Memory-Optimization for Self-Stabilizing Distributed Algorithms

Gabriel Le Bouder

THÈSE

PRÉSENTÉE À

SORBONNE UNIVERSITÉ

ÉCOLE DOCTORALE INFORMATIQUE,
TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE

Par **Gabriel Le Bouder**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisation de la Mémoire pour les Algorithmes Distribués Auto-Stabilisants

---

Thèse dirigée par Lélia Blin et Franck Petit préparée au Laboratoire d'Informatique de Paris 6 (LIP6), équipe Delys, **soutenue publiquement le 6 janvier 2023.**

après avis des rapporteurs :

Stéphane Devismes	–	Professeur, Université de Picardie Jules Verne, France
Christian Scheideler	–	Professeur, Université de Paderborn, Allemagne

et devant le jury composé de :

Rapporteur	Stéphane Devismes	–	Professeur, Université de Picardie Jules Verne, France
Rapporteur	Christian Scheideler	–	Professeur, Université de Paderborn, Allemagne
Examinateur	Nicolas Hanusse	–	Directeur de Recherche, CNRS, LaBRi, France
Examinateur	Alessia Milani	–	Professeure, Aix-Marseille Université, France
Examