\beta|\mathrm{var}(A)}$ as local state of P_β where ϖ is the largest index such that $\varpi < \iota$ for each $\iota \in \{i, \alpha, \cdots, \varsigma\}$. This argument can be extended inductively for all the non-root processes. Hence, it can be inferred that once process P_{root} gets the token again after the state σ_i , all the processes in the system have a 1-step consistent global snapshot. □

The state sequence in Figures 5.14 through 5.18 shows how the local copies of the global snapshot become 1-step consistent after the example system reaches state σ_{corsnp}. A process with correct snapshot is labeled as "C-GSNAP" and "1S-GSNAP" indicates that the process has a 1-step consistent snapshot. The root process P_0 has the correct global snapshot and the token in the first configuration of Figure 5.14. Process P_0 executes a guarded command of the modified use algorithm; this is a "correct" transition since P_0 has the correct global snapshot. The root process sends the token to its first child P_1 after writing its current local state to the snapshot compartment. Process P_1 gets the token with the latest local states of all the non-root processes and the current state of process P_0 –subsequent to potential execution of the guarded commands of $\grave{\mathbb{A}}$– appended to it. Process P_1 executes algorithm $\grave{\mathbb{A}}$ based on the global snapshot it received with the token. The local copy of snapshot at P_0 becomes 1-step consistent snapshot as a result of this. Process P_1 adds the latest value of the local variables of use algorithm $\mathbb{A}$ to the global snapshot compartment and passes the token to its child P_4 (see Figure 5.15). Process P_4 executes $\grave{\mathscr{A}}_4$ and returns the token to process P_1 after updating the snapshot. Process P_5 gets token from P_1 with current local states of all the processes appended to it. Process P_5 executes its critical section based on global snapshot it received and passes on the token to the first child in the subtree. Note that local states (of the use algorithm $\mathbb{A}$) of all other processes in the system have not changed in the meantime;

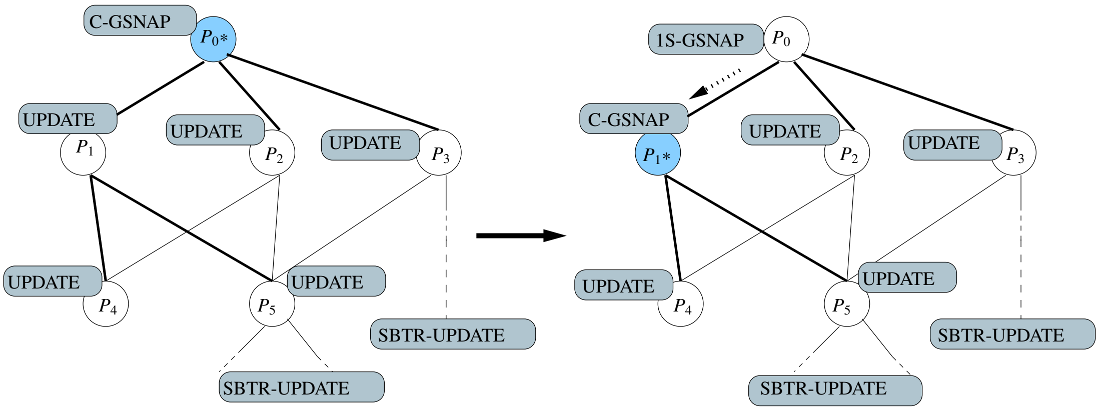

Fig. 5.14: Consistent Snapshot Sequence: P_0 Gets Outdated

the local copies of the snapshot of the processes which possessed the token prior to process P_5 become 1-step consistent due to the actions of P_5. The processes in the subtree of process P_5 return the token

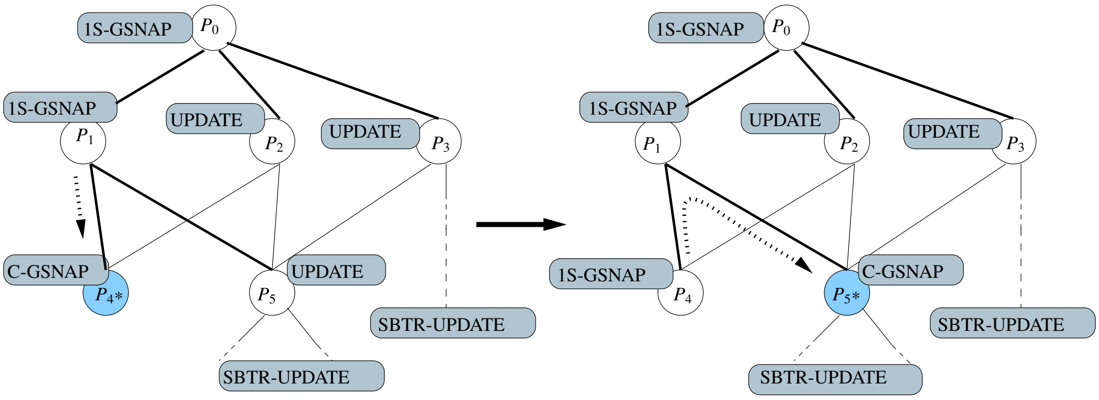

Fig. 5.15: Snapshot Sequence: Processes P_4 and P_5 Get Correct Snapshots

back to P_5 after appending their latest local states to it (Figure 5.16). The token is passed on to P_1 with the updated local states of P_5 and the processes in the subtree. Except the local copy of the rightmost descendant of P_5, all copies of the snapshots in the left subtree of the root process are 1-step consistent in the second configuration of Figure 5.16. The root process routes the token to its second child P_2 (see Figure 5.17). Process P_2 returns the token to P_0 after executing $\grave{\mathcal{A}}_2$. The token is further passed on to process P_3. It executes any enabled guarded command of the modified use algorithm, updates the snapshot and, passes the token to its first child. It should be observed that in this suffix of execution all the actions of $\grave{\mathcal{A}}$ are executed with the help of the correct snapshot. Additionally, the local copy of the snapshots in the left subtree of P_0 remain 1-step consistent despite the actions of P_2 because, P_2 executed $\grave{\mathcal{A}}$ only once between the states σ_{mutroot} and σ_{corsnp}. Each process in the subtree of P_3 executes the modified used algorithm using correct global snapshot (Figure 5.18). The change in the local state of any process in the subtree renders the snapshot copies of its predecessors 1-step consistent. Process P_3 sends the token back to the root process P_0 after every process in the subtree executes its critical

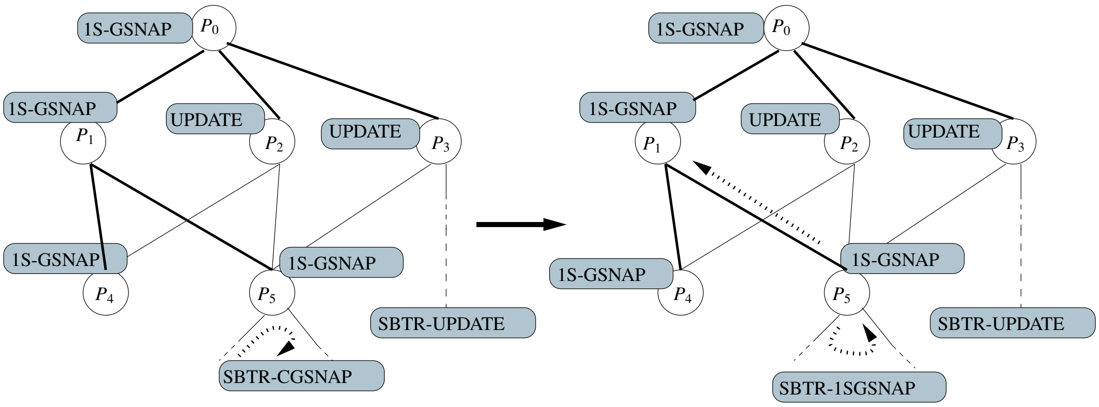

Fig. 5.16: Snapshot Sequence: The Subtree of Process P_5 Gets Correct Snapshots

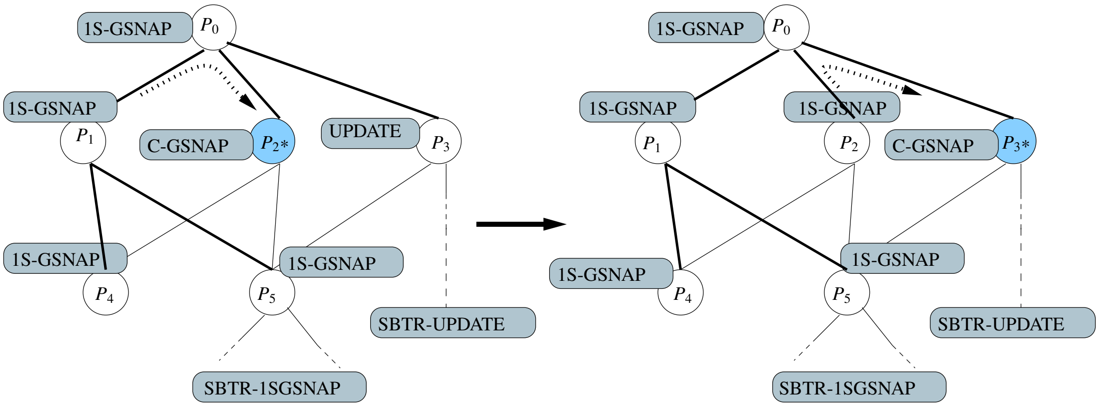

Fig. 5.17: Snapshot Sequence: Children of Process P_0 Get Correct Snapshot

section. Consequently, the root process' copy of snapshot becomes the correct global snapshot. Note that arrival of token at every process renders its snapshot globally correct.

Lemma 5.3. *A process executes an action of the modified use algorithm $\grave{\mathbb{A}}$ in any global state following σ_{corsnp} if the process has a correct global snapshot.*

Proof. Process P_{root} also has the token in the state σ_{corsnp} (from Lemma 5.1) which allows P_{root} to execute an enabled guarded command of the modified use algorithm $\grave{\mathbb{A}}$ (by construction). Thus, process P_{root} execute an action of $\grave{\mathbb{A}}$ only if it has a correct global snapshot. Let P_β be a process that gets the token in some state after state σ_i. Process P_β gets the correct snapshot when it gets the token (from Lemmata 5.1 and 5.2). Possession of the token also enables P_β to execute an enabled guarded command of $\grave{\mathbb{A}}$ (by construction). Thus, any non-root process executes an action of $\grave{\mathbb{A}}$ only if it has correct global snapshot. □

Lemma 5.4. *If an action of the modified use algorithm $\grave{\mathbb{A}}$ is executed by any process in a global state following σ_{corsnp}, then the projection of this execution step of the transformed algorithm $\mathscr{T}(\mathbb{A})$ over algorithm $\mathbb{A}$ leads to a decrease in the value of ranking function Δ_A provided $\neg\mathcal{P}_A$ holds.*

Proof. A process P_i executes a guarded command of the modified use algorithm $\grave{\mathbb{A}}$ only if it has the token (by construction). Also, an action of $\grave{\mathbb{A}}$ is executed only if process P_i has a correct global snapshot

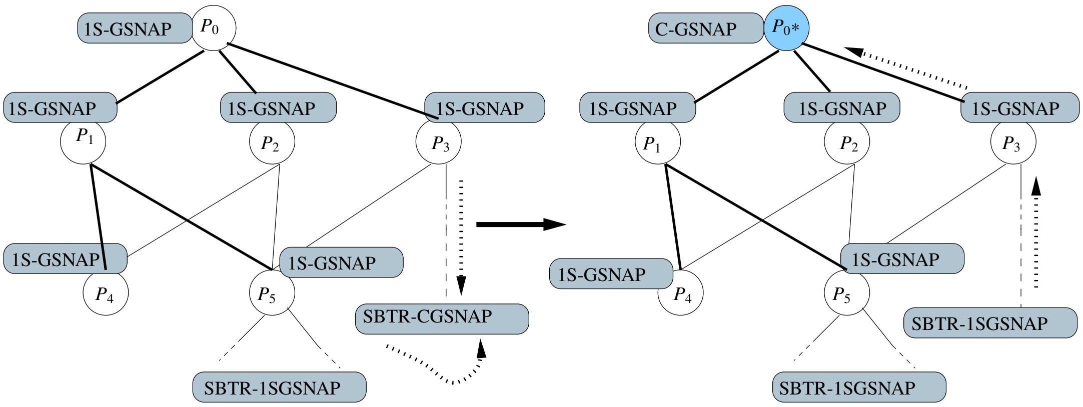

Fig. 5.18: Snapshot Sequence: The System Reaches 1-Step Consistent State

(from Lemma 5.1). The guards in every process are strengthened such that an assignment statement is executed only if its execution leads to a decrease in the value of Δ_A. This in conjunction with Lemma 5.3 completes the proof. □

Let $\Xi_{|\sigma\text{corsnp}}$ be the suffix of a maximal execution of the transformed algorithm $\mathscr{T}(\mathbb{A})$ under a weakly-fair scheduler such that σ_{corsnp} is the first state of $\Xi_{|\sigma\text{corsnp}}$.

Lemma 5.5. *The projection of the execution suffix $\Xi_{|\sigma corsnp}$ over use algorithm $\mathbb{A}$ is an execution of algorithm $\mathbb{A}$ under the scheduler $\mathbb{D}_A$.*

Proof. Algorithm $\mathbb{A}$ has the liveness property under the scheduler $\mathbb{D}_A$ which is proven by showing the existence of the ranking function Δ_A. This implies that 1) in every state $\sigma_{i\mathbb{A}}$ of $\mathbb{A}$ there exists at least one process with an enabled guarded command $\mathcal{G}_{ij}$ and 2) in every state at least one of the processes with enabled guarded commands has a guarded command $\mathcal{G}_{ij}$ enabled such that decreaes(Δ_A) holds true until predicate $\mathcal{P}_A$ is satisfied. Thus, in every state of $\mathbb{A}$ there exists at least one process with a modified guarded command $\grave{\mathcal{G}}_{ij}$ until $\mathcal{P}_A$ holds.

Let $\tilde{\Xi}_{A|\text{corsnp}_1}$ be the projection of the suffix $\Xi_{|\sigma\text{corsnp}}$ over use algorithm $\mathbb{A}$. Let ε be a slice of the projection $\tilde{\Xi}_{A|\text{corsnp}_1}$ of the suffix of a maximal execution Ξ of $\mathscr{T}(\mathbb{A})$ under a weakly fair scheduler such that 1) its first state is the state where the root process gets the correct global snapshot for the first time (σ_{corsnp}) and, 2) $\mathcal{P}_A$ does not hold in any state. As argued above, at least one process in the system has an enabled guarded command of the modified use algorithm $\grave{\mathbb{A}}$ in the first state of ε.

Let $\sigma_{i|\text{var}(A)} \rightarrow \sigma_{j|\text{var}(A)}$ be an execution step of $\grave{\mathbb{A}}$ in ε. There can be no execution –and therefore no transition of variables– of $\grave{\mathbb{A}}$ in a process unless it has the token (by construction). Only one process can execute a guarded command of $\grave{\mathbb{A}}$ after the mutual exclusion algorithm has stabilized (Theorem 5.2). Thus, $\sigma_{i|\text{var}(A)} \rightarrow \sigma_{j|\text{var}(A)}$ can only be brought about by execution of a guarded command of algorithm $\mathbb{A}$ by a single process in the system.

Let P_x be the process which executes the guarded command to bring about $\sigma_{i|\text{var}(A)} \rightarrow \sigma_{j|\text{var}(A)}$. Process P_x has the token in state σ_i. Process P_x executes an enabled modified guarded command of the modified used algorithm $\grave{\mathbb{A}}$ in $\sigma_{i|\text{var}(A)}$ based on latest global information (Lemma 5.3) and this step leads to a decrease in the value of ranking function Δ_A (Lemma 5.4). We argued above that use algorithm $\mathbb{A}$ has an enabled guarded command that leads to a decrease in the value of Δ_A in state $\sigma_{i\mathbb{A}}$ under scheduler $\mathbb{D}_A$ if $\sigma_{i\mathbb{A}}$ does not satisfy $\mathcal{P}_A$. It should also be observed that the assignment statements of $\mathbb{A}$ are unchanged during the transformation to $\mathscr{T}(\mathbb{A})$. Thus, there exists an execution step of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ which corresponds to $\sigma_{i|\text{var}(A)} \rightarrow \sigma_{j|\text{var}(A)}$. An execution step of algorithm $\mathbb{A}$

in state $\sigma_{i|\text{var}(A)}$ is possible only in process P_x. Note that, P_x cannot be denied the token indefinitely because the mutual exclusion layer algorithm ensures that each process gets the token infinitely often in any execution under any weakly-fair scheduler. Additionally, the execution of an enabled guarded command of algorithm $\grave{\mathbb{A}}$ in process P_x in state $\sigma_{i|\text{var}(A)}$ can be delayed by a scheduler if it does not select process P_x once it possesses the token. This implies that a continuously enabled process is not activated at all, since the guards pertaining to algorithm $\grave{\mathbb{A}}$ are enabled in P_x once it has the token. However, this implies that a continuously enabled process is never activated. Hence, an execution step of algorithm $\grave{\mathbb{A}}$ in process P_x cannot be delayed indefinitely. This argument can be extended to build an equivalent execution of algorithm A under state $\mathbb{D}_A$ which corresponds to $\tilde{\Xi}_{A|\text{corsnp}_1}$ until predicate $\mathcal{P}_A$ holds.

Let $\tilde{\Xi}_{A|\text{corsnp}_2}$ be projection of the suffix of a maximal execution of $\mathscr{T}(A)$ such that $\tilde{\Xi}_{A|\text{corsnp}_2}$ is not maximal. Let $\varepsilon_{\mathcal{P}_A}$ be a suffix of $\tilde{\Xi}_{A|\text{corsnp}_2}$ such that all states satisfy predicate $\mathcal{P}_A$. Thus, $\varepsilon_{\mathcal{P}_A}$ consists of states where a guarded command in a process is enabled but is never executed. This is, however, not possible because the mutual exclusion algorithm ensures that each process gets the token infinitely often in any execution. A process executes an enabled guarded command of algorithm $\mathbb{A}$ based on latest global state (Lemma 5.3) when it gets token. The result of such an execution is same as that of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ as the assignment parts of guarded commands are unchanged. □

Lemma 5.6. *If use algorithm $\mathbb{A}$ converges to a predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$ then the transformed algorithm $\mathscr{T}(\mathbb{A})$ converges to the predicate $\mathcal{P}_A$ under any weakly-fair scheduler.*

Proof. Let $\tilde{\Xi}_A$ be the projection of a maximal execution $\hat{\Xi}_{\mathscr{T}(\mathbb{A})}$ of the transformed algorithm $\mathscr{T}(\mathbb{A})$ over use algorithm $\mathbb{A}$. Let ε be a suffix of $\tilde{\Xi}_A$ such that σ_{corsnp} is the first state of ε (Lemma 5.1). ε is a maximal execution of $\mathbb{A}$ under scheduler $\mathbb{D}_A$ (Lemma 5.5). Let ε have no suffix that converges to a state satisfying predicate $\mathcal{P}_A$. This implies that there exists a maximal execution of use algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ which does not converge to a state satisfying the predicate $\mathcal{P}_A$. This, however, contradicts the precondition of the lemma statement. Also, since Δ_A is a monotonous function defined over a well-founded domain, $\mathscr{T}(\mathbb{A})$ reaches a state satisfying $\mathcal{P}_A$ in a finite number of execution steps. This completes the proof. □

Remark 5.2. The ranking function Δ_A embedded in the guards of modified use algorithm $\grave{\mathbb{A}}$ must be valid for all the executions under scheduler $\mathbb{D}_A$. A "mismatch" between a ranking function and a scheduler may lead to the transformed algorithm that generates executions extraneous to the executions under the original scheduler. More specifically, should ranking function Δ_A correspond to a scheduler less restrictive than scheduler $\mathbb{D}_A$, transformed algorithm $\mathscr{T}(\mathbb{A})$ may produce executions which cannot be produced under $\mathbb{D}_A$.

Theorem 5.3. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under any weakly-fair scheduler.*

Proof. Convergence of the transformed algorithm $\mathscr{T}(\mathbb{A})$ follows from Lemma 5.6. Closure follows from Lemma 5.5 and the assumption that the safety predicate $\mathcal{P}_A$ is closed under a weakly-fair scheduler. □

Corollary 5.1. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ is self-stabilizing with respect to the predicate $\mathcal{P}_{span} \wedge \mathcal{P}_{mutex} \wedge \mathcal{P}_A$ under any weakly fair scheduler.*

5.3.3 Concurrency Optimization

As a result of using the global mutual exclusion algorithm for the coordination among constituent processes, only one process is able to execute an action of the (modified) use algorithm in any system state. Although this restriction plays a pivotal role in preserving the self-stabilization property of use algorithm, it leads to an avoidable decrease of concurrency in certain scenarios. The loss of concurrency is particularly evident when a use algorithm, whose ranking function can be evaluated at any process using local states of processes belonging to k-neighborhood, is transformed.

For example, consider the algorithm shown in Figure 5.19. Algorithm $\mathbb{SSWMAC}$ loosely imitates Medium Access Control (MAC) algorithms meant for wireless sensor networks. Each process in the system tries to find a unique transmitting slot in a communication frame. A process also tries to ensure that the slot is positioned as close as possible to the beginning of a communication frame. Algorithm $\mathbb{SSWMAC}$ uses variables turn$_i$ and slot$_i$ to determine a slot for each process in a communication frame. Variable turn$_i$ can be assigned any positive integer value; slot$_i$ can be assigned a special symbol "$\perp$" in addition to any positive integer. The set of processes within κ hops of P_i is referred to as κ-neighborhood of P_i. We use $\tilde{n}$ to denote the number of processes in the largest κ-neighborhood of the system. Sub-algorithm $\mathcal{SSWMAC}_i$ has four guarded commands. Process P_i assigns slot$_i$ the value of turn$_i$ if 1) slot$_i$ is $\perp$, 2) turn$_i$ is less than or equal to $\tilde{n}$ and, 3) turn$_i$ is a "valid" slot number (Guarded command $\mathcal{G}_1$). A slot number is said to be valid if 1) it is not equal to the the turn variables in κ-neighborhood and, 2) it is less than those turn variables –within κ-hops– whose respective slot variables are equal to $\perp$. Guarded command $\mathcal{G}_2$ sets variable slot$_i$ to $\perp$ if turn$_i$ is equal to the turn variable of any process in the κ-neighborhood of process P_i. Variable slot$_i$ is assigned $\perp$ and turn$_i$ is incremented modulo $\tilde{n} - 1$ if either slot$_i$ or turn$_i$ is greater than $\tilde{n}$ (Guard command $\mathcal{G}_3$). Variable turn$_i$ is incremented modulo $\tilde{n} - 1$ by Guarded command $\mathcal{G}_4$ if 1) slot$_i$ is $\perp$ and 2) turn$_i$ is not unique in the κ-neighborhood or turn$_i$ is not minimum amongst the turn variables whose respective slot variables are $\perp$. Guarded command $\mathcal{G}_4$ is used by process P_i to traverse across the frame size to find a vacant slot in case the slot corresponding to the curent variable of variable turn$_i$ is not valid.

```
process $P_i$
{
  localvar  $\text{turn}_i \in \mathbb{Z}^+$;
  localvar  $\text{slot}_i \in \mathbb{Z}^+ \cup \{\perp\}$;
  const  $\mathcal{N}_i^\kappa = \{j |\ \text{dis}_{\min}(P_i, P_j) \leq \kappa\}$;
  const  $\mathcal{N}_i^{\kappa\perp} = \{j |\ (j \in \mathcal{N}_i^\kappa) \wedge (\text{slot}_j = \perp)\}$;
  const  $\tilde{n} = \max_{j \in \Pi}(|\mathcal{N}_i^\kappa|)$;
  macro $\min_{\mathcal{N}_i^{\kappa\perp}}(x) \equiv \forall_{j \in \mathcal{N}_i^{\kappa\perp}} : (x \neq \text{turn}_j) \wedge (x < (\text{turn}_j)$
  macro $\text{unique}(x) \equiv \forall_{j \in \mathcal{N}_i^\kappa} : (x \neq \text{turn}_j)$
  macro $\text{valid}(x) \equiv \min_{\mathcal{N}_i^{\kappa\perp}}(x) \wedge \text{unique}(x)$
      /* a valid slot is found */
  $\langle \mathcal{G}_1 \rangle :: (\text{slot}_i = \perp) \wedge (\text{turn}_i \leq \tilde{n}) \wedge \text{valid}(\text{turn}_i) \rightarrow \text{slot}_i := \text{turn}_i$;
      /* current slot is in conflict with other process */
  $\langle \mathcal{G}_2 \rangle :: [\!](\text{slot}_i \neq \perp) \wedge (\text{turn}_i \leq \tilde{n}) \wedge (\exists_{j \in \mathcal{N}_i^\kappa} : \text{turn}_i = \text{turn}_j) \rightarrow \text{slot}_i := \perp$;
      /* slot number is larger than frame size */
  $\langle \mathcal{G}_3 \rangle :: [\!](\text{slot}_i > \tilde{n}) \vee (\text{turn}_i > \tilde{n}) \rightarrow \text{turn}_i := (\text{turn}_i + 1) \ \mathbf{mod}\ [\tilde{n} - 1]$;  $\text{slot}_i := \perp$;
      /* turn variable is not valid, increment it */
  $\langle \mathcal{G}_4 \rangle :: [\!](\text{slot}_i = \perp) \wedge (\text{turn}_i \leq \tilde{n}) \wedge \neg(\text{valid}(\text{turn}_i)) \rightarrow \text{turn}_i := (\text{turn}_i + 1) \ \mathbf{mod}\ [\tilde{n} - 1]$;
}
```

Fig. 5.19: Sub-algorithm $\mathcal{SSWMAC}_i$

The correctness condition of $\mathbb{SSWMAC}$ –mirroring the correctness requirements of the MAC algorithms of wireless sensor networks– stipulates that a process must have slot number different from any other process within κ hops. Let $\mathcal{N}_i^\kappa$ denote the set of processes which are utmost κ hops away from process P_i. We use predicate $\mathcal{P}_{WMAC}$ to characterize the legal states of algorithm $\mathbb{SSWMAC}$; predicate $\mathcal{P}_{WMAC}$ holds true in a global system state if the value of turn_i is unique in the κ-neighborhood of process P_i and turn_i is equal to slot_i, that is,

$$\mathcal{P}_{WMAC} \equiv \forall i \in \Pi : \forall j \in \mathcal{N}_i^\kappa : (\text{slot}_i \neq \text{slot}_j) \wedge (\text{slot}_i = \text{turn}_i).$$

Algorithm $\mathbb{SSWMAC}$ reaches the states satisfying $\mathcal{P}_{WMAC}$ under rather stringent scheduling constraints because actions of each process must be coordinated with those of the processes belonging to the set $\mathcal{N}_i^\kappa$; every process must have the correct local states of processes belonging to $\mathcal{N}_i^\kappa$ to execute any enabled guarded command. Since a process cannot directly read the communication registers of a process κ hops away, a malevolent scheduler can ensure that P_i has outdated κ-neighborhood information whenever it is activated.

The scheduling requirements are not met even if every process is able to get a correct snapshot of its κ-neighborhood and coordinates its actions with the processes in its κ-neighborhood with the help of κ-local mutual exclusion algorithms. Consider a system of 6 processes shown in Figure 5.20. We would like to ensure that eventually no two process within two hops of each other have same slot number; κ is, therefore, instantiated to 2. Each node in the graph is labeled with the ID of the process, the value of slot_i, and the value of turn_i. The identifier of a process is represented by a bold numeral. Figure 5.20 shows an execution of $\mathbb{SSWMAC}$ under a scheduler that never selects two processes within two hops consecutively. We assume that each process gets the correct snapshot of the processes in its 2-neighborhood whenever it is selected by the scheduler and, the underlying scheduler never selects two processes within two hops of each other simultaneously. The system starts in the state where each slot_i variable is $\perp$. The turn variables of processes P_1 and P_5 are equal to 0 in the initial state; turn_2 and turn_6 are equal to 3, while turn_3 and turn_4 are equal to 2. Process P_3 is selected to execute an enabled guarded command in the initial state and, since turn_3 is neither unique nor minimum in the system, P_3 increments it to 3. Process P_4 is selected in the next state; although turn_4 is not equal to any other turn variable in the system, it is not the least among them and, therefore, turn_4 is incremented to 3 as well. The underlying scheduler selects processes P_2 and P_6 in the following state as these two processes are three hops away from each other. Variables turn_2 and turn_6 are incremented as they are neither unique nor minimum in the respective 2-neighborhoods. Note that the 2-neighborhoods of the processes P_3 and P_4 span the whole system.

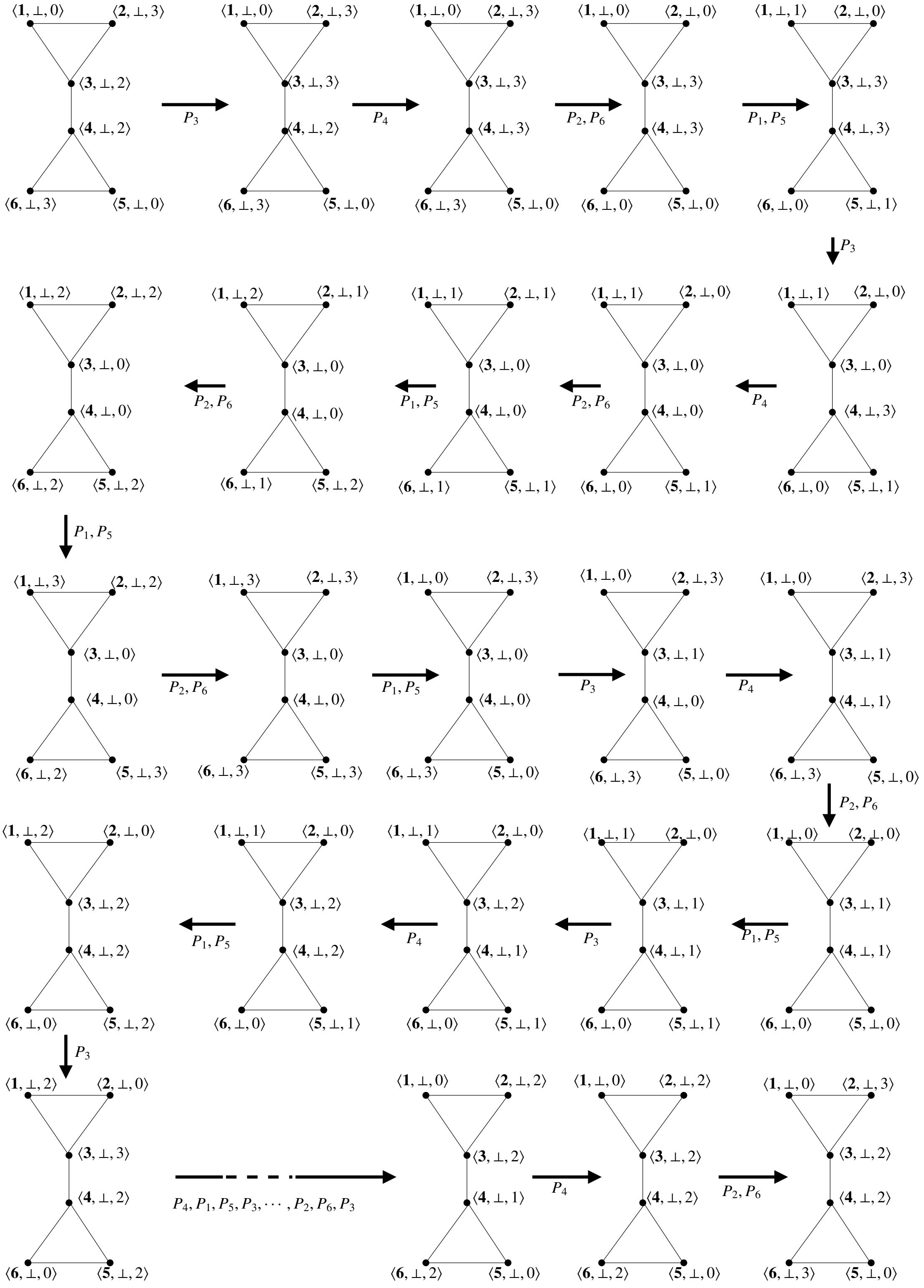

Fig. 5.20: A divergent execution of $\mathbb{SSWMAC}$

Since the the largest 2-neighborhood of the system contains five processes, the turn variables are incremented modulo 4. Therefore, the variables turn_2 and turn_4 are reset to 0. The scheduler next activates processes P_1 and P_5; although turn_1 and turn_5 are equal to 0, both variables are nevertheless incremented as turn_2 and turn_6 are equal to 0 as well. The execution is extended by selecting a process with the minimum turn variable in its 2-neighborhood only when some other process in the 2-neighborhood has its turn variable equal to the minimum value as well. As a result of this strategy, the system eventually gets back to the state in which the execution started. Also, none of the states in the execution satisfies predicate $\mathcal{P}_{WMAC}$. The execution can be extended by repetitive application of the strategy. Note that every process has an enabled guarded command in each state of the execution and, since, each process is selected infinitely often in the infinite execution, the strategy is strongly fair. Thus, despite a strongly fair scheduler that does not violate 2-local mutual exclusion and the availability of correct 2-neighborhood snapshot, the execution shown in Figure 5.20 does not have suffix that satisfies predicate $\mathcal{P}_{WMAC}$. The following definition characterizes the scheduler that fulfills the scheduling requirements of $\mathbb{SSWMAC}$ under the assumption that a lower layer provides a correct snapshot of every κ-neighborhood.

Definition 5.5 (Scheduler $\mathbb{D}_{WMAC}$). *Scheduler $\mathbb{D}_{WMAC}$ selects an enabled process in every execution step while fulfilling the following constraints:*

1) an infinitely often enabled process is selected infinitely often.
2) a process is selected in an execution step if it has a unique and the minimum turn variable in $\mathcal{N}_i^{\kappa\perp}$.
3) an enabled process is ignored if execution of its guarded command leads to an increase in the number of processes whose turn variables are either not unique or not minimum.

Note that, notwithstanding the seeming difficulty of implementing the scheduler specified by Definition 5.5, scheduler $\mathbb{D}_{WMAC}$ is used to show that the convergence of algorithm $\mathbb{SSWMAC}$ towards the states satisfying $\mathcal{P}_{WMAC}$ is guaranteed only under a rather weak scheduler. Moreover, scheduler $\mathbb{D}_{WMAC}$ is also used to underscore the utility of scheduler transformation to preserve the self-stabilization property of such distributed algorithms under much stronger schedulers. We now show that algorithm $\mathbb{SSWMAC}$ is stabilizing under scheduler $\mathbb{D}_{WMAC}$.

We define five counting functions over the state space of algorithm $\mathbb{SSWMAC}$. Function $alloted(\sigma_i)$ returns the number of processes whose slot and turn variables are unique in their respective κ−neighborhoods. The number of processes whose slot variables are $\perp$ and whose turn variables are unique and minimum in the κ-neighborhood is counted by the function $valid(\sigma_i)$. Function $seek(\sigma_i)$ counts the number of processes whose slot variables are $\perp$ and turn variables are either not unique or not the minimum in the κ-neighborhoods. Function $conflict(\sigma_i)$ returns the number of processes whose turn and slot variables are equal to that of at least one process in the κ- neighborhoods. Let dis_j denote the difference between the current $turn_j$ variable of process P_j and the smallest integer value which is unique in the κ-neighborhood of process P_j. Function $dis(\sigma_i)$ is defined as the sum of dis_j values corresponding to all the process in the system. The minimum value is 0 and the maximum value of all functions except $dis(\sigma_i)$ is n. Let $\Delta_{WMAC} = \langle conflict(\sigma_i), n - alloted(\sigma_i), n - valid(\sigma_i), seek(\sigma_i), dis_i\rangle$ be a lexicographic order on $\mathbb{N}^5$. The minimum value of Δ_{WMAC} is $\langle 0, 0, n, 0, 0\rangle$ and it is attained in the state which satisfies $\mathcal{P}_{WMAC}$. The value of function Δ_{WMAC} decreases for any execution step of $\mathbb{WMAC}$ under $\mathbb{D}_{WMAC}$. Guarded commands $\mathcal{G}_2$ and $\mathcal{G}_3$ decreases the value of the function *conflict* as they reset the value of slot variables. The execution of guarded command $\mathcal{G}_4$ in a process leads to, either, an increase in the value of function *valid* and decrease in the value of function *seek* or, a decrease in *dis*. This is due to the constraint on $\mathbb{D}_{WMAC}$ which bars $\mathbb{D}_{WMAC}$ from selecting a process whose action increases the number of turn variables which are either not unique or not minimum. Since scheduler $\mathbb{D}_{WMAC}$ selects only one process in any execution step and a process executes Guarded command $\mathcal{G}_1$ only if it has a valid turn variable, the execution of $\mathcal{G}_1$ in any process leads to a decrease in the value of *valid* without effecting the functions *conflict* and *seek*. The following theorem formally states the correctness of $\mathbb{SSWMAC}$ under the rather constrained scheduler.

Theorem 5.4. *Algorithm* $\$\$\mathbb{WMAC}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_{WMAC}$ *under the scheduler* $\mathbb{D}_{WMAC}$.

Algorithm $\$\$\mathbb{WMAC}$ can be transformed so that $\mathscr{T}(\$\$\mathbb{WMAC})$ is self-stabilizing under any weakly fair scheduler. However, transformation using a global mutual exclusion algorithm would allow only one process to execute an action of $\$\$\mathbb{WMAC}$. Since each process only requires local states of processes within κ hops to correctly evaluate Δ_{WMAC} –and thereby decide whether its action is beneficial for convergence or not– transformation using the global mutual exclusion algorithm, evidently, leads to an unnecessary decrease in concurrency.

The transformation can be optimized with respect to the latent concurrency of a transformed algorithm by using a modified κ-local mutual exclusion algorithm to evaluate a ranking function. As shown in the example above, the parameter κ can be extracted from the convergence proof via inspection of the ranking function. We now show how to optimize the transformation using a κ-local mutual exclusion algorithm. We use the κ-local mutual exclusion algorithm due to Boulinier and Petit [111] because it does not assume any underlying graph structure to function correctly. The κ-local mutual exclusion of [111] is modified to also gather κ-neighborhood snapshots.

Architecture of the κ-Local Optimized Transformer

Figure 5.21 shows the architecture of the optimized transformer. Unlike the transformer using a global mutual exclusion algorithm, this transformer has only two components. The κ-local mutual exclusion forms the lower layer of the transformer. Since the κ-local mutual exclusion of [111] does not require any specific topology to function correctly, this version of transformer does not uses any spanning tree algorithm for tree construction. The modified use algorithm is embedded in the κ-local mutual exclusion algorithm. We briefly describe the local mutual exclusion algorithm before proceeding with the transformation and the associated proofs.

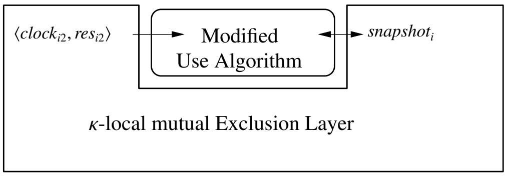
$\langle clock_{i2}, res_{i2} \rangle$
Modified Use Algorithm
$snapshot_i$
κ-local mutual Exclusion Layer

Fig. 5.21: Layered View of the k-Local Scheduler Transformer

κ-Local Mutual Exclusion Layer

The κ-local mutual exclusion algorithm essentially maintains “wavelets” which travel κ hops in each cycle. Each process synchronizes the actions of the modified used algorithm with arrival of wavelet wherein the arrival of wavelet allows a process to access its critical section. The algorithm ensures that if a process has the privilege to execute its critical section, then no process in its κ-neighborhood has the privilege simultaneously. Since the transformation requires the local states of the processes in each κ-neighborhood, the communication register is extended to gather the snapshot. As explained in Section 5.3.1, a compartment to hold the κ-local snapshot is appended to every communication register in the system.

The sub-algorithm implemented by each process is shown in Figure 5.22. The κ-local mutual exclusion algorithm consists of five guarded commands per process. The algorithm uses two variables per process –$clock_{i1}$ and $clock_{i2}$– to control the wavelets. A process P_i uses two auxiliary variables –

```
process $P_i$
{
  localvar $clock_{i1}, clock_{i2} \in \mathbb{Z}$;
  localvar $res_{i1}, res_{i2} \in \mathbb{Z} \times \Pi$;
  const $\mathcal{N}_i \equiv \{P_j | P_j$ is neighbor of $P_i\}$
  macro $normalstep_{ix} \equiv (clock_{ix} \geq 0) \wedge (\forall_{j \in \mathcal{N}_i} (clock_{ix} = clock_{jx})$
                                  $\vee (clock_{jx} = clock_{ix} + 1 \textbf{ mod } [K_x]))$;
  macro $locallycorrect_{ix} \equiv (clock_{ix} \geq 0) \wedge (\forall_{j \in \mathcal{N}_i} (clock_{ix} = clock_{jx})$
                                  $\vee (clock_{jx} = clock_{ix} + 1 \textbf{ mod } [K_x])$
                                  $\vee (clock_{ix} = clock_{jx} + 1 \textbf{ mod } [K_x]))$;
  macro $reset_{ix} \equiv \neg locallycorrect_{ix} \wedge clock_{ix} \geq 0$;
  macro $nextstep_i \equiv normalstep_{i1} \wedge locallycorrect_{i2}$;
  macro $convergestep_{ix} \equiv (clock_{ix} < 0) \wedge (\forall_{j \in \mathcal{N}_i} (clock_{ix} \leq 0) \wedge (clock_{ix} \leq clock_{jx}))$;
  $nextstep_i \longrightarrow$ if $clock_{i1} = (\kappa) \textbf{ mod } [\kappa + 1]$ {
                 if $normalstep_{i2} \wedge (\langle i, clock_{i2} \rangle = res_{i2})$ {
                   $\grave{\mathcal{A}}_i$ ⟨⟨ modified use algorithm⟩⟩;
                   $clock_{i2} := (clock_{i2} + 1) \textbf{ mod } [K_2]$ } ;
                 $res_{i1} := \langle clock_{i2}, i \rangle; res_{i2} := \langle clock_{i2}, i \rangle$; }
               elseif {
                 $res_{i1} := res_{i2}; res_{i2} := \langle clock_{i2}, i \rangle \oplus res_{jy}, j \in \mathcal{N}_i$; }
               writestate() ⟨⟨write latest local state of modified use algorithm⟩⟩;
               $clock_{i1} := (clock_{i1} + 1) \textbf{ mod } [K_1]$;
  $\forall_{x \in \{1,2\}}$ [] $convergestep_{ix} \longrightarrow clock_{ix} := (clock_{ix} + 1) \textbf{ mod } [K_x]$;
  $\forall_{x \in \{1,2\}}$ [] $reset_{ix} \longrightarrow clock_{ix} := -\alpha_x$;
}
```

Fig. 5.22: κ-Local transformed subalgorithm $\mathscr{T}(\mathcal{A}_i)$

res_{i1}, res_{i2}– to select a process which can execute critical section in the κ-neighborhood of P_i. Predicate $normalstep_{i1}$ ($normalstep_{i1}$) holds at process P_i if 1) variable $clock_{i1}$ ($clock_{i1}$) is positive, and 2) $clock_{j1}$ ($clock_{j2}$) variables of all the neighbors of P_i are either equal to $clock_{i1}$ ($clock_{j1}$) or one step ahead of $clock_{i1}$ ($clock_{j1}$). Predicate $locallycorrect_{i1}$ ($locallycorrect_{i2}$) holds at P_i if 1) $clock_{i1}$ ($clock_{i1}$) is positive, and 2) $clock_{j1}$ ($clock_{j2}$) variables of all the neighbors of P_i are equal to $clock_{i1}$ ($clock_{j1}$) or 3) one step ahead or one step behind $clock_{i1}$ ($clock_{j1}$). A process is in an inconsistent state if either one or both of its clock variables are positive and the respective $locallycorrect_{ix}$ predicate does not hold; an inconsistent process resets its $clock_{ix}$ to $-\alpha_x$. α_x must be instantiated to an integer greater than the largest cycle in the system to guarantee the convergence of the algorithm. Predicate $convergestep_{ix}$ holds true if variable $clock_{ix}$ is negative and at least one step behind the respective clock variables of its neighbors. Variable $clock_{ix}$ is incremented modulo K_x if predicate $convergestep_{ix}$ is true. The value of K_x should be greater than the product of the parameter κ and the longest cycle of the system. In a legal global system state, the value of clock variables at a process are synchronized to those of the neighboring processes. A process has to fulfill two sets of preconditions to enter its critical section. The first condition is fulfilled if 1) the value of $clock_{i1}$ is either equal to those if its neighbors or one step behind, 2) $clock_{i2}$ is locally consistent and, 3) $clock_{i1}$ is equal to $(\kappa) \textbf{ mod } [\kappa + 1]$. The second condition is satisfied if 1) $clock_{i2}$ is equal to or one step behind the neighboring clock variables and 2) process P_i has the lowest identifier among the neighbors whose $clock_{j2}$ is equal to $clock_{i2}$. If both conditions are fulfilled, then process P_i 1) accesses its critical section, 2) increments $clock_{i2}$ modulo K_2 and, 3) assigns res_{i1} and res_{i2} the tuple consisting of new value of $clock_{i2}$ and ID of P_i. Variableres_{i1} is assigned the value of res_{i2} and res_{i2} is assigned the the tuple consisting of $clock_{j2}$ and ID of neighbor P_j with least clock value.

Essentially variable res_{i1} contains the ID of the process which gets the access to its critical section next in the κ-neighborhood of process P_i. Variable $clock_{i1}$ is incremented by 1 if the first clock is not equal to $(\kappa)\mathbf{mod}[\kappa + 1]$. Every process, essentially, builds an ordering of res_{j2} in its κ-neighborhood and does not access the critical section if it is not the least element of the ordering. Note that a process P_j marked as the least element of ordering in the κ-neighborhood of another process P_i may not be the least element in its own κ-neighborhood.

Figure 5.23 shows a segment of an execution of the κ-local mutual exclusion with the parameter κ instantiated to 2. The system consists of six processes and the identifier of every process is marked in bold. It is assumed that the underlying scheduler selects every enabled process in every execution step. A process with privilege is marked with an asterisk; each process is labeled with the value of its clock variables. We assume that in the initial state of the execution, the res_{ix} variables are equal to a tuple consisting of the process' identifier and the value of $clock_{i2}$, that is, $res_{ix} = \langle i, 2\rangle$. As a result of this, the initial state is not a legal state, since every process has the privilege to access the critical section. Thus, each process increments both clock variables and again resets res_{ix} variables to the tuple of its identifier and the new value of $clock_{i2}$. Note that since all the clock$_{i1}$ variables are equal to each other, predicate $nextstep_i$ holds true in all the processes. Every process increments clock$_{i1}$ and recomputes the value of res_{i2} in states σ_2 and σ_3 because the difference between first clock variable and κ is not divisible by 3. The first condition to enter critical section is true when the system reaches states σ_4. However, only process P_0 can enter its critical section because, all $clock_{i2}$ variables in the 2- neighborhood of

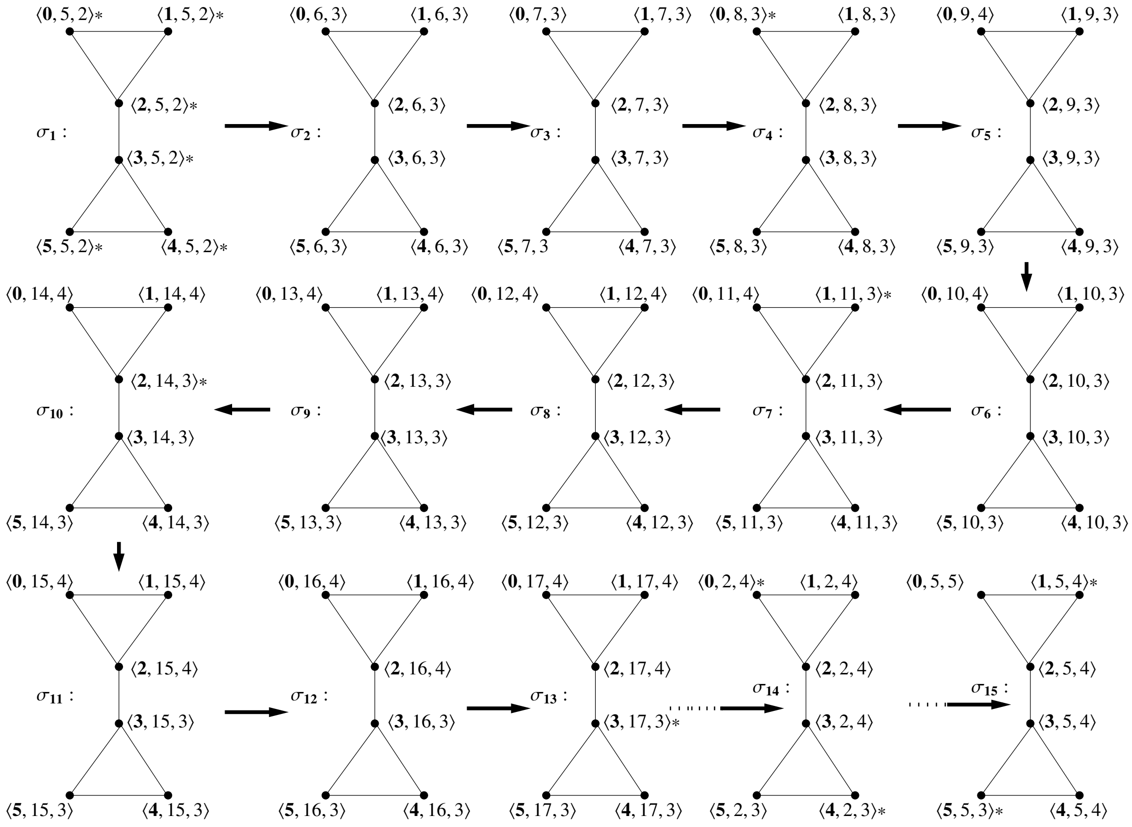

Fig. 5.23: A Segment of an Execution of κ-Local Mutual Exclusion Algorithm

P_0 are equal to each other and process P_0 has the lowest identifier in its 2- neighborhood. Process P_0 is the only process which enters the critical section in state σ_4 because, the process with the lowest identifier (*i. e.* P_2) in the 2-neighborhood of processes P_4 and P_5 selects another process (*i. e.* P_0) it is 2-neighborhood. Consequently, while process P_0 increments both clock variables, all other processes increment only their $clock_{i1}$ variable. No process has the privilege to access its critical section in states σ_5 and σ_6; nonetheless, each process computes variables res_{i1} and res_{i2} to determine the process that has either lowest $clock_{i2}$ or the lowest ID in their respective 2-neighborhood. Process P_1 has the privilege to access its critical section in state σ_7 as it has the lowest ID and the lowest $clock_{i2}$ value; process P_0 cannot access its critical section since $normalstep_{02}$ does not hold. In a similar manner, process P_2 gets the privilege in σ_{10}. As 2-neighborhood of process P_2 covers the whole graph, no other process has the privilege in state σ_{10}. Process P_3 accesses its critical after three execution rounds because the second clock variables of the processes with lower identifier are one step ahead of $clock_{22}$. Two processes get privilege simultaneously in σ_{14} and it is a legal state because processes P_0 and P_4 are three hops away. Process P_4 gets the privilege because it has the lowest $clock_{i2}$ in its 2-neighborhood; all the processes in the 2-neighborhood of P_0 have their second clock variable equal to four, nevertheless, process P_0 gets the privilege as it has the lowest identifier. In a similar fashion, the privilege to access critical section is passed on to processes P_1 and P_4 in state σ_{15}.

Modification of the Use Algorithm

The use algorithm is embedded in the κ-mutual exclusion algorithm as shown in the Figure 5.22. The guards of the use algorithm are modified as explained in Section 5.3.1. Note that the coordination mechanism used for gathering a correct snapshot remains transparent to transformed algorithm. An action of a transformed algorithm is executed when the mutual exclusion layer has the privilege to access its critical section. The latest values of the local variables of a use algorithm are written to the snapshot compartment of the communication register whenever a process increments either of its clock variables.

Correctness of κ-Local Scheduler Transformation

We now show that the κ-local transformer preserves the stabilization property of a use algorithm provided the evaluation of its ranking function requires only a κ-local snapshot. We use $\mathscr{T}_\kappa(\mathbb{A})$ to represent the transformed algorithm based on the κ-local mutual exclusion algorithm. Let predicate $\mathcal{P}_{\kappa mutex}$ hold true in state σ_i if the possession of privilege by process P_j in state σ_i implies that no process in the κ-neighborhood of process P_j has the privilege simultaneously.

Theorem 5.5 (based on [111]). *Irrespective of the initial state, algorithm $\mathscr{T}_\kappa(\mathbb{A})$ reaches a state satisfying predicate $\mathcal{P}_{\kappa mutex}$ within $O(n)$ execution rounds under any weakly-fair scheduler.*

Note that there exists at least one process where either $convergestep_{ix}$ or $reset_{ix}$ holds until both clock variables stabilize to a state where they are synchronized with the clock variables of the neighboring processes.

Let $\hat{\Xi}_{\mathscr{T}_\kappa(\mathbb{A})} = \langle \cdots \sigma_{\kappa mutexi}, \cdots \sigma_{\kappa mutexj}, \cdots \sigma_{\kappa mutexl}, \cdots \rangle$ be a maximal execution of a transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ such that $\sigma_{\kappa mutexi}$ is the state where predicate $\mathcal{P}_{\kappa mutex}$ holds and process P_i has the privilege in its κ-neighborhood for the first time in $\hat{\Xi}_{\mathscr{T}_\kappa(\mathbb{A})}$. The set of processes in the κ-neighborhood of a process P_i is referred to as $\mathcal{N}_i^\kappa$ in the following, that is, $\mathcal{N}_i^\kappa = \{P_j \,|\, \mathrm{dis}(P_i, P_j) \leq \kappa\}$. The set of immediate neighbors of P_i –that is, $\mathrm{dis}(P_i, P_j)$ is equal to 1– is referred to as $\mathcal{N}_i$.

Lemma 5.7. *In any execution of the transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$, process P_i gets correct snapshot of its κ-neighborhood within $O(\kappa \cdot |\mathcal{N}_\kappa^i|)$ execution rounds of reaching state $\sigma_{\kappa mutexi}$.*

Proof. The κ-neighborhood snapshot that process P_i possesses in $\sigma_{\kappa mutexi}$ might be inconsistent since P_i obtains it by reading the communication registers of its neighbors. Nevertheless, process P_i writes the latest value of local variables of the modified use algorithm $\grave{\mathbb{A}}$ before it increments the value of the first clock variable. Observe that after the κ-local mutual exclusion algorithm converges to a correct state, only the guarded command corresponding to the predicate $nextstep_l$ is enabled in any process P_l. Process P_i does not bid to enter its critical section in the next κ execution steps as it –along with the processes in its κ-neighborhood– determines the process which will get privilege in the next phase. While doing so process P_i increments variable $clock_{i1}$ and computes res_{i1} and res_{i2}. Also, every process writes its local state to its communication registers whenever it increments the value of variable $clock_{i1}$.

We show that a process selected to receive privilege also collects the latest local states of the processes in its κ-neighborhood. We prove it in two parts: first correctness of the snapshot is shown for the processes whose distance from the process with the privilege is less than κ, followed by the correctness of the snapshot of the process located exactly κ hops away from the process with the privilege.
Let ς_1 be the value of variable $clock_{i1}$ when process P_i gets the privilege exclusively for the first time in state $\sigma_{\kappa mutexi}$. The value of variable $clock_{j1}$ of any process P_j belonging to κ-neighborhood of process P_i lies between $\varsigma_1 - \kappa$ and $\varsigma_1 + \kappa$ in $\sigma_{\kappa mutexi}$. Note that since P_i has the privilege in state $\sigma_{\kappa mutexi}$, $\varsigma_1 \equiv (\kappa)\,\mathbf{mod}\,[\kappa+1]$ and, for all integers $\varsigma_1 - \kappa \leq \vartheta \leq \varsigma_1 + \kappa$, $\vartheta \not\equiv (\kappa)\,\mathbf{mod}\,[\kappa+1]$ if $\vartheta \neq \varsigma_1$. Additionally, the value of $clock_{j1}$ of process P_j ($P_j \in \mathcal{N}_i$) can either be equal to ς_1 or $\varsigma_1 + 1$. Since predicate $\mathcal{P}_{\kappa mutex}$ holds in state $\sigma_{\kappa mutexi}$, variable res_{j2} of process P_j ($P_j \in \mathcal{N}_i^\kappa$) is equal to $\langle i, clock_{i2}\rangle$ in the state where $clock_{j1}$ is equal to ς_1. A process in the κ-neighborhood of P_i is not able to change the local variables corresponding to algorithm $\mathbb{A}$ till variable $clock_{j1}$ ($P_j \in \mathcal{N}_i^\kappa$) is equal to $\varsigma_1 + \kappa + 1$ because, in the phase corresponding to clock value ς_1, Process P_i has the privilege and all other values are not equivalent to $(\kappa)\,\mathbf{mod}\,[\kappa + 1]$. Process P_i is the minimal element of the ordering induced by the tuples composed of the second clocks and the identifiers of the processes belonging to $\mathcal{N}_i^\kappa$ in the phase corresponding to ς_1. Before leaving its critical section, P_i increments variable $clock_{i2}$. Thereby, process P_i ceases to be the minimal element in $\mathcal{N}_i^k$ because, in the next phase variable $clock_{i2}$ has either the greatest second clock value in $\mathcal{N}_i^k$ or there exists at least one process P_j ($P_j \in \mathcal{N}_i^k$) such that $clock_{j2}$ is equal to $clock_{i2}$ and the identifier of process P_j is smaller than that of P_i. In each phase, at most one process P_j ($P_j \in \mathcal{N}_i^k$) increments $clock_{j2}$. Since the system contains a finite number of processes, process P_i becomes the minimal element in $\mathcal{N}_i^k$ after all the processes in $\mathcal{N}_i^k$ increment their second clock.

Let $\grave{\varsigma}_i$ be the value of $clock_{i1}$ in the state $\sigma_{\kappa mutexi2}$ in which P_i is the minimal element in $\mathcal{N}_i^k$. Since it takes κ steps for any process to decide whether it is the minimal element, $\grave{\varsigma}_1$ is equal to $\varsigma_1 + \lambda(\kappa + 1)$ where λ is a positive integer. It also implies that process P_i has the privilege in $\sigma_{\kappa mutexi2}$. If the distance between process P_i and any process P_j ($P_j \in \mathcal{N}_i^k$) is ϑ, then $|clock_{i1} - clock_{j1}| \leq \vartheta$. Consider a process $P_j \in \mathcal{N}_i$. Variable $clock_{j1} \in \{\grave{\varsigma}_1, \grave{\varsigma}_1 + 1\}$ otherwise predicate $normalstep_{i1}$ would not hold. If $clock_{j1}$ is equal to $\grave{\varsigma}_1 + 1$ then process P_j is in next phase and, did not change the local variables of algorithm $\mathbb{A}$ while incrementing variable $clock_{j1}$ because variable res_{j2} points to process P_i. Process P_i copy of local variables of sub-algorithm $\mathcal{A}_j$ are correct because process P_j did not change them after process P_i read the communication registers of P_j. Similarly, if variable $clock_{j1}$ is equal to $\grave{\varsigma}_1$ then, variable $clock_{j1}$ must have been equal to $\grave{\varsigma}_1$ or $\grave{\varsigma}_1 - 1$ when process P_i read local state of sub-algorithm $\mathcal{A}_j$ before incrementing variable $clock_{i1}$ to $\grave{\varsigma}_1$. In the both cases, process P_j does not change the variables of sub-algorithm $\mathcal{A}_j$ after process P_i read the communication registers of process P_j. Thus, process P_i has the correct snapshot of $\mathcal{N}_i$ when it has privilege in state $\sigma_{\kappa mutexi2}$.

We now consider the processes at the extremities of κ-neighborhood of process P_i. Consider a process P_j such that $\mathrm{dis}(P_i, P_j)$ is equal to κ. Assume that variable $clock_{j1}$ is equal to $\grave{\varsigma}_1 + \kappa$. Process P_j is in the phase corresponding to $\grave{\varsigma}_1 + \kappa + 1$. Process P_j is, thus, "one phase ahead" of process P_i. However, in the state where $clock_{j1}$ was equal to $\grave{\varsigma}_1$, res_{j2} pointed to process P_i as process P_i is the minimal process in the phase corresponding to $\grave{\varsigma}_1$. Hence, process P_j did not change the local variables of sub-algorithm $\mathcal{A}_j$ when it incremented variable $clock_{j1}$ from $\grave{\varsigma}_1$ to $\grave{\varsigma}_1 + 1$. process P_i did not change the local variables of sub-algorithm $\mathcal{A}_j$ in any intermediate state till it reached $\hat{\varsigma}_1 + \kappa$ because none of the intermediate $clock_{j1}$ clock values was equivalent to $(\kappa)\,\mathbf{mod}\,[\kappa + 1]$. Similarly, process P_j could not have changed the local variables of sub-algorithm $\mathcal{A}_j$ after it incremented variable $clock_{j1}$ from $\hat{\varsigma}_1 - \kappa - 1$ to $\hat{\varsigma}_1 - \kappa$. Assume that process P_j indeed modified the local variables of sub-algorithm $\mathcal{A}_j$ while incrementing variable $clock_{j1}$ from $\hat{\varsigma}_1 - \kappa - 1$ to $\hat{\varsigma}_1 - \kappa$. Process P_j must have incremented variable $clock_{j1}$ $2 \cdot \kappa$ times after it executed sub-algorithm $\mathcal{A}_j$ to reach state $\sigma_{\kappa mutexi2}$. Let process P_l be the immediate neighbor of P_j on the path between P_i and P_j. Process P_j can increment variable $clock_{j1}$ at most twice before process P_l increments variable $clock_{l1}$ because, the first clock of all the immediate neighbors of process P_j must be at least equal to $clock_{j1}$ for predicate $nextstep_{j1}$ would not hold. Because each change in first clock value involves reading communication registers of neighbors and writing latest values of the variables of algorithm $\mathbb{A}$, the local state of sub-algorithm $\mathcal{A}_j$ is pushed towards process P_l with the first increment of $clock_{l1}$ after process P_j executes $\mathcal{A}_j$. Similarly process P_l can increment variable $clock_{l1}$ at most twice before its immediate neighbor on the path to process P_i increments the first clock. Hence, the current local state of sub-algorithm $\mathcal{A}_j$ is copied by the immediate neighbor of process P_l when it increments the first clock after process P_l updates its copy of the variables of sub-algorithm $\mathcal{A}_j$. It can be, thus, inductively argued that process P_i must increment variable $clock_{i1}$ at least once so that process P_j is able to increment variable $clock_{j1}$ $2 \cdot \kappa$ times. A further implication of this argument is that process P_i receives the current values of the variables of sub-algorithm $\mathcal{A}_j$ when it increments variables $clock_{i1}$ from $\hat{\varsigma}_1 - 1$ to $\hat{\varsigma}_1$ at the latest. Therefore, process P_i has the current values of the variables of sub-algorithm $\mathcal{A}_j$ in state $\sigma_{\kappa mutexi2}$ even if variable $clock_{j1}$ is $\hat{\varsigma}_1 + \kappa$.

The minimum value of variable $clock_{j1}$ in state $\sigma_{\kappa mutexi2}$ is $\hat{\varsigma}_1 - \kappa + 1$ if the distance between process P_j and process P_i is equal to κ. Process P_j could have accessed its critical section when variable $clock_{j1}$ was equal to $\hat{\varsigma}_1 - \kappa - 1$. Variable $clock_{i1}$ could not have been larger than $\hat{\varsigma}_1 - 1$ in the state where $clock_{j1}$ was $\hat{\varsigma}_1 - \kappa - 1$. Consider the path from process P_j to P_i. Process P_i can increment $clock_{i1}$ from $\hat{\varsigma}_1 - 1$ to $\hat{\varsigma}_1$ only if first clocks of all its neighbors are at least equal to $\hat{\varsigma}_1 - 1$. Note that in the state where $clock_{i1}$ is $\hat{\varsigma}_1 - 1$ and $clock_{j1}$ is $\hat{\varsigma}_1 - \kappa - 1$, only process P_j has an enabled guarded command along path from process P_i to process P_j. Process P_j can increment variable $clock_{j1}$ at most twice before its neighbor on the path to process P_i increments the first clock and when the neighbor does so it copies the latest value of the variables of sub-algorithm $\mathcal{A}_j$. In this manner the latest state of sub-algorithm $\mathcal{A}_j$ is pushed towards process P_i. Process P_i gets the latest state of sub-algorithm $\mathcal{A}_j$ when it reads the communication registers of neighbor on the path to process P_j before incrementing $clock_{i1}$ to $\hat{\varsigma}_1$. Thus, process P_i gets the correct state of sub-algorithm $\mathcal{A}_j$ in state $\sigma_{\kappa mutexi2}$ even if $clock_{j1}$ is $\hat{\varsigma}_1 - \kappa + 1$ because process P_j does not change state of $\mathcal{A}_j$ in any intermediate state. The correctness of the process P_i's copy of state of sub-algorithm $\mathcal{A}_v$ ($\mathrm{dis}(P_i, P_v) = \vartheta, \vartheta < \kappa$) in global state $\sigma_{\kappa mutexi2}$ can be inferred in the similar manner. The lemma follows. □

Let $\varepsilon_{\kappa mutexall}$ be the prefix of $\hat{\Xi}_{\mathcal{T}_\kappa(\mathbb{A})}$ such that every process in the system gets the correct snapshot of its κ-neighborhood for the first time in $\varepsilon_{\kappa mutexsnp}$. Let $\sigma_{\kappa corsnp}$ be the first state of the suffix obtained by removing $\varepsilon_{\kappa mutexsnp}$ from $\hat{\Xi}_{\mathcal{T}_\kappa(\mathbb{A})}$.

Lemma 5.8. *A process executes an action of the modified use algorithm $\grave{\mathbb{A}}$ following the state $\varepsilon_{\kappa mutexsnp}$ if it has a correct snapshot of its κ-neighborhood.*

Proof. A process gets the correct snapshot of its κ-neighborhood whenever the condition to enter the critical section is fulfilled (Lemma 5.7). A process cannot execute any action of the modified use algorithm if it does not have privilege to enter its critical section (by construction). □

In the sequel, we assume that the convergence proof of use algorithm $\mathbb{A}$ provides a ranking function Δ_A which can be evaluated at any process with the help of the local states of the processes in its κ-neighborhood.

Lemma 5.9. *If an action of the modified use algorithm $\grave{\mathbb{A}}$ is executed in a maximal execution of a transformed use algorithm $\mathscr{T}_\kappa(\mathbb{A})$ following the state $\sigma_{\kappa corsnp}$, then the projection of the execution step on the use algorithm leads to a decrease in the value of ranking function $\varDelta_A$.*

Proof. An action of the modified algorithm $\grave{\mathbb{A}}$ is executed by a process if it has the correct snapshot of $\mathcal{N}_i^k$ (Lemmata 5.7 and 5.8). Additionally, the guards of $\grave{\mathbb{A}}$ are enabled only if the respective assignment part leads to a decrease in the ranking function (by construction). The lemma follows. □

We use $\varXi_{|\kappa corsnp}$ to denote the suffix of a maximal execution $\hat{\varXi}_{\mathscr{T}_\kappa(\mathbb{A})}$ under any weakly-fair scheduler such that the first state of $\varXi_{|\kappa corsnp}$ is $\sigma_{\kappa corsnp}$.

Lemma 5.10. *The projection of the suffix $\varXi_{|\kappa corsnp}$ of a maximal execution of $\mathscr{T}_\kappa(\mathbb{A})$ on use algorithm $\mathbb{A}$ is a maximal execution of $\mathbb{A}$ under the scheduler $\mathbb{D}_A$.*

Proof. Algorithm $\mathbb{A}$ has at least one enabled process in any global state by virtue being self- stabilizing with respect to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$. Let $\varXi_{\mathbb{A}|\kappa\mathbb{corsnp}}$ be the projection of $\varXi_{|\kappa corsnp}$ on use algorithm $\mathbb{A}$. Consider a prefix $\varepsilon_{\neg\mathcal{P}_A}$ of $\varXi_{|\kappa corsnp}$ such that no state in $\varepsilon_{\neg\mathcal{P}_A}$ satisfies predicate $\mathcal{P}_A$. If a process executes the modified use algorithm $\grave{\mathbb{A}}$ in any execution step in $\varXi_{|\kappa corsnp}$, then it is the only process to do so in $\mathcal{N}_i^k$ (Theorem 5.5). Whenever a process executes an enabled guarded command of algorithm $\mathbb{A}$, then it does so with the correct snapshot of $\mathcal{N}_i^k$ (Lemma 5.8) and each such execution step leads to a decrease in the value of ranking function $\varDelta_A$ (Lemma 5.9). Let $\sigma_{q|\mathbb{A}} \rightarrow \sigma_{r|\mathbb{A}}$ be an execution step in $\varepsilon_{\neg\mathcal{P}_A}$ such that P_i executes an enabled guarded command of algorithm $\mathbb{A}$. Since there exists at least one process in each state of algorithm $\mathbb{A}$ whose actions lead to a decrease in the value of ranking function $\varDelta_A$, there exists an execution step corresponding to $\sigma_{q|\mathbb{A}} \rightarrow \sigma_{r|\mathbb{A}}$ in a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$. Assume that $\varXi_{|\kappa corsnp}$ is maximal but the projection on algorithm $\mathbb{A}$ is not. It implies that a process with an enabled guard of the modified use algorithm $\grave{\mathbb{A}}$ is not enabled any more in an infinite execution or the final state of a finite execution has an enabled process. However, this is impossible as it would violate weak-fairness constraints. Similarly, in a suffix of $\varXi_{|\kappa corsnp}$ where all the states satisfy predicate $\mathcal{P}_A$, non-maximal projection on use algorithm can be constructed only by violating weak-fairness. This completes the proof. □

Lemma 5.11. *If a use algorithm $\mathbb{A}$ converges to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$, then the projection of the transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ over the use algorithm $\mathbb{A}$ converges to $\mathcal{P}_A$ under any weakly-fair scheduler.*

Proof. Consider the suffix $\varXi_{|\kappa corsnp}$ of any maximal execution of $\mathscr{T}_\kappa(\mathbb{A})$ under a weakly-fair scheduler. The projection of $\varXi_{|\kappa corsnp}$ on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ (Lemma 5.10). A projection that does not converge to the states satisfying predicate $\mathcal{P}_A$ is not possible since it falsifies the premise of the lemma. The lemma follows. □

Theorem 5.6. *The κ-local transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under any weakly-fair scheduler.*

Proof. The convergence of $\mathscr{T}_\kappa(\mathbb{A})$ follows from Lemma 5.11 and the closure follows from Lemma 5.10 and the closure of $\mathcal{P}_A$ under any weakly fair scheduler. □

Corollary 5.2. *The κ-local transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ is self-stabilizing with respect to predicate $\mathcal{P}_{\kappa mutex} \wedge \mathcal{P}_A$ under any weakly-fair scheduler.*

5.3.4 Efficiency of the Transformation

The scheduler oblivious transformation induces overhead in terms of extra memory requirement and delayed convergence. Assume that use algorithm $\mathbb{A}$ requires $M(n)$ bits per process and $T(n)$ execution rounds to stabilize under scheduler $\mathbb{D}_A$, where n is the number of processes in the system implementing distributed algorithm $\mathbb{A}$.

Memory Requirements

Proposition 5.1. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ requires at most $O(\log n)+n\cdot M(n)$ bits per process.*

The spanning tree layer algorithm uses five variables which store the process identifiers and, a process identifier can be stored using a register of $\log n$ bits if there are n processes in the system. The token variable of the mutual exclusion algorithm also requires $\log n$ bits. Since we need a global snapshot to evaluate ranking functionΔ_A, in the worst case, a process may need to store all the local variables of every process.

Proposition 5.2. *The κ-local transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ requires at most $O(\log D)$ + $n \cdot M(n)$ bits per process where D is the diameter of the system.*

The κ-local mutual exclusion algorithm requires $O(\log D)$ bits per process to store the clock variables.

Convergence Delay

Proposition 5.3. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ stabilizes in $O(n^2)$ + $O(T(n)\cdot n)$ execution rounds.*

The spanning tree algorithm requires $O(n^2)$ rounds to build a spanning tree. After a spanning tree is built, the general mutual exclusion algorithm also requires $O(n^2)$ rounds to reach a state where exactly one process has the privilege to access its critical section. A process gets privilege at least once in $O(n)$ execution rounds, therefore, an enabled guarded command of algorithm $\grave{\mathbb{A}}$ is executed at least once in $O(n)$ rounds.

Proposition 5.4. *The κ-local transformed algorithm $\mathscr{T}_\kappa(\mathbb{A})$ stabilizes in $O(n)+O(T(n)\cdot\frac{n\cdot(n-1)}{\kappa})$ execution rounds.*

The κ-local mutual exclusion stabilizes to a state that satisfies predicate $\mathcal{P}_{\kappa mutex}$ in $O(n)$ execution rounds. A process gets privilege to its critical section exclusively in $\mathcal{N}_i^k$ at least once in $O(\frac{n\cdot(n-1)}{\kappa})$ execution rounds.

5.3.5 Simulation Results

We now present the results of simulation carried out to study the efficiency of the transformation. The algorithm $\mathbb{SSWMAC}$ was transformed using both the general mutual exclusion algorithm and the κ-local mutual exclusion algorithm. We gauged the effect of the underlying synchronization layer and the size of the system on the convergence time of the use algorithm. Additionally, we measured the result of increasing the value of the parameter κ on the convergence time of algorithm $\mathscr{T}_\kappa(\mathbb{SSWMAC})$. The transformed algorithms were simulated on an Intel® Celeron® 2 GHz machine with 3 GB RAM. Algorithms were implemented in the Java programming language [112] with the help of the DAJ simulation toolkit [113]. The initial state of the algorithms in each simulation run was set randomly.

The parameter κ was instantiated to 2 while comparing the convergence time of algorithms $\mathscr{T}(\mathbb{SSWMAC})$ and $\mathscr{T}_\kappa(\mathbb{SSWMAC})$. The size of network was increased –in steps of 3– from 6 to 24. Convergence time was measured in the number of execution rounds. Figure 5.24 shows the variation of convergence time of the algorithms $\mathscr{T}(\mathbb{SSWMAC})$ and $\mathscr{T}_\kappa(\mathbb{SSWMAC})$ as the size of the network is increased. The plot shows the average convergence time –along with the confidence interval– at the confidence level of 0.95. Algorithm $\mathscr{T}(\mathbb{SSWMAC})$ outperforms $\mathscr{T}_\kappa(\mathbb{SSWMAC})$ for small sized systems. Although the κ-local mutual exclusion algorithm allows higher concurrency compared to the general mutual exclusion algorithm, no process is able to access its critical section until the κ-local mutual exclusion algorithm converges to its correct behavior. Unlike $\mathscr{T}_\kappa(\mathbb{SSWMAC})$, a process may

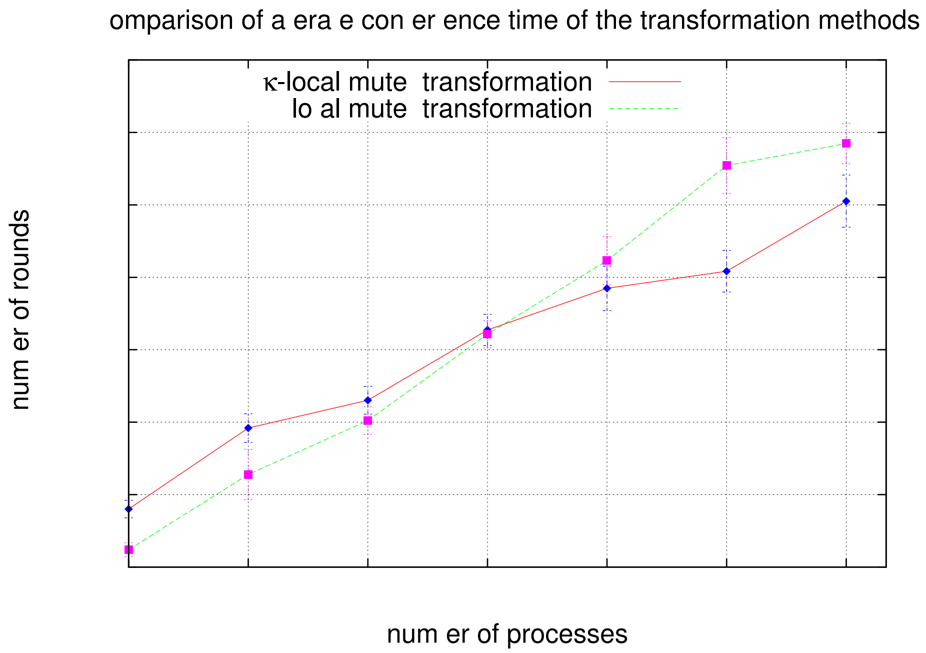

Fig. 5.24: Average Convergence Time of Algorithm $\mathbb{SSWMAC}$ With a Confidence Level of 0.95

execute its critical section in $\mathscr{T}(\mathbb{SSWMAC})$, despite the fact that neither the mutual exclusion algorithm nor the spanning tree algorithm might have stabilized. Therefore, contingent on the initial state and the scheduling strategy, $\mathscr{T}(\mathbb{SSWMAC})$ may reach a state satisfying predicate $\mathcal{P}_{WMAC}$ even before the lower layer stabilizes. The advantage of using the κ-mutual exclusion algorithm becomes clearer as system size is increased. The convergence time of algorithm $\mathscr{T}_\kappa(\mathbb{SSWMAC})$ is smaller than that of algorithm $\mathscr{T}(\mathbb{SSWMAC})$ for system sizes of 21 and 24 even if we take the confidence interval into account. Thus, κ-local scheduler transformation can be used to decrease the convergence time of the transformed algorithm in large systems if the corresponding ranking function can be computed using κ-local information.

Figure 5.25 shows the variation of the average convergence time of algorithm $\mathscr{T}_\kappa(\mathbb{SSWMAC})$ as κ is increased from 2 to 5 on a system with 21 processes. The average convergence time of the transformed algorithm increased substantially as κ was increased from two to six. Observe that an increase in the value of κ increases the number of processes with which each process must synchronize the actions of the transformed algorithm. Therefore, the waiting time of a process with an enabled guarded command of the transformed algorithm increases as well. The effect of κ on the convergence time of the transformed algorithm also underlines the importance of having a ranking function which requires smallest possible synchronization distance for its evaluation. Recall that the value of function $\Delta_{WMAC} = \langle conflict(\sigma_i), n - alloted(\sigma_i), n - valid(\sigma_i), seek(\sigma_i), dis_i\rangle$ decreases for any action of $\mathbb{SSWMAC}$ under scheduler $\mathbb{D}_{WMAC}$. Every process requires the values of the functions which constitute Δ_{WMAC} in order to decide whether its enabled guarded command is beneficial for the convergence of $\mathbb{SSWMAC}$. Since each of the constituting functions essentially encodes the relative state of every process in its κ-neighborhood (thus, a ranking function with "scope" of $2 \cdot \kappa$), $\mathscr{T}_\kappa(\mathbb{SSWMAC})$ can be implemented with the help of a $2 \cdot \kappa$-local mutual exclusion algorithm. Such an implementation provides process P_i the state of the κ-neighborhood of every process in $\mathcal{N}_i^\kappa$. Notwithstanding the cor-

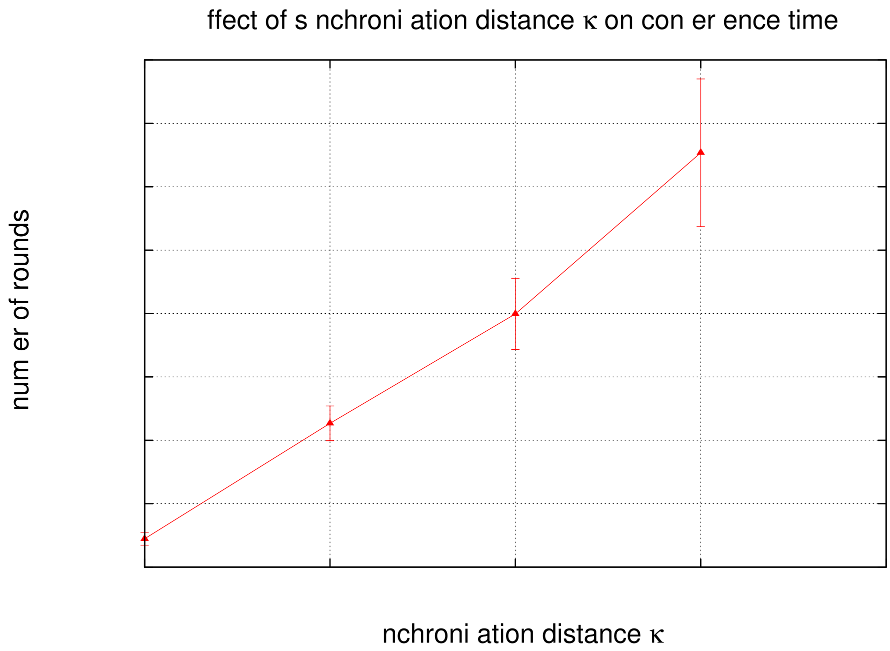

Fig. 5.25: Effect of the Increase in Synchronization Distance κ on the Average Convergence Time of κ-Local Transformed $\mathbb{SSWMAC}$ With a Confidence Level of 0.95

rectness of the resultant algorithm, it leads to an avoidable increase in the convergence time because, Δ_{WMAC} can be correctly evaluated at each process with the help of just the local states of the processes in its κ-neighborhood. As can be seen in Figure 5.25, the average convergence time of algorithm $\mathcal{T}_{\kappa}(\mathbb{SSWMAC})$ increases almost by a factor of 5 if synchronization distance is doubled from 2 to 4. Note that the correctness of the transformed use algorithm remains unaffected by the increase in synchronization distance. Thus, using a ranking function that requires smallest possible synchronization distance during the scheduler transformation can lead to minimum possible transformation-associated overhead. A simulation framework based on the transformation method is described in [114] and can be referred to for further simulation results.

5.4 Discussion

So far the perspective on the scheduler-oblivious transformation has been predominantly algorithmic. We now analyze the transformation on a more abstract level.

5.4.1 Knowledge-Theoretic Interpretation of the Transformation

The "cognizance" of the progress towards the set of legal states is the leitmotif of our approach towards scheduler-oblivious transformation. It is, therefore, pertinent to relate our approach to the general framework of knowledge in distributed systems. We briefly recall relevant definitions before we analyze the transformer from a knowledge-theoretic point of view.

Let ϕ denote a *fact* represented by a formula of a logical language.

Definition 5.6 (Distributed Knowledge [115]). *The knowledge of ϕ is said to be* distributed *in Π if an entity who knew everything that the members of Π know would know ϕ.*

Let $K_i\phi$ denote that P_i knows fact ϕ. $E_\Pi\phi$ holds true if all members of Π know ϕ.

Definition 5.7 (Common Knowledge [115]). *A formula ϕ is said to be* common *knowledge in Π if ϕ is $E^k_\Pi\phi$-knowledge for all $k \geq 1$ where*

$$E^1_\Pi\phi = E_\Pi\phi$$
$$E^{k+1}_\Pi\phi = E_\Pi E^k_\Pi\phi, \ k \geq 1$$

Informally, E_Π represents the phrase "everyone in Pi knows that." Thus, a fact ϕ becomes common knowledge in Π if the phrase "everyone knows that everyone knows ϕ" holds true for any positive integer k. The attainability of common knowledge during an execution is critical for various fundamental problems in distributed computing. Indeed, many impossibility results –for instance, coordinated attack [116]– can be derived by showing that common knowledge of some system property cannot be gathered by a group of processes. Since common knowledge refers to the ability of a group of processes to perform an action in absolute synchrony, attaining common knowledge of nontrivial facts–even if communication between the processes is guaranteed– is impossible [115]. Yet, in practice, distributed systems perform tasks which require coordination of actions of the constituent process; *prima facie* this is in conflict with the impossibility result stated above. However, closer inspection of the problems and the respective solutions reveal that for most of the problems in distributed computing weaker notions of common knowledge are sufficient [115, 117]. The notion of "timestamped" common knowledge is particularly useful in the context of asynchronous distributed systems.

Definition 5.8 (Timestamped Common Knowledge [115, 117]). *Let $K^T_i\phi$ denote that at time T on its clock, process P_i knows the fact ϕ and let $E^T_\Pi\phi$ be defined as $\bigwedge_{\forall P_i} K^T_i\phi$. A formula ϕ is said to be timestamped common knowledge if ϕ is $E^{Tk}_\Pi\phi$-knowledge for all positive integers k.*

In addition to the amount of common knowledge attained during an execution, the amount of common knowledge available in the initial state also effects the outcome of the execution. Halpern and Petride [118] argued that the amount of initial common knowledge a system has determines the complexity of the problem it can solve. Alternatively, relatively complex problems can be solved by a group of processes if each processes knows more about the system.

Consider the problem of identifying a "good" transition among the enabled transitions in a system. A transition can be ascribed a "goodness" attribute based on the problem under consideration. More specifically, a transition in our scheme of things is considered good if it leads to a decrease in the value of ranking function Δ_A. Intuitively, an entity that has access to the local states of all the processes in the system and the ranking function can identify a good transition. Hence, if we use ϕ to denote the fact that the enabled guarded command of process P_i is a good transition, then the above described global observer would know ϕ in every system state. Thus, the fact that the enabled guarded command of process P_i is a good transition in global state σ_x is distributed knowledge.

Common knowledge of the fact that process P_i has the good transition in a system state cannot be attained in any execution of an untransformed use algorithm. Common knowledge of the fact would require that all the processes have access to a global clock and the local states of all the other constituent processes. Note that it takes η execution rounds for a process to know the current local state of a neighbor which is η hops away. It can be, thus, inferred that even timestamped common knowledge of a good transition in any state cannot be attained in an untransformed use algorithm. Common knowledge of good transitions is important because it decreases the inherent superfluous non-determinism. The initial common knowledge that every process has is limited to the identifiers of the neighboring processes.

Consider now a transformed use algorithm $\mathscr{T}(\mathbb{A})$. Every process has the ranking function embedded in the guarded commands. This is additional knowledge compared to an untransformed use algorithm. Thus, initial common knowledge of a transformed use algorithm consists of the identifiers of the neighboring processes and the ranking function.

A process executing a transformed use algorithm $\mathscr{T}(\mathbb{A})$ gets the local states of all the constituent processes whenever it gets the privilege. We use "phase" to refer to a segment of an execution where each process gets privilege to access its critical section at least once. In every phase, a process chooses to execute an action of distributed algorithm $\mathbb{A}$ only if it infers that the action will lead to a decrease in the value of ranking function Δ_A. Alternatively, in each phase every process knows whether to execute an action of algorithm $\mathbb{A}$ or not. Hence, the fact that each process knows whether to execute an action of $\mathbb{A}$ in phase ι or not becomes timestamped common knowledge in phase ι.

The scheduler-oblivious transformation, thus, does away with the lack of common knowledge in a use algorithm. It increases initial common knowledge in the system. Additionally, it provides an underlying mechanism to attain timestamped common knowledge which is necessary for the convergence of the algorithm under an unfavorable scheduler.

5.4.2 A Scheduler-based Perspective of the Transformation

Recall that hyperfairness ensures that each transition emanating from a state is taken if the state appears infinitely often in an execution. A distributed algorithm that is weakly-stabilizing can be transformed into a self-stabilizing algorithm if the actions of the algorithm are synchronized with that of a hyperfair scheduler. Such a hyperfair scheduler should also be self-stabilizing so that an inconsistent initial state of the scheduler does not destroy the convergence property of the weakly-stabilizing algorithm. Attie *et al.* [26] proposed an implementation of a hyperfair scheduler. The hyperfair scheduler of [26] is realized as a separate process that has access to the local states of all the constituent processes. Moreover, the scheduler works only for the algorithms which use multi-party interaction as the communication primitive. Völzer [41] proposed another implementation of a hyperfair scheduler that uses timers and randomization to select a process among the enabled processes in any system state. A hyperfair scheduler for transformation of a self-stabilizing algorithm must be able to prioritize the enabled processes –in addition to the respective waiting times– on the basis of the overall progress of the algorithm towards the set of legal states.

An alternative of implementing a hyperfair scheduler is to synchronize the actions of the algorithm with the scheduler under which the algorithm indeed converges. A transformed algorithm $\mathscr{T}(\mathbb{A})$ can be viewed as a composition of a use algorithm and a scheduler that is derived from the scheduler under which the use algorithm is shown to be self-stabilizing. The scheduler is implemented in a distributive manner implying that instead of employing a separate process running the scheduler, each process runs an instance of the scheduler. The scheduler allows a process to execute its enabled guarded command only if it fulfills certain conditions. Our method of embedding the scheduler in a use algorithm is similar to the scheme proposed in [119] in the sense that, each conditional statement of the use sub-algorithm is strengthened to wait for the signal from the scheduler. However, if there exist multiple processes whose enabled guarded commands are conducive for overall progress, then the sequence in which the privilege is passed around determines the process which is selected by the scheduler. This owes to the fact that the scheduler uses a mutual exclusion algorithm for synchronization. Thus, the scheduler is not maximal where maximality [120] of a scheduler refers to the ability of the implemented scheduler to produce all executions that are possible under the original scheduler. Nevertheless, our approach of transformation preserves the self-stabilization property of a distributed algorithm under a much more powerful scheduler.

5.5 Summary

We presented a transformation technique that preserves the self-stabilization property of a distributed algorithm under a scheduler which is less constrained than then original scheduler. The transformation uses the proof artifacts to monitor the progress of the algorithm towards the set of legal states. The scheduler-oblivious transformer uses self-stabilizing mutual exclusion algorithms for achieving synchronization among the constituent processes. We also showed how to optimize the concurrency in the transformed algorithm by using κ-local mutual exclusion algorithms for synchronization.

The target semantics was restricted to a sequential weakly-fair scheduler. It would be, nevertheless, worthwhile to extend the scheme for distributed schedulers. We would also like to investigate the effect of relaxing atomicity assumptions on the scheduler-oblivious transformation. Primary motivation for developing this transformation technique was to provide required algorithmic machinery for the lifting composition. It is, therefore, in order to investigate the role of the transformer if the component algorithms do not have orthogonal state space. We address the above mentioned issues in the following chapter.

6

Generalized Compositional Operators

6.1 Introduction

The recurrent theme in this work has been the exploitation of the liveness proofs of self-stabilizing algorithms to counter adversarial schedulers. We used the knowledge of ranking function to devise a compositional operator to transcend the incompatibility of the schedulers of the components. Convergence proofs have also been used to preserve the self-stabilization property of a distributed algorithm under a stronger scheduler. Despite the common kernel, the lifting composition and the transformation are, seemingly, oblique to each other. We need to bridge the two methods to meet our aim of coming up with a compositional framework for the design of self-stabilizing algorithms. Although the lifting composition offers the potential skeleton of a compositional framework, it needs to be integrated with the scheduler-oblivious transformation in order to be practically applicable. However, the scheduler oblivious transformation itself is defined for a rather restricted execution semantics. The transformation method, should, therefore, be extended to preserve the correctness properties of self-stabilizing algorithms under much stronger schedulers and reduced atomicity. Another deficiency of the lifting composition is the lack of support for the algorithms which are "conditionally" self-stabilizing. The presence of variable dependency between component algorithms introduces additional constraints, since inconsistent states of variables of one component may interfere with the convergence of the other component. The lifting composition, therefore, needs to be enhanced so that self-stabilization is preserved even if the component algorithms read variables belonging to their counterparts.

Outline. The aim of this chapter is to develop a compositional operator that can preserve the self-stabilization property despite variable dependency and incompatible schedulers. To that end, we show that the scheduler-oblivious transformation preserves the self-stabilization property of self-stabilizing algorithms even under less coarse atomicity and stronger schedulers. We integrate the scheduler-oblivious transformation with the lifting composition to define the generic compositional operator which can compose self-stabilizing algorithms that are designed under different schedulers, communication model, and atomicity assumptions. The generic lifting compositional operator is then further extended to define a truly symmetric compositional operator that works even under variable dependency.

The chapter is organized as follows. The scheduler-oblivious extension of Chapter 5 is extended for stronger execution models in Section 6.2. The extended scheduler-oblivious transformation is integrated with the lifting composition in Section 6.3. The resultant compositional operator is further generalized in Section 6.4. The chapter ends with a summary in Section 6.5.

6.2 Extensions of the Scheduler-Oblivious Transformation

We now analyze the effectiveness of the scheduler-oblivious transformation under a relaxed model and execution semantics. We first consider models which do not offer composite atomicity and, subsequently, preservation of self-stabilization property under a distributed scheduler is proven.

6.2.1 Read/Write Atomicity and Scheduler Transformation

The underlying execution model so far assumed composite atomicity; each process executed its enabled guarded command as an atomic operation. This implies that a process reads the communication registers of its neighbors, does internal computation and, writes the new values of its local variables to its communication registers in one indistinguishable operation. As we discussed earlier in this work, although composite atomicity eases the task of providing correctness proofs, it restricts the size of the state space of the algorithm under consideration. Thus, the result of the scheduler-oblivious transformation must preserve the self-stabilization property of a use algorithm even under read/write atomicity.

Altered Execution Model

We assume that the target system offers read/write atomicity only. More specifically, if a process is enabled by the scheduler, it either reads the write communication registers of its neighbors or, it does internal computation and writes back the updated local variables to its write communication registers. Note that such a result of relaxation model increases the size of state space and the number of possible executions. This is because, under the read/write atomicity model, each process requires extra local variables to store the values read from the write communication registers of its neighbors, and these new variables are also susceptible to transient faults.

Correctness of the Transformation under Read/Write Atomicity

Assume that a use algorithm $\mathbb{A}$ is transformed to $\mathscr{T}(\mathbb{A})$ in the manner described in Chapter 5. Since the transformed algorithm requires the lower layer algorithms to converge, we recall the correctness of the spanning tree algorithm and the mutual exclusion algorithm under read/write atomicity.

Corollary 6.1 (based on [94]). *Irrespective of the initial state, in a bounded number of execution steps a correct tree spans the system under the read/write atomicity model.*

Corollary 6.2 (based on [109]). *Every execution of the mutual exclusion algorithm reaches a state where exactly one process has the token under the read/wrtie atomicity model.*

Corollary 6.2 follows from the fact the process designated as root increments $token_{\mathrm{root}}$ modulo $4 \cdot n - 5$. We now show that the projection of a maximal execution of $\mathscr{T}(\mathbb{A})$ is an execution of $\mathbb{A}$ under $\mathbb{D}_A$ even if $\mathscr{T}(\mathbb{A})$ is executed under read/write atomicity. Let $\Xi_{|\sigma_{corsnpRW}}$ be the suffix of a maximal execution of $\mathscr{T}(\mathbb{A})$ under a weakly-fair scheduler and read/write atomicity such that only one process gets the token in any state of $\Xi_{|\sigma_{corsnpRW}}$.

Lemma 6.1. *The projection of $\Xi_{|\sigma_{corsnpRW}}$ over use algorithm $\mathbb{A}$ is a maximal execution of $\mathbb{A}$ under scheduler $\mathbb{D}_A$.*

Proof. Assume that use algorithm $\mathbb{A}$ is designed for read/write atomicity. The lemma can be proven in a manner analogous to the proof of Lemma 5.5. We now consider the case where algorithm $\mathbb{A}$ is designed only for composite atomicity. All the guarded commands of process P_i are grouped inside the critical section of the mutual exclusion algorithm. Therefore, read and write operations corresponding to the guarded commands of algorithm $\mathbb{A}$ are executed only when process P_i has the token (by construction).

It implies that when process P_i executes an action of algorithm $\mathbb{A}$ in the transformed algorithm $\mathscr{T}(\mathbb{A})$, it completes read operation, internal computation and, write operation in one uninterrupted step. Recall that, under scheduler $\mathbb{D}_A$ each process P_i completes read, internal computation and write operation as one indistinguishable step as it is executed under the composite atomicity model. Thus, each step of the modified use algorithm $\grave{\mathbb{A}}$ under a weakly-fair scheduler is executed in a manner analogous to an execution step of $\mathbb{A}$ under $\mathbb{D}_A$. Rest of the proof can be drawn in a fashion similar to Lemma 5.5. □

Theorem 6.1. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under any weakly fair scheduler and read/write atomicity model.*

Proof. Convergence of algorithm $\mathscr{T}(\mathbb{A})$ towards predicate $\mathcal{P}_A$ follows from the fact that, the projection of any maximal execution of algorithm $\mathscr{T}(\mathbb{A})$ over algorthm $\mathbb{A}$ is an execution of $\mathbb{A}$ under $\mathbb{D}_A$ even if read/write atomicity model is assumed (Lemma 6.1). Closure follows from the assumption that predicate $\mathcal{P}_A$ is closed under a weakly fair scheduler. □

The scheduler-oblivious transformation, therefore, preserves the self-stabilization property of a use algorithm even if the atomicity assumption are violated during the execution of the transformed algorithm.

6.2.2 Scheduler Transformation and Distributed Scheduler

We have so far assumed that the target scheduler is a sequential scheduler; in each execution step it selects exactly one enabled process while fulfilling weak fairness constraints. However, as we argued earlier, compared to a sequential scheduler, a distributed scheduler mirrors the implementation scenario more closely. We now analyze the behavior of the transformed algorithm under a distributed scheduler.

Altered Execution Model

We now assume that the target scheduler is a distributed scheduler; in each step, the scheduler can select any subset of enabled processes as long as the selection does not violate the weak fairness constraint. Relaxation of this assumption also leads to increased state space size and number of possible executions, since the number of strategies a scheduler can employ increases.

Correctness of the Transformer under a Distributed Scheduler

Any execution of an algorithm produced under a distributed scheduler can be reproduced if the algorithm is executed under the read/write atomicity model and a sequential scheduler. For example, consider an execution where a distributed scheduler selects processes P_i and P_j. An equivalent execution step assuming read/write atomicity can be generated by following the selection of read operations of P_i and P_j with the respective write operations. The corollaries stated below follow immediately.

Corollary 6.3 (based on [94]). *Irrespective of the initial state, in a bounded number of execution steps a correct tree spans the system under a distributed scheduler.*

Corollary 6.4 (based on [109]). *Every execution of the mutual exclusion algorithm reaches a state where exactly one process has the token under a distributed scheduler.*

It remains to be proven that, under a distributed scheduler, the transformed algorithm $\mathscr{T}(\mathbb{A})$ preserves the properties of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$. Let $\Xi_{|\sigma_{corsnpDS}}$ be the counterpart of $\Xi_{|\sigma_{corsnpRW}}$ under any weakly-fair distributed scheduler.

Lemma 6.2. *The projection of $\Xi_{|\sigma_{corsnpDS}}$ over use algorithm $\mathbb{A}$ is a maximal execution of $\mathbb{A}$ under scheduler $\mathbb{D}_A$.*

Proof. Consider an execution step of algorithm $\mathscr{T}(\mathbb{A})$ in which the values of local variables of algorithm $\grave{\mathbb{A}}$ change. Since the actions of algorithm $\grave{\mathbb{A}}$ are embedded in the critical section of the mutual exclusion algorithm, execution of the enabled guarded command of $\grave{\mathbb{A}}$ in a single process led to the aforementioned state change (by construction). There exists an equivalent execution step of $\mathbb{A}$ under scheduler $\mathbb{D}_A$ since assignment parts are unchanged. The lemma follows. □

The theorem below follows immediately.

Theorem 6.2. *The transformed algorithm $\mathscr{T}(\mathbb{A})$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under any weakly-fair distributed scheduler.*

Note that the theorem holds even if the target system only offers read/write atomicity. Thus, the transformed algorithm preserves the self-stabilization property of the use algorithm even if assumptions about sequential scheduler *and* composite atomicity are removed.

6.2.3 An Extension for the Message Passing Communication Model

The properties of the transformed algorithm $\mathscr{T}(\mathbb{A})$ have been proven so far assuming the shared memory communication model. However, more often than not, distributed algorithms are implemented on platforms that offer communication via messages over unreliable communication links. It is, therefore, desirable to make the scheduler-oblivious transformation work under the message passing communication model. However, unlike the extensions discussed above, scheduler-oblivious transformer needs extra algorithmic components to work under the message passing communication model. The self-stabilizing extensions [121, 122] of the alternating bit algorithm can be used to adapt the transformed algorithm to the relaxed communication model. The self-stabilizing data link algorithm essentially implements a token passing algorithm over a pair of processes. One of the processes acts as the receiver and the other dons the role of the sender. Each process maintains a counter variable that tracks the label of the last message sent or received. The sender process has the token if the label of the message received is greater than its counter variable. The receiver process gets the token if the message label is not equal to its counter variable. The sender process increments the value of counter if it gets the token; the receiver process updates its counter variable when it gets the token. In a legal state, the message labels and the counter variables of both processes are equal implying that a send operation of a message is followed by the read operation of the message with the same label. An execution of the transformed algorithm under the shared memory model can be simulated by implementing the two instances of the self-stabilizing data link algorithm –one for each direction of communication– on each communication link in the system.

6.3 Extensions of Lifting Composition

We now bring the focus back on the lifting composition. We first modify the composition for general communication topologies. Subsequently, we provide an extension of the lifting composition such that, the composed algorithm is self-stabilizing under any weakly-fair scheduler irrespective of the schedulers of the component algorithms.

6.3.1 Lifting Composition for General Communication Graphs

While proving the correctness of the lifting composition in Chapter 4, it was assumed that communication topology or a lower layer allows every process to read the local states of all the processes in the system. We relax this assumption and show that, after slight modification, lifting composition preserves the self-stabilization property of the component algorithms even in general graphs.

Definition

We briefly recall the properties of the component algorithms. Algorithm $\mathbb{A}$ is self- stabilizing with respect to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$; algorithm $\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Algorithm $\mathbb{A}$ has exactly one enabled guarded command in every state and scheduler $\mathbb{D}_A$ never violates weak fairness. Let m_i denote the number of guarded commands in the mutual exclusion sub-algorithm run by a process P_i. Variable x_i refers to the vector of the local variables of sub-algorithm $\mathcal{A}_i$; y_i represents the vector of the local variables of sub-algorithm $\mathcal{B}_i$. Note that lifting composition requires the evaluation of ranking function Δ_B of component $\mathbb{B}$ to function correctly. Lifting composition for general graphs, therefore, uses the global snapshot collection framework of the scheduler transformation to evaluate ranking function Δ_B in any global state. Under these assumptions lifting composition for general graphs is defined as follows.

Definition 6.1 (Lifting Composition for General Graphs). *Sub-algorithm $\mathcal{A}_i{\triangle_{\mathrm{G}}}\mathcal{B}_i$ consists of $l_i \cdot m_i$ guarded commands of the following structure:*

$$\mathcal{G}_{Ai_p} \wedge have\_token_i \quad \rightarrow \left\{ \begin{array}{l} \grave{\mathcal{G}}_{Bi_1} \rightarrow act_{Ai_p}; act_{Bi_1}; paint\_token; \\ \qquad \vdots \\ \grave{\mathcal{G}}_{Bi_\eta} \rightarrow act_{Ai_p}; act_{Bi_\eta}; paint\_token; \end{array} \right\} \quad (1)$$

$$\mathcal{G}_{Ai_p} \wedge \neg have\_token_i \quad \rightarrow act_{Ai_p}; token\_op; \qquad (2)$$

for all $q \in \{1, \cdots, l_i\}$. Algorithm $\mathbb{A}{\triangle_{\mathrm{G}}}\mathbb{B}$ is the union of all sub-algorithms $\mathcal{A}_i{\triangle_{\mathrm{G}}}\mathcal{B}_i$

$$\mathbb{A}{\triangle_{\mathrm{G}}}\mathbb{B} = \bigcup_{\forall i} \mathcal{A}_i{\triangle_{\mathrm{G}}}\mathcal{B}_i$$

run by the processes in Π.

Description

The composed algorithm has two types of guarded commands. There are l_i guarded commands of Type 1. A Type 1 guard is conjunction of the guard of algorithm $\mathbb{A}$ and the predicate *have_token*$_i$. Predicate *have_token*$_i$ holds true when the mutual exclusion sub-algorithm run by process P_i gets the privilege to access the critical section. If a Type 1 guard is true, then process P_i executes an action of algorithm $\mathbb{A}$ followed by an action of algorithm $\mathbb{B}$ only if it leads to a decrease in the value of ranking function Δ_B; Process P_i updates the snapshot part of communication register independent of whether a guarded command of algorithm $\mathbb{B}$ is executed or not. Sub-algorithm $\mathbb{A}{\triangle_{\mathrm{G}}}\mathbb{B}$ consists of $l_i \cdot m_i - 1$ type 2 guarded commands. A Type 2 guard is enabled in process P_i if a guard of algorithm $\mathbb{A}$ holds true but process P_i does not possess the privilege. The respective action of algorithm $\mathbb{A}$ is executed if Type 2 guard is enabled. Additionally, process P_i performs the assignment statement of the enabled guard of the mutual exclusion sub-algorithm. Note that we assume that guarded commands of the mutual exclusion are structured such that a single guard corresponds to the state where a process has the privilege and all the other guarded commands manage the token circulation.

Essentially, the composition operation wraps the actions of algorithm $\grave{\mathbb{B}}$ inside the guards of the mutual exclusion algorithm so that the adversarial selections of scheduler $\mathbb{D}_A$ do not hamper the convergence of algorithm $\mathbb{B}$. The guards of algorithm $\mathbb{A}{\triangle}\mathbb{B}$ differ from $\mathbb{A}{\triangle_{\mathrm{G}}}\mathbb{B}$ with respect to the presence of unaltered guards of $\mathbb{B}$. The shielding of guards of $\mathbb{B}$ is required for the correctness of the composed algorithm. The structure of the guards of $\mathbb{A}{\triangle}\mathbb{B}$ could have been reused in $\mathbb{A}{\triangle_{\mathrm{G}}}\mathbb{B}$ without affecting the correctness of the resultant algorithm. However, it would slow down the convergence of the algorithm $\mathbb{A}$ because, actions of $\mathbb{A}$ would also be synchronized with those of the mutual exclusion algorithm. Since algorithm $\mathbb{A}$ exhibits convergence under scheduler $\mathbb{D}_A$ directly, such synchronization is not required. Note that the mutual exclusion algorithm can be selected based on the structure of the ranking function $\varDelta_B$ of algorithm $\mathbb{B}$. The actions of the spanning tree layer algorithm do not affect the outcome of the composition in case the global mutual exclusion algorithm is selected for synchronization, since the intra-process fairness ensures that actions of composed algorithm and the spanning tree algorithm are executed infinitely often in any maximal execution of $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$. The overhead of using lower layers for synchronization and global (κ-local) snapshot is solely born by algorithm $\mathbb{B}$.

Correctness of the Composition

We now show that $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ preserves the self-stabilization property of both the components.

Lemma 6.3. *The projection of any maximal execution of algorithm $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ under scheduler $\mathbb{D}_A$ on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$.*

Proof. Assignments statements of algorithm $\mathbb{A}$ are unchanged in $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$. Lemma follows immediately if a κ-local mutual exclusion algorithm is used because guards of algorithm $\mathbb{A}$ remain enabled independent of whether a process has privilege or not. If the general mutual exclusion algorithm is used, then the actions of the composed algorithm cannot be delayed indefinitely by any process as each process activates the guards of the spanning tree algorithm and $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ equally often. □

The following lemma can be directly inferred from Lemma 6.3.

Lemma 6.4. *If all maximal executions of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ satisfy predicate $\mathcal{P}_A$ then, the projection of any execution of $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ under scheduler $\mathbb{D}_A$ on algorithm $\mathbb{A}$ also satisfies predicate $\mathcal{P}_A$.*

Lemma 6.5. *The projection of any maximal execution of the composed algorithm $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ under weakly-fair scheduler $\mathbb{D}_A$ on algorithm $\mathbb{B}$ is a maximal execution of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$.*

Proof. Recall that, in each state of algorithm $\mathbb{B}$, there exists at least one process whose guarded command leads to a decrease in the value of ranking function $\varDelta_B$ because algorithm $\mathbb{B}$ converges to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Let P_i be the process in which an enabled guarded command of algorithm $\mathbb{B}$ leads to a decrease in the value of ranking function $\varDelta_B$ in state σ_x of a maximal execution of $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$. Assume that process P_i does not possess the token in state σ_x. The execution can be extended by scheduler $\mathbb{D}_A$ by selecting process which has token in state σ_x. Eventually, the system will reach a state where process P_i gets the privilege to access the critical section. Thereon, scheduler $\mathbb{D}_A$ can delay a step of algorithm $\mathbb{B}$ only by not selecting process P_i despite the fact that it has the privilege. Process P_i has an enabled guarded command of algorithm $\mathbb{A}$ as well, therefore, the execution cannot be extended by ignoring process P_i as it violates weak fairness constraint. The lemma follows from the fact that the assignment statements of algorithm $\mathbb{B}$ are unchanged. □

Theorem 6.3. *Algorithm $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under scheduler $\mathbb{D}_A$.*

Proof. Preservation of self-stabilization of algorithm $\mathbb{A}$ in $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ follows from Lemma 6.4. Convergence and closure of algorithm $\mathbb{B}$ follows from Lemma 6.5. □

Note that, since the underlying mutual exclusion algorithms retain their respective correctness even under read/write atomicity and distributed scheduler, the following corollaries can be directly inferred from Theorem 6.3.

Corollary 6.5. *Algorithm $\mathbb{A}{\vartriangle_{\mathrm{G}}}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under scheduler $\mathbb{D}_A$ even if $\mathbb{D}_A$ is a distributed scheduler.*

Corollary 6.6. *If algorithm $\mathbb{A}$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$ and read/write atomicity, then algorithm $\mathbb{A}^{\triangle_G}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under scheduler $\mathbb{D}_A$ and read/write atomicity model.*

The overhead induced by the composition is due to the usage of lower layers for synchronization. Let algorithm $\mathbb{A}$ require $M_A(n)$ bits per process –n being the number of processes in the system– to converge under $\mathbb{D}_A$; $M_B(n)$ denote the bits per process required by algorithm $\mathbb{B}$ to converge under scheduler $\mathbb{D}_B$. Memory overhead is caused by the extra bits required to run the lower layer algorithms and to store the local variables of algorithm $\mathbb{B}$ at each process.

Proposition 6.1. *Algorithm $\mathbb{A}^{\triangle_G}\mathbb{B}$ requires at most $O(\log n) + n \cdot M_B(n) + M_A(n)$ bits per process.*

Similarly, $\mathbb{A}^{\triangle_G}\mathbb{B}$ under scheduler $\mathbb{D}_A$ takes longer than algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$ to converge to predicate $\mathcal{P}_B$ because actions of algorithm $\mathbb{B}$ are synchronized with the mutual exclusion algorithms. Let algorithm $\mathbb{A}$ takes $T_A(n)$ to converge to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$ and algorithm $\mathbb{B}$ takes $T_B(n)$ to converge to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$.

Proposition 6.2. *Algorithm $\mathbb{A}^{\triangle_G}\mathbb{B}$ converges to a state satisfying $\mathcal{P}_A \wedge \mathcal{P}_B$ in at most $O(n^2) + T_A(n) + O(T_B(n) \cdot n)$ execution rounds.*

6.3.2 Symmetric Lifting Composition

The focus of the lifting composition has been to preserve the self-stabilization property of one of the component algorithm under the scheduler of the other component. We now focus on the scenario where two self-stabilizing algorithms are required to be composed such that the composed algorithm is self-stabilizing under a scheduler that is stronger than the individual schedulers. Since the asymmetric version of lifting composition can only shield one of the components from the scheduler of other component, a symmetric composition operator that preserves the self-stabilization properties of both components is required.

We define symmetric lifting composition for self-stabilizing algorithms that ensures that the composed algorithm is self-stabilizing under a weakly-fair scheduler regardless of the respective schedulers of the components. However, correct evaluation of ranking functions of component algorithms may require different synchronization mechanisms. Therefore, we present three versions of symmetric lifting composition. It is assumed throughout this section that each component algorithm has exactly one enabled guarded command per process. Algorithm $\mathbb{A}$ consists of l_i guarded commands and algorithm $\mathbb{B}$ has m_i guarded commands per process. In order to gather a snapshot of the system, the communication registers are extended as described in Chapter 5.

Symmetric Lifting Composition with Global Mutual Exclusion

In case both components require a global mutual exclusion algorithm (referred to as GME in the sequel), the symmetric composition is defined as follows.

Definition 6.2 (Symmetric Lifting Composition with GME). *Sub-algorithm $\mathcal{A}_i{}^{\triangle_S}\mathcal{B}_i$ consists of $9{\cdot}l_i{\cdot}m_i$ guarded commands of the following structure:*

$$\begin{array}{ll}
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} < 0) \wedge (\delta_{Aix} < 0) & \rightarrow act_{Ai_x}; act_{Bi_y}; paint\_token; (1) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \mathcal{P}_B & \rightarrow act_{Ai_x}; act_{Bi_y}; paint\_token; (2) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} < 0) \wedge (\delta_{Aix} \geq 0) & \rightarrow act_{Bi_y}; paint\_token; (3) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} \geq 0) \wedge (\delta_{Aix} < 0) & \rightarrow act_{Ai_x}; paint\_token; (4) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} < 0) & \rightarrow act_{Ai_x}; act_{Bi_y}; paint\_token; (5) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \mathcal{P}_B \wedge (\delta_{Aix} < 0) & \rightarrow act_{Ai_x}; act_{Bi_y}; paint\_token; (6) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} \geq 0) & \rightarrow act_{Ai_x}; paint\_token; (7) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \mathcal{P}_B \wedge (\delta_{Aix} \geq 0) & \rightarrow act_{Bi_y}; paint\_token; (8) \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} \geq 0) \wedge (\delta_{Aix} \geq 0) & \rightarrow paint\_token; (9)
\end{array}$$

for all $x \in \{1, \cdots, l_i\}$ and all $y \in \{1, \cdots, m_i\}$. Algorithm $\mathbb{A}^{\triangle s}\mathbb{B}$ is the union of $\mathcal{A}_i{}^{\triangle s}\mathcal{B}_i$ run by all processes in Π.

Sub-algorithm $\mathcal{A}_i{}^{\triangle s}\mathcal{B}_i$ is embedded in the critical section of the global mutual exclusion algorithm run by process P_i. As explained previously, the global mutual exclusion algorithm uses the spanning tree algorithm to function correctly. Each process has $l_i \cdot m_i$ guarded commands of Type 1. Each Type 1 guard corresponds to a guard of algorithms $\mathbb{A}$ and $\mathbb{B}$ each. A Type 1 guard is true 1) if corresponding guards of the component algorithms are true, 2) neither predicate $\mathcal{P}_A$ nor predicate $\mathcal{P}_B$ holds, and 3) the assignment statements lead to decrease of the respective ranking functions. A Type 2 guard is true if the corresponding guards of the component algorithms are true and both predicates hold. In the both cases respective assignment statements of algorithms $\mathbb{A}$ and $\mathbb{B}$ are executed. The guards of Type 3 and 4 are enabled when the safety predicates of both algorithms do not hold, and the corresponding assignment statement of one of the components does not lead to a decrease in the value of the respective ranking function. Process P_i executes the action of the component whose assignment statement leads to a decrease in the value of the ranking function if a Type 3 or Type 4 guard is enabled. Type 5 and 6 guards are true in a state if the safety predicate of one of the component holds while the safety predicate of the other component does not and, the assignment statement of the component whose safety predicate does not hold leads to a decrease in the value of respective ranking function. Assignment statesment of both the components are executed if a Type 5 or 6 guard is enabled. Type 7 and 8 guarded commands correspond to the scenarios where only one of the safety predicates hold but the guarded command of the component whose safety predicate does not hold does not lead to a decrease in the value of the ranking function; in this case, the action of the component whose safety predicate holds is executed. In case both the safety predicates do not hold and none of the components' assignment statements lead to a decrease in the value of the respective ranking functions, then process P_i simply updates the snapshot compartment of the communication register.

The composed algorithm might have progress inducing guarded commands of component algorithms in two different processes; however, concurrent execution can lead to inconsistent snapshots. Therefore, the composed algorithm is nested in the critical section of a single instance of the mutual exclusion algorithm. We now prove that the composition preserves the self-stabilization property of both component algorithms.

Lemma 6.6. *The projection of any maximal execution of composed algorithm $\mathbb{A}^{\triangle s}\mathbb{B}$ under a weakly fair scheduler on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$.*

Proof. Consider a prefix $\Xi_{|\neg\mathcal{P}_A}$ of a maximal execution of $\mathbb{A}^{\triangle s}\mathbb{B}$ such that the projection of any state on algorithm $\mathbb{A}$ in $\Xi_{|\neg\mathcal{P}_A}$ does not satisfy predicate $\mathcal{P}_A$. Since algorithm $\mathbb{A}$ converges to the states satisfying predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$, there exists at least one process whose enabled guarded command guarantees decrease in the value of ranking function Δ_A. Let process P_i be the only such process in the system. Either of a Type 1, Type 4, or Type 6 guarded command is enabled in process P_i. A scheduler can extend $\Xi_{|\neg\mathcal{P}_A}$ by selecting all other processes except process P_i. Clearly, projection of the suffix extended in this manner on scheduler $\mathbb{A}$ is not a maximal execution. However, such an execution of $\mathbb{A}\triangle_s\mathbb{B}$ will violate weak-fairness because process P_i remains continuously enabled without being ever selected. □

Lemma 6.7. *The projection of any maximal execution of the composed algorithm* $\mathbb{A}\triangle_s\mathbb{B}$ *under a weakly-fair scheduler on algorithm* $\mathbb{B}$ *is a maximal execution of algorithm* $\mathbb{B}$ *under scheduler* $\mathbb{D}_B$.

Proof. The proof of this lemma can be derived in the fashion similar to Lemma 6.6 by showing that a maximal execution of $\mathbb{A}\triangle_s\mathbb{B}$, whose projection on algorithm $\mathbb{B}$ is not a maximal execution of algorithm $\mathbb{B}$, does not fulfill weak-fairness constraint. □

Theorem 6.4. *Algorithm* $\mathbb{A}\triangle_s\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_A \wedge \mathcal{P}_B$ *under any weakly-fair scheduler.*

Proof. Convergence of algorithms $\mathbb{A}$ and $\mathbb{B}$ follows from Lemma 6.6 and Lemma 6.7, respectively. Since both safety predicates are assumed to be closed under any weakly-fair scheduler, the theorem follows. □

Symmetric Lifting Composition with κ-Local Mutual Exclusion

We now consider the case where both components require the κ-local mutual exclusion (κ-LME) algorithm for synchronization. Although symmetric lifting composition with the global mutual exclusion can be used to compose such algorithms, however –as argued earlier in this work– it delays the convergence by reducing concurrency of the system.

Let algorithms $\mathbb{A}$ and $\mathbb{B}$ require κ_1-local mutual exclusion and κ_2-local mutual exclusion respectively for continuous evaluation of the corresponding ranking functions. The result of κ-LME symmetric lifting composition of algorithms $\mathbb{A}$ and $\mathbb{B}$ is shown in Figure 6.1. Component sub-algorithms are modified in the manner described in Chapter 5; $\grave{\mathcal{A}}_i$ and $\grave{\mathcal{B}}_i$ represent the modified component sub-algorithms. The composed algorithm $\mathbb{A}\triangle_\kappa\mathbb{B}$ uses a single instance of the κ-local mutual exclusion algorithm for synchronization and evaluation of the ranking functions. The κ-mutual exclusion is slightly modified to facilitate the synchronization of two algorithms. Each process has two secondary clocks –$clock_{i2}$ and $clock_{i3}$– instead of a single secondary clock. Since every process constructs ordering of secondary clocks in its κ-neighborhood, two extra variables –res^2_{i1} and res^2_{i2} are also required. The composition operation binds the actions of a component sub-algorithm to one of the secondary clocks. More specifically, an action of sub-algorithm $\grave{\mathcal{A}}_i$ is executed only if process P_i is the process with a minimal $clock_{i2}$ in its κ_1 neighborhood. Similarly, process P_i executes a guarded command of $\grave{\mathcal{B}}_i$ if P_i has the minimal $clock_{i3}$ value in its κ_2 neighborhood. Essentially, each process has two critical sections –each corresponding to a component algorithm– whose entries are regulated by two different variables. Therefore, it is possible that process P_i may have privilege to enter only one of the two critical sections in a state. In such a scenario, process P_i updates the snapshot compartment of its communication registers notwithstanding which component sub-algorithm executes its guarded command. Both secondary clock variables are incremented modulo K_2. The larger of the two parameters κ_1 and κ_2 is used to select an appropriate value of K_2.

Remark 6.1. Note that κ-LME symmetric composition could have been implemented by using two instances of the κ-local mutual exclusion algorithm. However, such an implementation would use two primary clocks. Since the purpose of a primary clock variable is to maintain asynchronous unison in the system, two primary clocks introduce superfluous overhead without offering any added functionality.

```
process $P_i$
{
  localvar $clock_{i1}, clock_{i2}, clock_{i3} \in \mathbb{Z}$;
  localvar $res^1_{i1}, res^1_{i2}, res^2_{i1}, res^2_{i2} \in \mathbb{Z} \times \Pi$;
  const $\mathcal{N}_i \equiv \{P_j | P_j$ is neighbor of $P_i\}$
  macro $normalstep_{ix} \equiv (clock_{ix} \geq 0) \wedge (\forall_{j \in \mathcal{N}_i}\ (clock_{ix} = clock_{jx})$
                            $\vee (clock_{jx} = clock_{ix} + 1$ mod $[K_x]))$;
  macro $locallycorrect_{ix} \equiv (clock_{ix} \geq 0) \wedge (\forall_{j \in \mathcal{N}_i}\ (clock_{ix} = clock_{jx})$
                            $\vee (clock_{jx} = clock_{ix} + 1$ mod $[K_x])$
                            $\vee (clock_{ix} = clock_{jx} + 1$ mod $[K_x]))$;
  macro $reset_{ix} \equiv \neg locallycorrect_{ix} \wedge clock_{ix} \geq 0$;
  macro $nextstep_i \equiv normalstep_{i1} \wedge locallycorrect_{i2} \wedge locallycorrect_{i3}$;
  macro $convergestep_{ix} \equiv (clock_{ix} < 0) \wedge (\forall_{j \in \mathcal{N}_i}\ (clock_{ix} \leq 0) \wedge (clock_{ix} \leq clock_{jx}))$;
   $nextstep_i \rightarrow$ if $clock_{i1} = (\kappa_1)$mod$[\kappa_1 + 1]$  {
                  if $normalstep_{i2} \wedge (\langle i, clock_{i2} \rangle = res^1_{i2})$ {
                    $\grave{\mathcal{A}}_i$ ⟨⟨ modified component algorithm $\mathcal{A}_i$⟩⟩;
                    $clock_{i2} := (clock_{i2} + 1)$ mod $[K_2]$;  }
                 $res^1_{i1} := \langle clock_{i2}, i \rangle; res^1_{i2} := \langle clock_{i2}, i \rangle$;  }
                elseif {
                 $res^1_{i1} := res^1_{i2}; res^1_{i2} := \langle clock_{i2}, i \rangle \oplus res^1_{jy}, j \in \mathcal{N}_i$;  }
                 if $clock_{i1} = (\kappa_2)$ mod $[\kappa_2 + 1]$  {
                  if $normalstep_{i3} \wedge (\langle i, clock_{i3} \rangle = res^2_{i2})$ {
                    $\grave{\mathcal{B}}_i$ ⟨⟨ modified component algorithm $\mathcal{B}_i$⟩⟩;
                    $clock_{i3} := (clock_{i3} + 1)$ mod $[K_2]$;  }
                 $res^2_{i1} := \langle clock_{i3}, i \rangle; res^2_{i2} := \langle clock_{i3}, i \rangle$;  }
                elseif {
                 $res^2_{i1} := res^2_{i2}; res^2_{i2} := \langle clock_{i2}, i \rangle \oplus res^2_{jy}, j \in \mathcal{N}_i$;  }
                writestate() ⟨⟨write latest local state of modified components⟩⟩;
                $clock_{i1} := (clock_{i1} + 1)$ mod $[K_1]$;
  $\forall_{x \in \{1,2,3\}}$ [] $convergestep_{ix} \rightarrow clock_{ix} := (clock_{ix} + 1)$ mod $[K_x]$;
  $\forall_{x \in \{1,2,3\}}$ [] $reset_{ix} \rightarrow clock_{ix} := -\alpha_x$;
}
```

Fig. 6.1: Sub-algorithm $\mathcal{A}_i{}^{\triangle_\kappa}\mathcal{B}_i$

We now show that the self-stabilization properties of both the components are preserved by κ-LME symmetric composition. In particular, we show that the presence of two secondary clocks in a single instance of the κ-local mutual exclusion algorithm does not violate the consistency of the snapshots. We assume that $clock_{i2}$ regulates the access critical section corresponding to component $\mathcal{A}_i$ and $clock_{i3}$ manages the execution of the guarded commands of $\grave{\mathcal{B}}_i$.

Lemma 6.8. *The projection of any maximal execution of the composed algorithm $\mathbb{A}^{\triangle_\kappa}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$.*

Proof. We consider the two aspects of any maximal computation of composed algorithm $\mathbb{A}^{\triangle_\kappa}\mathbb{B}$: consistency of the snapshot that process P_i receives when it gets the privilege, and the effect of $clock_{i3}$ on the execution of $\mathbb{A}$.

Consider a state σ_ι in a maximal execution $\hat{\Xi}$ of $\mathbb{A}^{\triangle_\kappa}\mathbb{B}$ such that process P_i executes an enabled guarded command of algorithm $\grave{\mathcal{A}}_i$ exclusively in its κ_1 neighborhood for the second time in $\hat{\Xi}$. Let $clock_{i2}$ be equal to η_1 in σ_i. Since process P_i is the minimal process in its κ_1 neighborhood, process P_j

($P_j \in \mathcal{N}_i^{\kappa_1}$) does not execute sub-algorithm $\grave{\mathcal{A}}_j$ when $clock_{j2}$ is equal to η_1. Process P_j does not execute an action of $\grave{\mathcal{A}}_i$ while $\eta_1 - \kappa_1 < clock_{j2} < \eta_1 + \kappa_1$. A process can unilaterally increment its primary clock at most twice else its predicate $locallycorrect_{i1}$ would not hold. Thus, if process P_j ($\forall P_j \in \mathcal{N}_i^{\kappa_1}$) executed an action of sub-algorithm $\mathcal{A}_j$ in a state where $clock_{j2} < \eta_1 - \kappa_1$, then process P_i gets the updated local variables of sub-algorithm $\mathcal{A}_j$ in state σ_i.

Assume that there exists a process P_l ($P_l \in \mathcal{N}_i^{\kappa_1}$) such that $clock_{l1}$ is equivalent to $(\kappa_2)\,\mathbf{mod}\,[\kappa_2 + 1]$ and, $clock_{l3}$ is minimum in the κ_2-neighborhood of process P_l. Predicate $nextstep_l$ holds at process P_l; since $clock_{l3}$ is minimal, predicate $normalstep_{l3}$ holds as well. However, since process P_i is the minimal element in its κ_1-neighborhood in the phase corresponding to η_1 and $P_l \in \mathcal{N}_i^{\kappa_1}$, process P_l cannot execute any action of $\grave{\mathcal{A}}_i$ when $clocl_{l2}$ is equal to $clock_{i2}$. Let the distance between processes P_i and P_l be ς ($\varsigma \leq \kappa_1$). The value of $clock_{l1}$ cannot be equal to η_1 and lies either in the interval $[\eta_1 - \varsigma, \eta_1)$ or $(\eta_1 + \varsigma]$. Process P_l could have executed an action of $\grave{\mathcal{A}}_l$ $\varsigma - \kappa_1 - 1$ phases ago at the earliest. Nonetheless, process P_i has the updated value of the local variables of $\grave{\mathcal{A}}_l$ because process P_l cannot increment $clock_{l1}$ unilaterally. Additionally, when process P_l executes an action of sub-algorithm $\grave{\mathcal{B}}_l$ it does not enter the critical section corresponding to sub-algorithm $\mathcal{A}_l$. Therefore, process P_i has the correct values of the local variables of all the process in $\mathcal{N}_i^{\kappa_1}$ in global state σ_i even if one of them has simultaneous privilege to access the critical section corresponding to sub-algorithm $\mathcal{B}_j$. The lemma follows from the fact that an action of algorithm $\mathbb{A}$ cannot be delayed indefinitely by a weakly-fair scheduler. □

The following lemma can also be proven in an analogous manner to Lemma 6.8.

Lemma 6.9. *The projection of any maximal execution of the composed algorithm* $\mathbb{A}{\triangle_\kappa}\mathbb{B}$ *under any weakly-fair scheduler on algorithm* $\mathbb{B}$ *is a maximal execution of algorithm* $\mathbb{B}$ *under scheduler* $\mathbb{D}_B$.

The following theorem follows directly from Lemmata 6.8 and 6.9

Theorem 6.5. *Algorithm* $\mathbb{A}{\triangle_\kappa}\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_A \wedge \mathcal{P}_A$ *under any weakly-fair scheduler.*

Remark 6.2. Note that if κ_1 is equal to κ_2 then κ-LME symmetric lifting composition can achieved by a single critical section. Guarded commands of the composed algorithm can be constructed in the manner analogous to GME symmetric lifting composition. Modified guarded commands are then placed in the critical section of the κ-local mutual exclusion algorithm.

The κ-LME lifting composition can be used to compose self-stabilizing algorithms even if one of them requires global mutual exclusion for synchronization. Actions of the component requiring global mutual exclusion are executed when the primary clock is equal to D, where D is the diameter of the network. Let $\mathbb{A}$ refer to the component which requires global mutual exclusion and $\mathbb{B}$ the component for which κ_2-local mutual exclusion is sufficient. Then, the composed algorithm $\mathbb{A}{\triangle_\kappa}\mathbb{B}$ can be obtained by instantiating κ_1 to D. The following corollary can be inferred directly from Theorem 6.5.

Corollary 6.7. *Algorithm* $\mathbb{A}{\triangle_\kappa}\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_A \wedge \mathcal{P}_B$ *under any weakly-fair scheduler even if the modified component algorithm* $\grave{\mathbb{A}}$ *requires global mutual exclusion for synchronization.*

6.4 Lifting Composition with Variable Dependencies

The lifting composition and its variants have been defined for self-stabilizing algorithms which do not have any variable dependencies, implying that, none of the components reads the variables of its counterpart while evaluating its guards. We now consider the cases where composition is required to preserve the self-stabilization property of the component algorithms –in addition to the incompatible schedulers of the components– in presence of variable dependencies. To that end, we define the extensions of the lifting composition which cater to preserving the correctness of the components even if they read each others variables. However, the notion of variable dependency between two self-stabilizing algorithms need to be formulated before defining the desired compositional operators.

Definition 6.3 (Unidirectional Variable Dependency). *There exists a unidirectional variable dependency between algorithms $\mathbb{A}$ and $\mathbb{B}$ if* exactly one *of the following condition holds:*

- *for each process P_i, the local variables of sub-algorithm $\mathcal{B}_i$ appear either in the guards or on the right-hand side of the expressions in the assignment statements of sub-algorithm $\mathcal{A}_i$ or $\mathcal{A}_j$, where P_j is a neighbor of P_i.*
- *for each process P_i, the local variables of sub-algorithm $\mathcal{A}_i$ appear either in the guards or on the right-hand side of the expressions in the assignment statements of sub-algorithm $\mathcal{B}_i$ or $\mathcal{B}_j$, where process P_j is a neighbor of process P_i.*

Definition 6.4 (Bidirectional Variable Dependency). *There exists a bidirectional dependency between algorithms $\mathbb{A}$ and $\mathbb{B}$ if* both *of the following conditions hold:*

- *for each process P_i, the local variables of sub-algorithm $\mathcal{B}_i$ appear either in the guards or on the right-hand side of the expressions in the assignment statements of sub-algorithm $\mathcal{A}_i$ or $\mathcal{A}_j$, where process P_j is a neighbor of process P_i.*
- *for each process P_i, the local variables of sub-algorithm $\mathcal{A}_i$ appear either in the guards or on the right-hand side of the expressions in the assignment statements of sub-algorithm $\mathcal{B}_i$ or $\mathcal{B}_j$, where P_j is a neighbor of P_i.*

Note that the dependency relation between the algorithms does not allow the modification of local variables of a component algorithm by the other component. The dependency relation essentially partitions local variables of each component algorithm into two classes: immutable and mutable local variables. A variable is said to be immutable if a sub-algorithm can only read it, whereas mutable variable can be read as well as modified. Thus, a variable of component $\mathbb{A}$ becomes an immutable local variable of component $\mathbb{B}$ if it appears in the guards or the assignment expressions of $\mathbb{B}$.

6.4.1 Lifting Composition and Unidirectional Dependency

Since we intend to define compositional operators based on lifting composition, two scenarios need to be differentiated between while defining compositional operators for algorithms with unidirectional variable dependency. More specifically, assume that a unidirectional variable dependency exists between algorithms $\mathbb{A}$ and $\mathbb{B}$. It leads to two possibilities with respect to the direction of the dependency, either the variables of $\mathbb{A}$ appear in the guards or the assignment expressions of algorithm $\mathbb{B}$ or vice versa. It is, therefore, necessary to investigate both scenarios while defining the suitable variants of the lifting composition. We first present the asymmetric variant of the lifting composition for the algorithms with unidirectional variable dependency.

Asymmetric Lifting Composition with Unidirectional Dependency

Assume that the variables of sub-algorithm $\mathcal{A}_i$ appear in the guards or the assignment expressions of sub-algorithm $\mathcal{B}_i$ or $\mathcal{B}_j$ where process P_j is a neighbor of process P_i. Algorithm $\mathbb{A}$ self-stabilizes to predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$. Since there exists a unidirectional variable dependency between algorithm $\mathbb{A}$ and $\mathbb{B}$, we need to qualify the assumptions under which algorithm $\mathbb{B}$ self-stabilizes to predicate $\mathcal{P}_B$. The requirement to qualify the self-stabilization property of algorithm $\mathbb{B}$ stems from the fact that certain values of the immutable variables of $\mathbb{B}$ may not allow $\mathbb{B}$ to converge to the states satisfying predicate $\mathcal{P}_B$.

We, therefore, assume that algorithm $\mathbb{B}$ is “conditionally” self-stabilizing with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. More specifically, algorithm $\mathbb{B}$ converges to the states satisfying $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$ if its immutable variables – the variables of algorithm $\mathbb{A}$ which appear in the guards of algorithm $\mathbb{B}$– satisfy predicate $\mathcal{P}_A$. It follows immediately that algorithm $\mathbb{B}$ can only stabilize after algorithm $\mathbb{A}$ has converged to the states satisfying $\mathcal{P}_A$. However, since stabilization of $\mathbb{B}$ also depends on the underlying scheduler as well, any compositional operator defined for algorithms $\mathbb{A}$ and $\mathbb{B}$ must also account for potential incompatibility of the schedulers of the components. In the light of above discussion, the generalized version of asymmetric lifting composition can preserve the self-stabilization property of both components despite unidirectional variable dependency. This property of generic asymmetric lifting composition is formally proven by the following lemmata.

Lemma 6.10. *The projection of any maximal execution of algorithm* $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ *under a weakly-fair scheduler* $\mathbb{D}_A$ *over algorithm* $\mathbb{A}$ *is a maximal execution of* $\mathbb{A}$ *under* $\mathbb{D}_A$ *even if the local variables of* $\mathbb{A}$ *appear in the guards or the right-hand side assignment expressions of* $\mathbb{B}$.

Proof. The lemma follows directly from the fact that the guards of $\mathbb{A}$ are unchanged in the composed algorithm $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$. □

Lemma 6.11. *The projection of any maximal execution of algorithm* $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ *under a weakly fair scheduler* $\mathbb{D}_A$ *over algorithm* $\mathbb{B}$ *is a maximal execution of algorithm* $\mathbb{B}$ *under scheduler* $\mathbb{D}_B$ *even if the local variables of algorithm* $\mathbb{A}$ *appear in the guards or the right-hand side assignment expressions of algorithm* $\mathbb{B}$.

Proof. Consider a prefix $\varXi_{\neg\mathcal{P}_B}$ of a maximal execution of $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ such that no state in $\varXi_{\neg\mathcal{P}_B}$ satisfies predicate $\mathcal{P}_B$. Since actions of $\mathbb{A}$ are unhindered in $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ (Lemma 6.10) and algorithm $\mathbb{B}$ cannot converge to predicate $\mathcal{P}_B$ before predicate $\mathcal{P}_A$ holds, there exists a state in $\varXi_{\neg\mathcal{P}_B}$ after which predicate $\mathcal{P}_A$ holds and predicate $\mathcal{P}_B$ does not hold. Let $\sigma_{\mathcal{P}_A\wedge\neg\mathcal{P}_B}$ be such state and let $\varepsilon_{|\sigma_{\mathcal{P}_A\wedge\neg\mathcal{P}_B}}$ be suffix of $\varXi_{\neg\mathcal{P}_B}$ starting with $\sigma_{\mathcal{P}_A\wedge\neg\mathcal{P}_B}$. In any state of $\varepsilon_{|\sigma_{\mathcal{P}_A\wedge\neg\mathcal{P}_B}}$, there exists at least one process with an enabled guarded command $\grave{\mathcal{G}}_{B_i}$ because, algorithm $\mathbb{B}$ is self- stabilizing with respect to predicate $\mathcal{P}_B$ provided predicate $\mathcal{P}_A$ holds. Also, once predicate $\mathcal{P}_A$ holds in an execution it is not violated unless a transient fault occurs. An infinite extension of $\varepsilon_{|\sigma_{\mathcal{P}_A\wedge\neg\mathcal{P}_B}}$ in which scheduler $\mathbb{D}_A$ never selects the process with an enabled $\grave{\mathcal{G}}_{B_i}$ is not possible under $\mathbb{D}_A$, because that would violate scheduler $\mathbb{D}_A$ since it is weakly fair. □

The following theorem follows immediately from Lemmata 6.10 and 6.11.

Theorem 6.6. *Algorithm* $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ *is self-stabilizing with respect to predicate* $\mathcal{P}_A \wedge \mathcal{P}_B$ *under a weakly-fair scheduler* $\mathbb{D}_A$ *even if the variables of algorithm* $\mathbb{A}$ *appear in the guards or the right-hand side assignment expressions of algorithm* $\mathbb{B}$.

We now analyze the effect of asymmetric lifting composition if the direction of unidirectional variable dependency is reversed. Let the mutable variables of algorithm $\mathbb{B}$ appear in the guards of algorithm $\mathbb{A}$. As assumed above, algorithm $\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Algorithm $\mathbb{A}$, on the other hand, converges to the states satisfying predicate $\mathcal{P}_A$ under scheduler $\mathbb{D}_A$ if its immutable variables –the variables of $\mathbb{B}$ appearing in the guards of $\mathbb{A}$– satisfy predicate $\mathcal{P}_B$. It is also assumed that both algorithms have exactly one enabled guarded command per process in every state. We now show that the change in the direction of variable dependency does not not affect the result of the generic asymmetric lifting composition.

Lemma 6.12. *The projection of any maximal execution of algorithm* $\mathbb{A}{\scriptstyle\triangle\mathrm{G}}\mathbb{B}$ *under a weakly-fair scheduler* $\mathbb{D}_A$ *on algorithm* $\mathbb{B}$ *is a maximal execution of* $\mathbb{B}$ *under scheduler* $\mathbb{D}_B$ *even if the variables of* $\mathbb{B}$ *appear in the guards or the assignment expressions of* $\mathbb{A}$.

Proof. At least one process has an enabled guard $\grave{\mathcal{G}}_{B_i}$ in $\mathbb{A}\triangle_{G}\mathbb{B}$ because algorithm $\mathbb{B}$ converges to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$. Also, the mutual exclusion algorithm ensures that every process gets the token infinitely often in any execution; thus, the predicate *have_token* is infinitely often true in every process in $\mathbb{A}\triangle_{G}\mathbb{B}$. Consider a state σ_x in a maximal execution of $\mathbb{A}\triangle_{G}\mathbb{B}$ such that predicate $\mathcal{P}_B$ does not hold in global state σ_x. Since convergence of algorithm $\mathbb{A}$ is contingent on the variables of algorithm $\mathbb{B}$ satisfying predicate $\mathcal{P}_B$, state σ_x does not satisfies predicate $\mathcal{P}_A$ as well. As reasoned above, at least on process in state σ_x has a guarded command of algorithm $\grave{\mathbb{B}}$; assume that process P_i –the process with an enabled guarded command of algorithm $\grave{\mathbb{B}}$– possesses the token in state σ_x. As every process has an enabled guarded command of algorithm $\mathbb{A}$ in every state, process P_x has an enabled guarded command which allows execution of actions of both algorithms. Process P_x cannot be ignored indefinitely by scheduler $\mathbb{D}_A$ owing to the fact that $\mathbb{D}_A$ is weakly-fair. The lemma follows directly from the fact that algorithm $\mathbb{B}$ does not depend on algorithm $\mathbb{A}$ for convergence. □

Lemma 6.13. *The projection of any maximal execution of algorithm $\mathbb{A}\triangle_{G}\mathbb{B}$ under a weakly-fair scheduler $\mathbb{D}_A$ on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ even if the variables of algorithm $\mathbb{B}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{A}$.*

Proof. Note that algorithm $\mathbb{A}$ can converge to the states satisfying predicate $\mathcal{P}_A$ only after the projection of a state of $\mathbb{A}\triangle_{G}\mathbb{B}$ on algorithm $\mathbb{B}$ satisfies predicate $\mathcal{P}_B$. Let σ_y be the first state in a maximal execution of $\mathbb{A}\triangle_{G}\mathbb{B}$ such that predicate $\mathcal{P}_B$ holds. Consider the suffix $\varepsilon_{|\sigma_y}$ of the maximal execution of $\mathbb{A}\triangle_{G}\mathbb{B}$ starting with state σ_y. Actions of algorithm $\mathbb{A}$ in $\varepsilon_{|\sigma_y}$ are unhindered (by construction) and once predicate $\mathcal{P}_B$ is not violated unless a transient fault occurs. □

The following theorem is a direct consequence of Lemmata 6.12 and 6.13.

Theorem 6.7. *Algorithm $\mathbb{A}\triangle_{G}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under a weakly-fair scheduler $\mathbb{D}_A$ even if the variables of algorithm $\mathbb{A}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{B}$.*

Symmetric Lifting Composition under Unidirectional Dependency

Having shown that the asymmetric lifting composition preserves the self-stabilization property of the component algorithms despite unidirectional variable dependencies, we next analyze the result of the symmetric lifting composition of self-stabilizing algorithms.

Consider the case where the variables of algorithm $\mathbb{A}$ are used in the guards of algorithm $\mathbb{B}$. Since convergence of algorithm $\mathbb{B}$ is contingent on the convergence of algorithm $\mathbb{A}$, any execution of algorithm $\mathbb{B}$ till $\mathbb{A}$ converges will not reach the states satisfying predicate $\mathcal{P}_B$. Also, actions of algorithm $\mathbb{A}$ in $\mathbb{A}\triangle_{S}\mathbb{B}$ are unaffected by the truth values of the guards of algorithm $\mathbb{B}$. The proof of the following lemma is similar to the proof of Lemma 6.6.

Lemma 6.14. *The projection of any maximal execution of algorithm $\mathbb{A}\triangle_{S}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{A}$ is a maximal execution of $\mathbb{A}$ under scheduler $\mathbb{D}_A$ even if the variables of $\mathbb{A}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{B}$.*

Lemma 6.15. *The projection of any maximal execution of algorithm $\mathbb{A}\triangle_{S}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{B}$ is a maximal execution of $\mathbb{B}$ under scheduler $\mathbb{D}_B$ even if the variables of $\mathbb{A}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{B}$.*

Proof. Actions of algorithm $\mathbb{A}$ are unaffected by the composition operation (from Lemma 6.14). Let σ_x be the state in a maximal execution where predicate $\mathcal{P}_A$ holds and predicate $\mathcal{P}_B$ does not hold. Predicate $\mathcal{P}_A$ remains true in any execution suffix which starts in state σ_x. Rest of the proof is similar to Lemma 6.7. □

The theorem below follows directly from Lemmata 6.14 and 6.15.

Theorem 6.8. *Algorithm $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under any weakly-fair scheduler even if the variables of algorithm $\mathbb{A}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{B}$.*

The correctness of symmetric generic lifting composition is now analyzed under assumption that the variables of algorithm $\mathbb{B}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{B}$. As is the case with the asymmetric lifting composition, the symmetric variant of the generic lifting composition preserves the self-stabilization properties of component algorithms. We provide the proof of the lemma central to the correctness proof.

Lemma 6.16. *The projection of any maximal execution of algorithm $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{B}$ is a maximal execution of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$ even if the variables of algorithm $\mathbb{B}$ appear in the guards or the assignment expressions of algorithm $\mathbb{A}$.*

Proof. As a result of self-stabilization of the algorithms providing mutual exclusion, every process gets the token infinitely often in any maximal execution of $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$. There exists at least one process which has an enabled Type 3 guarded command as algorithm $\mathbb{B}$ is self- stabilizing with respect to predicate $\mathcal{P}_B$. Note that algorithm $\mathbb{A}$ cannot converge to predicate $\mathcal{P}_A$ in a suffix of a maximal execution of $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ unless predicate $\mathcal{P}_B$ holds in the prefix. A weakly-fair scheduler cannot starve the process with an enabled Type 3 guarded command indefinitely. □

The proof of the following lemma is similar to Lemma 6.14 by dissecting maximal executions of $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ after a state satisfying predicate $\mathcal{P}_B$ is reached.

Lemma 6.17. *The projection of any maximal execution of algorithm $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{A}$ is a maximal execution of algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ even if the variables of variables $\mathbb{B}$ appear in the guards or the assignment expressions of algorithm $\mathbb{A}$.*

Theorem 6.9. *Algorithm $\mathbb{A}^{\vartriangle}\mathrm{s}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under any weakly-fair scheduler even if the variables of algorithm $\mathbb{B}$ appear in the guards or the right-hand side assignment expressions of algorithm $\mathbb{A}$.*

We have, thus, shown that the direction of unidirectional variable dependency does not affect the result of the lifting composition. The following corollary can be directly deduced from Theorems 6.6, 6.7, 6.8, and 6.9.

Corollary 6.8. *If algorithm $\mathbb{A}$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ and algorithm $\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_B$, then the result of the generic lifting composition of algorithms $\mathbb{A}$ and $\mathbb{B}$ is self-stabilizing with respect to the predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ even if there exits unidirectional variable dependency between algorithms $\mathbb{A}$ and $\mathbb{B}$.*

Discussion

It is in order here to remark that the lifting composition in presence of unidirectional dependency is similar to the hierarchical composition. Recall that the hierarchical composition [81] is defined over the self-stabilizing algorithms having unidirectional variable dependency. However, unlike the hierarchical composition, the lifting composition does not put restrictions on the execution semantics of the component algorithms. Assume that algorithm $\mathbb{B}$ requires the variables of algorithm $\mathbb{A}$ to satisfy predicate $\mathcal{P}_B$ in order to stabilize. Hierarchical composition would require that the actions of algorithm $\mathbb{B}$ are suspended till algorithm $\mathbb{A}$ converges to predicate $\mathcal{P}_A$ and, subsequently the actions of algorithm $\mathbb{A}$ are suspended. This requirement restricts the class of self-stabilizing algorithms which can be constructed using the hierarchical composition. More specifically, if algorithm $\mathbb{A}$ is a non-silent algorithm then, the closure property of algorithm $\mathbb{A}$ is not guaranteed by the hierarchical composition because, the hierarchical composition requires that the actions of algorithm $\mathbb{A}$ must be suspended once it satisfies predicate $\mathcal{P}_A$. The lifting composition, on the other hand, only assumes that closure properties of algorithms with respect to their respective safety predicates are preserved. The hierarchical composition also implicitly assumes that the (conditional) self-stabilization of component algorithms require schedulers which are compatible. On the contrary, the lifting composition preserves the self-stabilization property of component algorithms despite incompatibility of the respective schedulers.

6.4.2 Lifting Composition and Bidirectional Dependency

We now define a compositional operator for the scenario where 1) bidirectional variable dependency exists between the component algorithms and, 2) the respective schedulers are incompatible. While the generic lifting composition without modification is able to preserve the self-stabilization properties of algorithms with unidirectional dependency, bidirectional dependency requires enhanced lower layer support because the system may start in a state where no component satisfies its respective safety predicate.

Consider algorithms $\mathbb{A}$ and $\mathbb{B}$ such that the mutable variables of algorithm $\mathbb{A}$ appear in the guards and/or the right-hand side assignment expressions of algorithm $\mathbb{B}$ and vice versa. Algorithms $\mathbb{A}$ and $\mathbb{B}$ also have the following properties. Algorithm $\mathbb{A}$ converges to the states where predicate $\mathcal{P}_A$ holds if the immutable variables of algorithm $\mathbb{A}$ satisfy predicate $\mathcal{P}_B$; similarly, algorithm $\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_B$ under scheduler $\mathbb{D}_B$ if its immutable variables satisfy predicate $\mathcal{P}_A$. Additionally, the safety predicate of algorithm $\mathbb{A}$ ($\mathbb{B}$) is not violated in any maximal execution of algorithm $\mathbb{A}$ ($\mathbb{B}$) once it reaches a state which satisfies predicate $\mathcal{P}_A$ ($\mathcal{P}_B$). Assume that algorithms $\mathbb{A}$ and $\mathbb{B}$ are composed using the generic asymmetric lifting composition and let σ_x be the state where neither of the algorithms satisfies its respective safety predicate. σ_x is a deadlock state because each component can only converge if the safety predicate of its counterpart holds. Therefore, an extraneous mechanism to break the deadlock is required. There are two alternatives with respect to the additional corrective mechanism: repairing and resetting. A repair operation [95, 123] typically involves saving local history at each process, and reconstructing an execution based on the local histories stored at the constituent processes. A reset operation [97], on the other hand, sets a distributed system to a predefined legal state after an inconsistency is detected. We opt for a reset operation because a repairing mechanism is memory- intensive and delays the convergence of the composed algorithm. A reset mechanism, contrastingly, requires only a spanning tree to function correctly, and therefore, does not require any extra algorithmic support, because the algorithm obtained by the lifting composition constructs a spanning tree. In the sequel we assume that the mutable variables of each algorithm can be reset to a predefined state which satisfies its respective safety predicate.

Reset Operation

The communication registers of the constituent processes are extended to support the functioning of the reset operation. Since we need to monitor the execution of both the components, each process writes the latest values of the variables of algorithm $\mathbb{A}$ to its communication registers. The reset operation is implemented with the help of the wave-based reset algorithm of [97]. It is integrated in the generic lifting composition by adding an extra layer. The layered structure of the modified algorithm is shown in Figure 6.2. The wave-based reset algorithm due to Arora and Gouda [97] uses two variables to set the variables to predefined values; variable flag_i indicates the phase in which the reset algorithm is in, and variable serial_i tracks the number of reset requests serviced in any epoch. The algorithm needs a distinguished process and a spanning tree to function correctly, and therefore, a reset layer is positioned above the spanning tree layer. The reset algorithm is initiated when a process P_i sets the variable *flag* to *initiate*. As the reset operation needs to be initiated by an algorithm that monitors the system state, the reset layer is positioned below the mutual exclusion layer. The reset algorithm works in three phases.

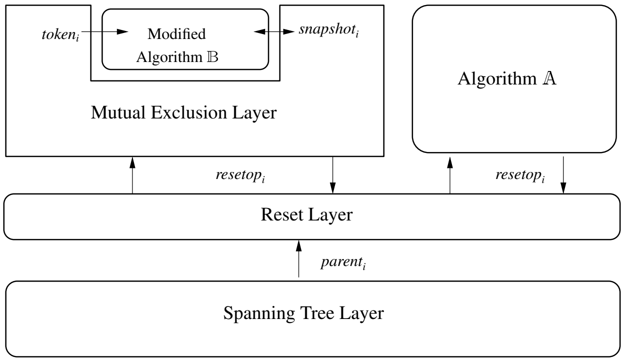

Fig. 6.2: Layered View of the Lifting Composition with Bidirectional Dependency

The first phase is started when a process P_i –on detecting an “anomaly”– sends a request for reset to the root of the spanning tree. The request is forwarded to the root process and, while doing so, requests initiated by other process are merged into a single request. The second phase of the algorithm begins when the request reaches the root process and the root process starts a reset wave by setting $flag_{\text{root}}$ to *reset*. Each process sets its variables to a predefined state when the $flag_j$ variable of its parent is equal to *reset*. The third phase starts when the the leaf processes of the spanning tree reset their variables and set their $flag_i$ variables to *nrml*. The algorithm ends when the completion wave reaches the root process.

Asymmetric Lifting Composition with Bidirectional Dependency

Definition 6.5 (Asymmetric lifting Composition with Bidirectional Dependency). *Sub-algorithm $\mathcal{A}_i{\triangle_{\mathbf{B}}}\mathcal{B}_i$ consists of $l_i \cdot m_i + 1$ guarded commands of the following structure:*

$$\begin{array}{lll}
\neg\mathcal{P}_A \wedge \neg\mathcal{P}_B & \rightarrow reset\_action(\mathbb{B}); & (1) \\
\mathcal{G}_{Ai_p} \wedge have\_token_i \wedge (flag_i = nrml) & \rightarrow \left\{\begin{array}{l} \grave{\mathcal{G}}_{Bi_1} \rightarrow act_{Ai_p}; act_{Bi_1}; paint\_token; \\ \quad\vdots \\ \grave{\mathcal{G}}_{Bi_\eta} \rightarrow act_{Ai_p}; act_{Bi_\eta}; paint\_token; \end{array}\right\} & (2) \\
\mathcal{G}_{Ai_p} \wedge \neg have\_token_i \wedge (flag_i = nrml) & \rightarrow act_{Ai_p}; token\_op; & (3)
\end{array}$$

for all $q \in \{1, \cdots, l_i\}$. Algorithm $\mathbb{A}{\triangle_{\mathbf{B}}}\mathbb{B}$ is the union of all sub-algorithms $\mathcal{A}_i{\triangle_{\mathbf{B}}}\mathcal{B}_i$

$$\mathbb{A}{\triangle_{\mathbf{B}}}\mathbb{B} = \bigcup_{\forall i} \mathcal{A}_i{\triangle_{\mathbf{B}}}\mathcal{B}_i$$

run by the processes in Π.

Algorithm $\mathbb{A}{\triangle_{\mathbf{B}}}\mathbb{B}$ has an additional guarded command compared to $\mathbb{A}{\triangle_{\mathbf{G}}}\mathbb{B}$. The Type 1 guarded command is enabled if none of the safety predicates hold in the system and, the process with enabled Type 1 guard initiates a request wave to reset the variables of algorithm $\mathbb{A}$. The rest of the guarded commands are similar to the guarded commands of algorithm $\mathbb{A}{\triangle_{\mathbf{G}}}\mathbb{B}$ except for the extra conjunctive term in the guards. An assignment statement of algorithm $\mathbb{A}$ is executed if the respective guard is true and no reset wave is in operation. A process executes an action of algorithm $\mathbb{B}$ if it decreases the value of ranking function $\varDelta_B$ and no reset wave is in operation. Note that the wave-based reset algorithm of [97] is a self-stabilizing algorithm.

We now show that self-stabilization properties of both components are preserved by $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$.

Theorem 6.10 (based on [97]). *Irrespective of the initial state, the reset algorithm is guaranteed to eventually reach a state where, for all $P_i \in \Pi$, $flag_i$ is not equal to request and $serial_i$ is equal to η where η is an arbitrary integer.*

Lemma 6.18. *The projection of any maximal execution of $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$ on algorithm $\mathbb{B}$ has a suffix which satisfies predicate $\mathcal{P}_B$.*

Proof. Consider the projection $\breve{\Xi}_{|\mathbb{B}}$ of a maximal execution of $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$ on algorithm $\mathbb{B}$. Let $\sigma_i(\sigma_{i|\mathbb{B}})$ be the first state of $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}(\breve{\Xi}_{|\mathbb{B}})$.
Assume that predicate $\mathcal{P}_A$ holds in state σ_i. Since algorithm $\mathbb{A}$ under scheduler $\mathbb{D}_A$ does not violate predicate $\mathcal{P}_A$ once it holds and, the actions of algorithm $\mathbb{A}$ are unchanged in $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$, Type 1 guarded command is never enabled in $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}$. As predicate $\mathcal{P}_A$ holds, the value of variable *$flag_i$* is equal to *nrml* in all processes in the system. As only Type 2 and 3 guards are enabled in the system, the projection of $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}$ on algorithm $\mathbb{B}$ is a maximal execution of algorithm $\mathbb{B}$ under scheduler $\mathbb{D}_B$ (from Lemma 6.5). $\breve{\Xi}_{|\mathbb{B}}$ reaches a state satisfying predicate $\mathcal{P}_B$ because predicate $\mathcal{P}_A$ is not violated in $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}$.
We now consider the case where neither predicate $\mathcal{P}_A$ nor predicate $\mathcal{P}_B$ holds in state σ_i. Since each process collects the snapshot of the variables of algorithm $\mathbb{A}$ as well, eventually a process in the system will detect that predicate $\mathcal{P}_A$ does not hold. The process which detects that predicate $\mathcal{P}_A$ does not hold will request a reset as guard of Type 1 will be enabled at that process. The lemma follows immediately because, the reset algorithm is guaranteed to reach state where the variables of algorithm $\mathbb{B}$ are set to the values which satisfy predicate $\mathcal{P}_B$ (Theorem 6.10). □

Lemma 6.19. *The projection of any maximal execution of $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$ on algorithm $\mathbb{A}$ has a suffix which satisfies predicate $\mathcal{P}_A$.*

Proof. Consider the first state σ_i of a maximal execution $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}$ of $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$. If neither predicate $\mathcal{P}_A$ nor predicate $\mathcal{P}_B$ holds in state σ_i then eventually a reset wave will be started by some process in the system; predicate $\mathcal{P}_B$ will hold at the completion of the wave-based reset algorithm (Theorem 6.10). Let σ_j be the first state in $\hat{\Xi}_{\mathbb{A}\vartriangle_{\mathrm{B}}\mathbb{A}}$ where predicate $\mathcal{P}_B$ holds. The actions of algorithm $\mathbb{A}$ are unhindered in any suffix starting with state σ_j (by construction). The proof is completed by the fact that algorithm $\mathbb{A}$ converges to predicate $\mathcal{P}_A$ under $\mathbb{D}_A$. □

The following theorem is an immediate consequence of Lemmata 6.18 and 6.19.

Theorem 6.11. *Algorithm $\mathbb{A}{\vartriangle_{\mathrm{B}}}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under any weakly-fair scheduler $\mathbb{D}_A$.*

Symmetric Lifting Composition with Bidirectional Dependency

We now define the symmetric variant of the lifting composition that preserves the self-stabilization properties of component algorithms under any weakly-fair scheduler even if bidirectional dependency exists between the component algorithms.

Definition 6.6 (Symmetric Lifting Composition with Bidirectional Dependency). *Sub-algorithm $\mathcal{A}_i{\vartriangle_{\beta}}\mathcal{B}_i$ consists of $5 \cdot l_i \cdot m_i + 1$ guarded commands of the following structure:*

$$\begin{array}{ll}
\neg\mathcal{P}_A \wedge \neg\mathcal{P}_B & \rightarrow \mathit{reset\_action}(\mathbb{B}); \quad \mathit{(1)} \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \mathcal{P}_B \wedge (\mathit{flag}_i = \mathit{nrml}) & \rightarrow \mathit{act}_{Ai_x}; \mathit{act}_{Bi_y}; \mathit{paint\_token}; \mathit{(2)} \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} < 0) \wedge (\mathit{flag}_i = \mathit{nrml}) & \rightarrow \mathit{act}_{Ai_x}; \mathit{act}_{Bi_y}; \mathit{paint\_token}; \mathit{(3)} \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \mathcal{P}_B \wedge (\delta_{Aix} < 0) \wedge (\mathit{flag}_i = \mathit{nrml}) & \rightarrow \mathit{act}_{Ai_x}; \mathit{act}_{Bi_y}; \mathit{paint\_token}; \mathit{(4)} \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \mathcal{P}_A \wedge \neg\mathcal{P}_B \wedge (\delta_{Biy} \geq 0) \wedge (\mathit{flag}_i = \mathit{nrml}) & \rightarrow \mathit{act}_{Ai_x}; \mathit{paint\_token}; \mathit{(5)} \\
\mathcal{G}_{Ai_x} \wedge \mathcal{G}_{Bi_y} \wedge \neg\mathcal{P}_A \wedge \mathcal{P}_B \wedge (\delta_{Aix} \geq 0) \wedge (\mathit{flag}_i = \mathit{nrml}) & \rightarrow \mathit{act}_{Bi_y}; \mathit{paint\_token}; \mathit{(6)}
\end{array}$$

for all $x \in \{1, \cdots, l_i\}$ and all $y \in \{1, \cdots, m_i\}$. Algorithm $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$ is the union of $\mathcal{A}_i{\vartriangle_\beta}\mathcal{B}_i$ run by all processes in Π.

The guarded commands of algorithm $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$ are embedded in the critical section of the mutual exclusion algorithm, unlike $\mathbb{A}{\vartriangle_{\mathbf{B}}}\mathbb{B}$ where only modified guards of algorithm $\mathbb{B}$ are embedded in the critical section. Nevertheless, a reset layer is common to both the compositional operators. If none of the safety predicates hold in a state, then Type 1 guard is enabled, and the mutable variables of algorithm $\mathbb{B}$ are set to the values which satisfy predicate $\mathcal{P}_B$. The rest of the guarded commands are enabled only if no reset request is being processed. In a state where predicate $\mathcal{P}_A$ does not hold actions of algorithm $\mathbb{A}$ are executed only if they guarantee a decrease in the value of ranking function Δ_A (Type 4 guarded command); similarly actions of $\mathbb{B}$ in a state where predicate $\mathcal{P}_B$ does not hold are executed if they lead to a decrease in the value of ranking function Δ_B (Type 3 guarded command).

The proof of the following lemma is similar to the proof of Lemma 6.18.

Lemma 6.20. *The projection of any maximal execution of algorithm $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{B}$ has a suffix which satisfies predicate $\mathcal{P}_B$.*

Lemma 6.21. *The projection of any maximal execution of algorithm $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$ under any weakly-fair scheduler on algorithm $\mathbb{A}$ has a suffix which satisfies predicate $\mathcal{P}_B$.*

Proof. Assume that neither predicate $\mathcal{P}_A$ nor predicate $\mathcal{P}_B$ holds in the first state of a maximal execution of $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$. Eventually at least one process will detect that predicate $\mathcal{P}_A$ does not hold and, the process start a reset wave. Predicate $\mathcal{P}_B$ would hold at the termination of the reset algorithm. Rest of the proof is analogous to Lemma 6.6. □

We get the following theorem directly from Lemmata 6.18 and 6.19.

Theorem 6.12. *Algorithm $\mathbb{A}{\vartriangle_\beta}\mathbb{B}$ is self-stabilizing with respect to predicate $\mathcal{P}_A \wedge \mathcal{P}_B$ under any weakly-fair scheduler.*

6.5 Summary

We extended scheduler-oblivious transformation to handle the case where the transformed algorithm runs under read/write atomicity or distributed scheduler. In addition, an extension of the transformation method for the message passing model was proposed. The scheduler-oblivious transformation method was mated with the lifting composition to devise a generic compositional scheme that preserves the self-stabilization properties of the component algorithms despite incompatible schedulers. It was also shown that the generic lifting composition preserves self-stabilization even if convergence of one of the components is a prerequisite for the convergence of other component. We presented another variant of the generic lifting composition extending the scope of our compositional framework to the self-stabilizing algorithms with bidirectional variable dependencies. Figure 6.3 summarizes the various compositional operators defined in this chapter along with the conditions under which algorithm designers may use them.

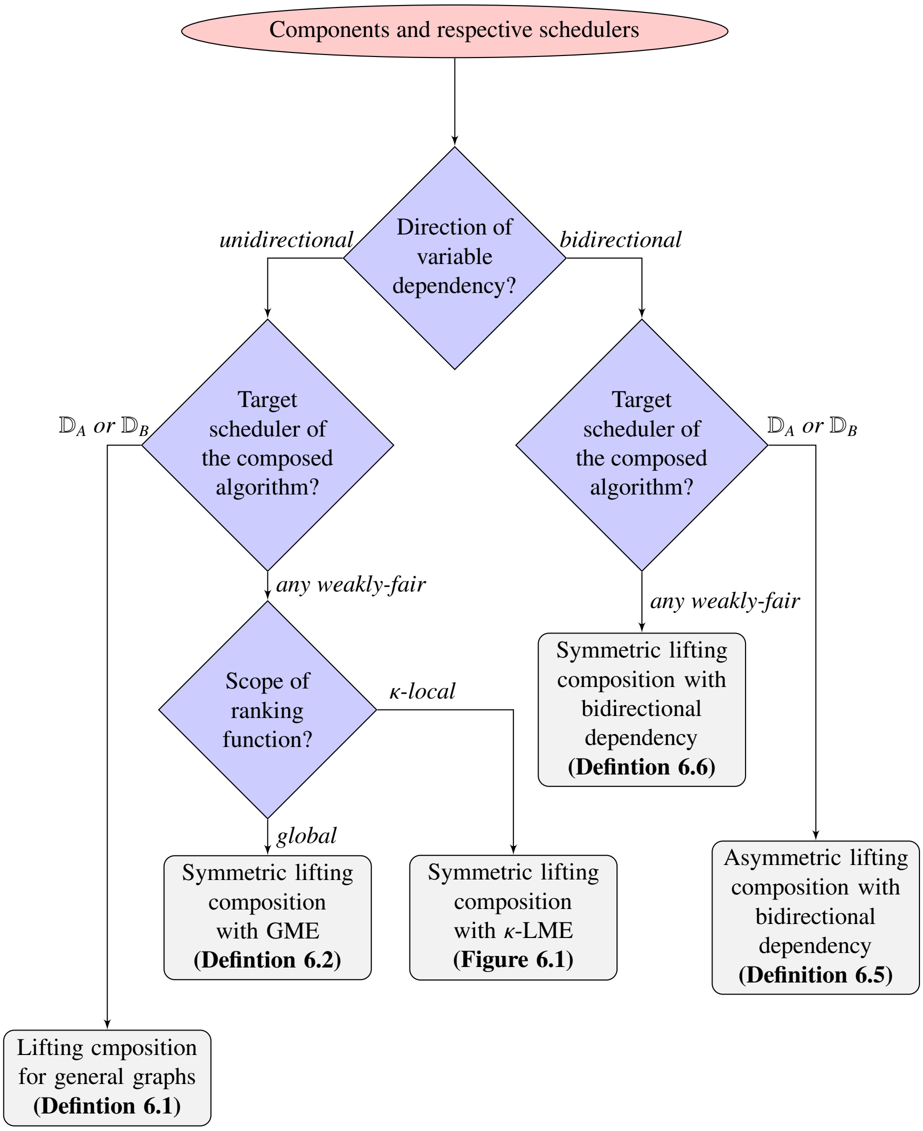
Components and respective schedulers
Direction of variable dependency?
unidirectional
bidirectional
Target scheduler of the composed algorithm?
Target scheduler of the composed algorithm?
$\mathbb{D}_A$ or $\mathbb{D}_B$
$\mathbb{D}_A$ or $\mathbb{D}_B$
any weakly-fair
any weakly-fair
Scope of ranking function?
κ-local
global
Symmetric lifting composition with bidirectional dependency (Defintion 6.6)
Symmetric lifting composition with GME (Defintion 6.2)
Symmetric lifting composition with κ-LME (Figure 6.1)
Asymmetric lifting composition with bidirectional dependency (Definition 6.5)
Lifting cmposition for general graphs (Defintion 6.1)

Fig. 6.3: Concise Summary of the Compositional Framework

7

Conclusion

We summarize the contributions of this dissertation in the scope of designing large self-stabilizing algorithms. Before we conclude, we also discuss the potential extensions of the results presented in this work.

7.1 Summary

Essentially, this work delivers a set of compositional methods for self-stabilizing algorithms. A compositional method can be selected based on the properties of the self-stabilizing algorithms acting as the building blocks. The suite of the compositional methods distinguishes itself from the hitherto proposed compositional methods in that it allows composition of distributed algorithms whose self-stabilization require mutually incompatible assumptions. Another distinguishing feature of the compositional framework is the potential of automatization of the composition operation. The potential of automatization owes to the fact that the guards of component algorithms appear unchanged in the guards of the composed algorithm. Moreover, auxiliary constructs used to deal with the incompatible proof assumptions of the component algorithms do not depend on the component algorithms.

We began the quest of defining "oblivious" composition methods by asking whether two self-stabilizing algorithms can be composed –without compromising the respective self-stabilization properties– if the respective self-stabilization proofs require different schedulers. The question was answered positively, and the skeleton of the generic composition method, lifting composition, was devised in the process. It was realized that the constraints imposed by the respective schedulers of the component algorithms can be overcome by using the information culled from the respective proofs of self-stabilization. We showed that the use of a ranking function – a by-product of the self-stabilization proof of a distributed algorithm– in the guards of the composed algorithm preserves the stabilization property of a component algorithm under the scheduler of its counterpart. However, the evaluation of ranking function requires a degree of synchronization between the processes implementing the composed algorithm.

In order to pave the way for the implementation of lifting composition, we proposed a transformation method which preserves the self-stabilization of a distributed algorithm under any weakly-fair scheduler. The method uses a spanning tree algorithm and a mutual exclusion algorithm to facilitate the coordination required for the evaluation of a ranking function. However, the coordination between the processes leads to an increase in the convergence time of the transformed algorithm. The increase in convergence time is unavoidable because coordination is required for the correctness of the transformed algorithm. Nevertheless, we showed that the convergence of the transformed algorithm can be reduced by exploiting the structure of the ranking function supplied by the self-stabilization proof. More specifically, we proved that a κ-local mutual exclusion algorithm is sufficient for coordination among processes if evaluation of a ranking function at every process requires only κ-hops information. The benefits of using the local mutual exclusion algorithm for coordination, and the importance of using a local ranking function became apparent as system size was increased during the simulations. Another feature of the transformation method is its modular nature which allows usage of suitable spanning tree and mutual exclusion algorithms because the transformation method is not bound to a specific auxiliary algorithm.

Lifting composition and scheduler-oblivious transformation form the foundation of the compositional framework. The scheduler-oblivious transformation and lifting composition were mated to define the generic lifting composition. As a result we had two variants of lifting composition; the first method uses general mutual exclusion and the other is based on local mutual exclusion. Symmetric variants of the generic lifting composition were defined to preserve the self-stabilization of component algorithms under any weakly-fair scheduler. It was also proven that the composed algorithm is self-stabilizing under any distributed scheduler even though component algorithms themselves might be self-stabilizing only under a sequentialized scheduler. In order to extend the scope of the framework, generic lifting composition was used as a kernel to define composition operation over the algorithms that have bidirectional variable dependencies. Thus, the compositional framework provides the methods which allow system designers to select component algorithms –without being constrained by potentially incompatible proof assumptions or variable dependencies– solely based on the functional requirements.

7.2 Outlook

The framework presented in this dissertation relies on the scheduler-oblivious transformation method to compose self-stabilizing algorithms with disparate schedulers. The transformation method itself uses the proof artifacts of the component algorithms to monitor the execution of the composed algorithm. The scheduler transformation method can also be viewed as a method to transform a non-self-stabilizing algorithm into a self-stabilizing algorithm because, prior to the transformation, a scheduler –stronger than the one assumed during the stabilization proof– can generate an execution that never reaches the states satisfying the safety predicate. In contrast to the earlier methods of transforming a non-self-stabilizing algorithms to a self-stabilizing algorithm with help of reset operations [95, 97], our approach exploits the fact that a use algorithm might exhibit convergence under a restricted scheduler. As a result, unlike the methods based on reset operations, the scheduler transformation method does not need to store the complete execution history. Each approach, however, has associated costs, namely, the second approach requires history to be stored at each process and the time required to reset the system. The convergence time of an algorithm becomes larger if the first approach is used. There is, hence, a trade-off involved between the methods. Therefore, it would be worthwhile to compare the two approaches. More specifically, we need to map the above two methods to the scenarios where one of the two methods is superior to the other. In particular, the performance of the two methods should be measured in terms of 1) memory usage, 2) message complexity, and 3) time required to converge to the safety predicate. If neither of the two methods outperforms the other method in all possible scenarios, then we need to investigate, whether the state space of a transformed algorithm can be demarcated into regions where one of the two methods outperforms the other method. A positive answer to this problem would pave the way for "scheduler-adaptive" self-stabilizing algorithms. Scheduler-adaptivity in the context of self-stabilizing algorithms would ensure that a self-stabilizing algorithm converges to the set of safe states in an optimal number of steps despite an adversarial or easily realizable scheduler. Thus, a scheduler-adapter would relieve system designers of the burden of optimizing the performance and proving the correctness of self-stabilizing algorithms in all possible implementation scenarios.

References

[1] Anish Arora et al. "Project ExScal (Short Abstract)". In: *First IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*. Ed. by Viktor K. Prasanna et al. Vol. 3560. Lecture Notes in Computer Science. Springer, 2005, pp. 393–394 (cit. on p. 1).

[2] Felix C. Gärtner. "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments". In: *ACM Comput. Surv.* 31.1 (1999), pp. 1–26 (cit. on pp. 1, 19, 20).

[3] Edsger W. Dijkstra. "Self-stabilizing Systems in Spite of Distributed Control". In: *Communications of ACM* 17.11 (1974), pp. 643–644 (cit. on pp. 1, 10, 18).

[4] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1 (2003), pp. 41–50 (cit. on p. 1).

[5] Ian F. Akyildiz et al. "A Survey on Sensor Networks". In: *IEEE Communications Magazine* 40.8 (2002), pp. 102–116 (cit. on p. 1).

[6] Fabrice Theoleyre and Fabrice Valois. "About the Self-stabilization of a Virtual Topology for Self-Organization in Ad Hoc Networks". In: *Self-Stabilizing Systems*. Ed. by Ted Herman and Sébastien Tixeuil. Lecture Notes in Computer Science 3764. Springer-Verlag, 2005, pp. 214–228 (cit. on p. 2).

[7] Hongwei Zhang and Anish Arora. "GS^3: Scalable Self-Configuration and Self-Healing in Wireless Networks". In: *Proceedings of the twenty-first annual symposium on Principles of Distributed Computing (PODC)*. ACM, 2002, pp. 58–67 (cit. on p. 2).

[8] Emmanuelle Anceaume et al. "Towards a Theory of Self-Organization". In: *Nineth International Conference on Principles of Distributed Systems (OPODIS)*. Ed. by James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer. Vol. 3974. Lecture Notes in Computer Science. Springer, 2005, pp. 191–205 (cit. on p. 2).

[9] Olga Brukman et al. "Self-Stabilization as a Foundation for Autonomic Computing". In: *IEEE ARES 2007 Workshop on Foundation of Fault-tolerance Distributed Computing*. 2007 (cit. on p. 2).

[10] Andrew Berns and Sukumar Ghosh. "Dissecting Self-* Properties". In: *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE Computer Society, 2009, pp. 10–19 (cit. on p. 2).

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. ISBN 0-262-02649-X. The MIT Press, 2008, p. 975 (cit. on p. 2).

[12] Mohamed G. Gouda, Rodney R. Howell, and Louis E. Rosier. "The Instability of Self-Stabilization". In: *Acta Informatica* 27.8 (1989), pp. 697–724 (cit. on p. 2).

[13] Abhishek Dhama and Oliver Theel. "A Tranformational Approach for Designing Scheduler-Oblivious Self-stabilizing Algorithms". In: *Proceedings of 12th International Symposium on*

Stabilization, Safety, and Security of Distributed Systems (SSS). Ed. by Shlomi Dolev et al. Vol. 6366. Lecture Notes in Computer Science. Springer, 2010, pp. 80–95 (cit. on p. 3).

[14] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2008. ISBN: 0-470-12872-0 (cit. on p. 5).

[15] Romit Roy Choudhury and Nitin H. Vaidya. "Deafness: A MAC Problem in Ad Hoc Networks when using Directional Antennas". In: *ICNP '04: Proceedings of the 12th IEEE International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 283–292. ISBN: 0-7695-2161-4 (cit. on p. 5).

[16] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996 (cit. on p. 5).

[17] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. "Efficient Synchronization on Multiprocessors with Shared Memory". In: *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. ACM, 1986, pp. 218–228. ISBN: 0-89791-198-9 (cit. on p. 7).

[18] Allan Gottlieb et al. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer". In: *IEEE Transactions on Computers* 32.2 (1983), pp. 175–189 (cit. on p. 7).

[19] Maurice Herlihy. "Wait-free synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1 (1991), pp. 124–149. ISSN: 0164-0925 (cit. on p. 7).

[20] Ambuj K. Singh, James H. Anderson, and Mohamed G.Gouda. "The elusive atomic register". In: *J. ACM* 41.2 (1994), pp. 311–339. ISSN: 0004-5411 (cit. on p. 7).

[21] Eli Upfal and Avi Wigderson. "How to share memory in a distributed system". In: *Journal of the ACM* 34.1 (1987), pp. 116–127 (cit. on p. 7).

[22] Tim J. Harris. "A survey of PRAM simulation techniques". In: *ACM Computing Surveys* 26.2 (1994), pp. 187–206 (cit. on p. 7).

[23] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing memory robustly in message-passing systems". In: *Journal of ACM* 42.1 (1995), pp. 124–142. ISSN: 0004-5411 (cit. on pp. 7, 8).

[24] Edsger W. Dijkstra. "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs." In: *Communications of ACM* 18 (1975), pp. 453–457 (cit. on pp. 8, 9).

[25] Nechama Allenberg-Navony, Alon Itai, and Shlomo Moran. "Average and randomized complexity of distributed problems". In: *Distributed Algorithms*. Vol. 857. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1994, pp. 311–325 (cit. on p. 10).

[26] Paul C. Attie, Nissim Francez, and Orna Grumberg. "Fairness and Hyperfairness in Multi-Party Interactions". In: *Distributed Computing* 6.4 (1993), pp. 245–254 (cit. on pp. 10, 12, 15, 91).

[27] Shlomi Dolev, Amos Israeli, and Shlomo Moran. "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity". In: *Distributed Computing* 7.1 (1993), pp. 3–16 (cit. on pp. 11, 12, 57).

[28] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. "On Relaxing Interleaving Assumptions". In: *Proceedings of the MCC Workshop on Self-Stabilization*. MCC Tech. Rep. STP-379-89. 1989 (cit. on pp. 11, 12, 53).

[29] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. "Maintaining Digital Clocks in Step". In: *Parallel Processing Letters* 1 (1991), pp. 11–18 (cit. on pp. 11, 12).

[30] Shing-Tsaan Huang, Lih-Chyau Wuu, and Ming-Shin Tsai. "Distributed Execution Model for Self-Stabilizing Systems". In: *ICDCS:Proceedings of the 14th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 1994, pp. 432–439 (cit. on pp. 11, 12).

[31] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992. ISBN: 978-0-387-97664-8 (cit. on pp. 13, 15, 24).

[32] Bowen Alpern and Fred B. Schneider. “Recognizing Safety and Liveness”. In: *Distributed Computing* 2.3 (1987), pp. 117–126 (cit. on pp. 13, 14).

[33] Zohar Manna and Amir Pnueli. “A Hierarchy of Temporal Properties”. In: *PODC:Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1990, pp. 377–410 (cit. on p. 14).

[34] A. Prasad Sistla. “On Characterization of Safety and Liveness Properties in Temporal Logic”. In: *PODC: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1985, pp. 39–48 (cit. on p. 14).

[35] Bowen Alpern and Fred B. Schneider. “Defining Liveness”. In: *Information Processing Letters* 21.4 (1985), pp. 181–185 (cit. on p. 14).

[36] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. “Defining Fairness”. In: *Concurrency Theory, 16th International Conference, CONCUR*. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 458–472. ISBN: 3-540-28309-9 (cit. on p. 14).

[37] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach*. CRC Press, 2006. ISBN: 9781584885641 (cit. on p. 14).

[38] Marta Zofia Kwiatkowska. “Fairness for Non-Interleaving Concurrency”. PhD thesis. University of Leicester, 1989 (cit. on p. 14).

[39] Nissim Francez. *Fairness*. Springer-Verlag, 1986. ISBN: 0-387-96235-2 (cit. on pp. 14, 15).

[40] Edsger W. Dijkstra. “Hierarchical ordering of sequential processes”. In: *Acta Informatica* 1.2 (1971), pp. 115–138 (cit. on p. 15).

[41] Hagen Völzer. “On Conspiracies and Hyperfairness in Distributed Computing”. In: *Distributed Computing, 19th International Conference, DISC*. Vol. 3724. Lecture Notes in Computer Science. Springer, 2005, pp. 33–47. ISBN: 3-540-29163-6 (cit. on pp. 15, 41, 91).

[42] Leslie Lamport. “Fairness and hyperfairness”. In: *Distributed Computing* 13.4 (2000), pp. 239–245 (cit. on p. 15).

[43] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. ISBN: 978-0201058666 (cit. on pp. 15, 24).

[44] Agathe Merceron. “Fair Processes”. In: *Advances in Petri Nets*. Vol. 266. Lecture Notes in Computer Science. Springer, 1987, pp. 181–195 (cit. on p. 15).

[45] Eike Best. “Fairness and Conspiracies”. In: *Information Processing Letters* 18.4 (1984), pp. 215–220 (cit. on p. 15).

[46] Marco Schneider. “Self-stabilization”. In: *ACM Comput. Surv.* 25.1 (1993), pp. 45–67. ISSN: 0360-0300. DOI: `http://doi.acm.org/10.1145/151254.151256` (cit. on p. 17).

[47] Algirdas Avižienis et al. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on pp. 19, 20).

[48] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994, p. 448 (cit. on pp. 19, 20).

[49] Anish Arora and Sandeep S. Kulkarni. “Designing Masking Fault Tolerance via Nonmasking Fault Tolerance”. In: *Proceedings of 14th Symposium on Reliable Distributed Systems*. IEEE Computer Society, 1995, pp. 174–185 (cit. on p. 20).

[50] Sandeep S. Kulkarni and Ali Ebnenasir. “Enhancing The Fault-Tolerance of Nonmasking Programs”. In: *Proceedings of 25th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2003, pp. 441–449 (cit. on p. 20).

[51] Felix C. Freiling and Sukumar Ghosh. “Code Stabilization”. In: *Proceedings of 7th International Symposium on Self-Stabilizing Systems (SSS)*. Ed. by Ted Herman and Sebasitien Tixeuil. Vol. 3764. Lecture Notes in Computer Science. Springer, 2005, pp. 128–139 (cit. on p. 20).

[52] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. “Stabilization and Pseudo-Stabilization”. In: *Distributed Computing* 7.1 (1993), pp. 35–42 (cit. on pp. 20, 21).

[53] Mohamed G. Gouda. “The Theory of Weak Stabilization”. In: *Self-Stabilizing Systems*. Vol. 2194. LNCS. Springer, 2001, pp. 114–123 (cit. on pp. 20, 40, 41, 53).

[54] Ted Herman. “Probabilistic Self-Stabilization”. In: *Information Processing Letters* 35.2 (1990), pp. 63–67 (cit. on p. 21).

[55] Dana Angluin. “Local and Global Properties in Networks of Processors (Extended Abstract)”. In: *Twelfth Annual ACM Symposium on Theory of Computing, STOC*. 1980, pp. 82–93 (cit. on p. 21).

[56] Kenneth L. Mcmillan. *Symbolic Model Checking*. Kluwer Academic, 1993, p. 194 (cit. on p. 23).

[57] Tatsuhiro Tsuchiya et al. “Symbolic Model Checking for Self-Stabilizing Algorithms”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.1 (2001), pp. 81–95 (cit. on pp. 23, 24).

[58] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Transactions on Programming Languages and Systems* 8.2 (1986), pp. 244–263 (cit. on p. 23).

[59] Nikolaos D. Liveris et al. “State space abstraction for parameterized self-stabilizing embedded systems”. In: *8th ACM & IEEE International conference on Embedded software (EMSOFT)*. Ed. by Luca de Alfaro and Jens Palsberg. ACM, 2008 (cit. on p. 23).

[60] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002, p. 384 (cit. on p. 24).

[61] Mohamed G. Gouda and Nicholas J. Multari. “Stabilizing Communication Protocols”. In: *IEEE Transactions on Computers* 40.4 (1991), pp. 448–458 (cit. on p. 24).

[62] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. “Finite-state self-stabilizing protocols in message-passing systems”. In: *ICDCS Workshop on Self-stabilizing Systems (WSS)*. Ed. by Anish Arora. IEEE Computer Society, 1999, pp. 62–69 (cit. on p. 24).

[63] Michael Siegel. “Phased Design and Verification of Stabilizing Systems”. PhD thesis. University of Kiel, 1996 (cit. on p. 24).

[64] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 1977, pp. 46–57 (cit. on p. 24).

[65] Sam Owre, John M. Rushby, and Natarajan Shankar. “PVS: A Prototype Verification System”. In: *11th International Conference on Automated Deduction (CADE)*. Ed. by Deepak Kapur. Vol. 607. Lecture Notes in Computer Science. Springer, 1992, pp. 748–752 (cit. on p. 24).

[66] Ignatius Sri Wishnu Brata Prasetya. “Mechanically Supported Design of Self-stabilizing Algorithms”. PhD thesis. Utrecht University, 1995 (cit. on p. 24).

[67] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993. ISBN: 0521441897 (cit. on p. 24).

[68] Joffroy Beauquier et al. “Proving Convergence of Self-stabilizing Systems using First-Order Rewriting and Regular languages”. In: *Distributed Computing* 14.2 (2001), pp. 83–95 (cit. on p. 25).

[69] Nachum Dershowitz and Jean-Pierre Jouannaud. “Handbook of Theoretical Computer Science”. In: ed. by Jan van Leeuwen. Vol. B: Formal Methods and Semantics. Elsevier and MIT Press, 1990. Chap. Rewrite Systems, pp. 243–320 (cit. on p. 25).

[70] Oliver Theel and Felix C. Gärtner. “An Exercise in Proving Convergence through Transfer Functions”. In: *ICDCS Workshop on Self-stabilizing Systems (WSS)*. Ed. by Anish Arora. IEEE Computer Society, 1999, pp. 41–47 (cit. on p. 25).

[71] Hassan K. Khalil. *Nonlinear Systems*. 3rd ed. Prentice Hall, 2001, p. 750. ISBN: 0130673897 (cit. on p. 25).

[72] Oliver Theel. "Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms". In: *Proceedings of the 14th Symposium on Distributed Computing (DISC)*. Ed. by Maurice Herlihy. Vol. 1914. Lecture Notes in Computer Science. Springer, 2000, pp. 209–203 (cit. on p. 25).

[73] Oliver Theel. "An Exercise in Proving Self-Stabilization through Ljapunov Functions". In: *ICDCS Workshop on Self-Stabilization (WSS)*. Ed. by Anish Arora. IEEE Computer Society, 2001, pp. 727–730 (cit. on p. 25).

[74] Jens Oehlerking, Abhishek Dhama, and Oliver Theel. "Towards Automatic Convergence Verification of Self-Stabilizing Algorithms". In: *Self-Stabilizing Systems, 7th International Symposium on Self-Stabilizing Systems*. Ed. by T. Herman and S. Tixeuil. Vol. 3764. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 198–213 (cit. on p. 25).

[75] Abhishek Dhama, Jens Oehlerking, and Oliver Theel. "Verification of Orbitally Self-Stabilizing Distributed Algorithms using Lyapunov Functions and Poincaré Maps". In: *Proceedings of 12th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, 2006, pp. 23–30 (cit. on p. 25).

[76] Hans S. Witsenhausen. " A class of hybrid-state continuous-time dynamic systems". In: *IEEE Transactions on Automatic Control* 11.2 (1966), pp. 161–167 (cit. on p. 25).

[77] Aleksandr Mikhailovich Lyapunov. "Probléme général de la stabilité du movement". In: *Annales de la Faculté des Sciences de Toulouse* 9 (1907). Translation of a paper published in Comm. Soc. math. Kharkow, 1893, reprinted in Ann. math. Studies No. 17, Princeton University Press, 1949, pp. 203–474 (cit. on p. 25).

[78] Stefan Pettersson. "Analysis and Design of Hybrid Systems". PhD thesis. Chalmers University of Technology, 1999 (cit. on p. 25).

[79] Stephen Boyd et al. *Linear Matrix Inequalities in Systems and Control Theory*. SIAM, 1994, p. 193 (cit. on p. 26).

[80] John Guckenheimer and Philip Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Springer, 1983, p. 459 (cit. on p. 26).

[81] Mohamed G. Gouda and Ted Herman. "Adaptive Programming". In: *IEEE Transactions on Software Engineering* 17.9 (1991), pp. 911–921 (cit. on pp. 27, 107).

[82] William Leal and Anish Arora. "Scalable Self-Stabilization via Composition". In: *24th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2004, pp. 12–21 (cit. on p. 27).

[83] Anish Arora and Sandeep S. Kulkarni. "Component Based Design of Multitolerant Systems". In: *IEEE Transactions on Software Engineering* 24.1 (1998), pp. 63–78 (cit. on p. 27).

[84] George Varghese. "Compositional Proofs of Self-stabilizing Protocols". In: *3rd Workshop on Self-stabilizing Systems (WSS)*. Ed. by Sukumar Ghosh and Ted Herman. Carleton University Press, 1997, pp. 80–94. ISBN: 0886293332 (cit. on p. 28).

[85] Mohamed G. Gouda and F. Furman Haddix. "The Alternator". In: *Distributed Computing* 20.1 (2007), pp. 21–28 (cit. on pp. 30, 53, 65).

[86] Joep L. W. Kessels. "An Exercise in Proving Self-Stabilization with a Variant Function". In: *Information Processing Letters* 29.1 (1988), pp. 39–42 (cit. on p. 31).

[87] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. "Randomized Self-Stabilizing and Space Optimal Leader Election under Arbitrary Schedulers on Rings". In: *Distributed Computing* 20.1 (2007), pp. 75–93 (cit. on pp. 39, 40).

[88] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. "Weak vs. Self vs. Probabilistic Stabilization". In: *International Conference on Distributed Computing Systems ICDCS*. IEEE Computer Society, 2008, pp. 681–688 (cit. on pp. 41, 53).

[89] Ananda Basu et al. "Priority Scheduling of Distributed Systems Based on Model Checking". In: *CAV*. Vol. 5643. LNCS. 2009, pp. 79–93 (cit. on p. 53).

[90] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. “Modular Ranking Abstraction”. In: *International Journal of Foundations of Computer Science* 18.1 (2007), pp. 5–44 (cit. on p. 53).

[91] Shmuel Katz and Kenneth J. Perry. “Self-Stabilizing Extensions for Message-Passing Systems”. In: *Distributed Computing* 7.1 (1993), pp. 17–26 (cit. on p. 54).

[92] Baruch Awerbuch and George Varghese. “Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols (Extended Abstract)”. In: *32nd Annual Symposium on Foundations of Computer Science*. IEEE, 1991, pp. 258–267 (cit. on p. 54).

[93] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. “Self-Stabilization By Local Checking and Correction (Extended Abstract)”. In: *32nd Annual Symposium on Foundations of Computer Science*. IEEE, 1991, pp. 268–277 (cit. on p. 54).

[94] Yehuda Afek, Shay Kutten, and Moti Yung. “The Local Detection Paradigm and Its Application to Self-Stabilization”. In: *Theoretical Computer Science* 186.1-2 (1997), pp. 199–229 (cit. on pp. 54, 57, 58, 60–62, 66, 94, 95).

[95] Yehuda Afek and Shlomi Dolev. “Local Stabilizer”. In: *Journal of Parallel and Distributed Computing* 62.5 (2002), pp. 745–765 (cit. on pp. 54, 108, 114).

[96] Joffroy Beauquier et al. “Transient Fault Detectors”. In: *12th International Symposium on Distributed Computing (DISC)*. Vol. 1499. Lecture Notes in Computer Science. Springer, 1998, pp. 62–74 (cit. on p. 54).

[97] Anish Arora and Mohamed G. Gouda. “Distributed Reset”. In: *IEEE Transactions on Computers* 43.9 (1994), pp. 1026–1038 (cit. on pp. 54, 108, 110, 114).

[98] Masaaki Mizuno and Hirotsugu Kakugawa. “A Timestamp Based Transformation of Self-Stabilizing Programs for Distributed Computing Environments”. In: *International Workshop on Distributed Algorithms WDAG*. Vol. 1151. LNCS. 1996, pp. 304–321 (cit. on p. 54).

[99] Hirotsugu Kakugawa, Masaaki Mizuno, and Mikhail Nesterenko. “Development of Self - Stabilizing Distributed Algorithms using Transformation: Case Studies”. In: *WSS*. Carleton University Press, 1997, pp. 16–30. ISBN: 0886293332 (cit. on p. 54).

[100] K. Mani Chandy and Jayadev Misra. “The Drinking Philosopher’s Problem”. In: *ACM Transactions on Programming Languages and Systems* 6.4 (1984), pp. 632–646 (cit. on p. 54).

[101] Joffroy Beauquier et al. “Self-Stabilizing Local Mutual Exclusion and Daemon Refinement”. In: *DISC*. Vol. 1914. LNCS. Springer, 2000, pp. 223–237 (cit. on p. 54).

[102] Mikhail Nesterenko and Anish Arora. “Stabilization-Preserving Atomicity Refinement”. In: *J. Parallel Distrib. Comput.* 62.5 (2002), pp. 766–791 (cit. on p. 54).

[103] Christian Boulinier, Franck Petit, and Vincent Villain. “When Graph Theory helps Self-Stabilization”. In: *Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing*. 2004, pp. 150–159 (cit. on p. 54).

[104] Mehmet Hakan Karaata. “Self-Stabilizing Strong Fairness under Weak Fairness”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.4 (2001), pp. 337–345 (cit. on p. 54).

[105] Matthew Lang and Paolo A. G. Sivilotti. “A Distributed Maximal Scheduler for Strong Fairness”. In: *Proceedings of 21st International Symposium on Distributed Computing (DISC)*. Vol. 4731. Lecture Notes in Computer Science. Springer, 2007, pp. 358–372 (cit. on p. 54).

[106] Alain Cournier et al. “Self-Stabilizing PIF Algorithm in Arbitrary Rooted Networks”. In: *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2001, pp. 91–98 (cit. on p. 56).

[107] Baruch Awerbuch et al. “Time optimal self-stabilizing synchronization”. In: *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1993, pp. 652–661 (cit. on p. 56).

[108] Ichiro Suzuki and Tadao Kasami. “A Distributed Mutual Exclusion Algorithm”. In: *ACM Transactions on Computer Systems* 3.4 (1985), pp. 344–349 (cit. on p. 57).

[109] Shlomi Dolev. *Self-Stabilization*. The MIT Press, 2000. ISBN: 0-262-04178-2 (cit. on pp. 58, 60, 63, 66, 94, 95).

[110] Monika Rauch Henzinger and Valerie King. “Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation”. In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, STOC*. ACM, 1995, pp. 519–527 (cit. on p. 63).

[111] Christian Boulinier and Franck Petit. “Self-stabilizing Wavelets and ϱ-Hops Coordination”. In: *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008, pp. 1–8 (cit. on pp. 80, 83).

[112] James Gosling et al. *The Java® Language Specification*. 3rd ed. Addison Wesley, 2005 (cit. on p. 87).

[113] Wolfgang Schreiner. *A Toolkit for the Simulation of Distributed Algorithms in Java*. Accessed on February 1, 2011. DOI: `https://www.risc.jku.at/software/daj/` (cit. on p. 87).

[114] Leon Winter. *Simulationsumgebung zur Transformation selbststabilisierender Algorithmen*. Bachelor Thesis, Carl von Ossietzky University of Oldenburg. 2011 (cit. on p. 89).

[115] Joseph Y. Halpern and Yoram Moses. “Knowledge and Common Knowledge in a Distributed Environment”. In: *Journal of ACM* 37.3 (1990), pp. 549–587 (cit. on p. 90).

[116] Jim N. Gray. *Notes on Database Operating Systems*. Tech. rep. RJ2188. San Jose, California 95193: IBM Research Laboratory, 1978 (cit. on p. 90).

[117] Gil Neiger and Sam Toueg. “Simulating Synchronized Clocks and Common Knowledge in Distributed Systems”. In: *Journal of ACM* 40.2 (1993), pp. 334–367 (cit. on p. 90).

[118] Joseph Y. Halpern and Sabina Petride. “A knowledge-based analysis of global function computation”. In: *Distributed Computing* 23.3 (2010), pp. 197–224 (cit. on p. 90).

[119] Ernst-Rüdiger Olderog and Krzysztof R. Apt. “Fairness in Parallel Programs: The Transformational Appproach”. In: *ACM Transactions on Programming Languages and Systems* 10.3 (1988), pp. 420–455 (cit. on p. 91).

[120] Matthew Lang and Paolo A.G. Sivilotti. “On the Impossibility of Maximal Scheduling for Strong Fairness with Interleaving”. In: *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2009, pp. 482–489 (cit. on p. 91).

[121] Yehuda Afek and Geoffrey M. Brown. “Self-Stabilization Over Unreliable Communication Media”. In: *Distributed Computing* 7.1 (1993), pp. 27–34 (cit. on p. 96).

[122] Shlomi Dolev, Amos Israeli, and Shlomo Moran. “Resource Bounds for Self-Stabilizing Message-Driven Protocols”. In: *SIAM Journal on Computing* 26.1 (1997), pp. 273–290 (cit. on p. 96).

[123] Shlomi Dolev and Ted Herman. “Superstabilizing Protocols for Dynamic Distributed Systems”. In: *Chicago Journal of Theoretical Computer Science* 1997 (1997) (cit. on p. 108).

List of Publications

1. Pepijn Crouzen, Ernst Moritz Hahn, Holger Hermanns, Abhishek Dhama, Oliver Theel, Ralf Wimmer, Bettina Braitling and Bernd Becker. Bounded Fairness for Probabilistic Distributed Algorithms. In B. Caillaud, J. Carmona and K. Hiraishi, editors, *Eleventh International Conference on Application of Concurrency to System Design. ACSD*, 2011, pages 89–97. IEEE .
2. Ralf Wimmer, Bettina Braitling, Bernd Becker, Ernst Moritz Hahn, Pepijn Crouzen, Holger Hermanns, Abhishek Dhama and Oliver Theel. Symblicit Calculation of Long-Run Averages for Concurrent Probabilistic Systems. In *Seventh International Conference on the Quantitative Evaluation of Systems. QEST*, 2010, pages 27–36. IEEE Computer Society.
3. Abhishek Dhama and Oliver Theel. A Transformational Approach for Designing Scheduler-Oblivious Self-stabilizing Algorithms. In S. Dolev, J. A. Cobb, M. J. Fischer and M. Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, SSS, volume LNCS 6366, 2010, pages 80–95. Springer-Verlag.
4. Abhishek Dhama, Oliver Theel, Pepijn Crouzen, Holger Hermanns, Ralf Wimmer and Bernd Becker. Dependability Engineering of Silent Self-stabilizing Systems. In R. Guerraoui and F. Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, SSS, volume LNCS 5873, 2009, pages 238–253. Springer-Verlag.
5. Nils Müllner, Abhishek Dhama, and Oliver Theel. Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation. In *Forty-first Annual Simulation Symposium. ANSS*, 2008, pages 183–192. IEEE Computer Society.
6. Steffen Becker, Wilhelm Hasselbring, Alexandra Paul, Marko Boskovic, Heiko Koziolek, Jan Ploski, Abhishek Dhama, Henrik Lipskoch, Matthias Rohr, Daniel Winteler, Simon Giesecke, Roland Meyer, Mani Swaminathan, Jens Happe, Margarete Muhle and Timo Warns. Trustworthy software systems: a discussion of basic concepts and terminology. In *ACM SIGSOFT Software Engineering Notes*, volume 31, (2006) pages 1–18.
7. Abhishek Dhama, Oliver Theel and Timo Warns. Reliability and Availability Analysis of Self-Stabilizing Systems. In A. Datta and M. Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems*, SSS, volume LNCS 4280, 2006, pages 241–261. Springer-Verlag.
8. Abhishek Dhama, Jens Oehlerking, and Oliver Theel. Verification of Orbitally Self-Stabilizing Distributed Algorithms using Lyapunov Functions and Poincaré Maps. In *12th International Conference on Parallel and Distributed Systems (ICPADS)*, 2006, pages 23–30. IEEE Computer Society.
9. Jens Oehlerking, Abhishek Dhama and Oliver Theel. Towards Automatic Convergence Verification of Self-stabilizing Algorithms. In T. Herman and S. Tixeuil, editors, *Self-Stabilizing Systems*, SSS, volume LNCS 3764, 2005, pages 198–213. Springer-Verlag.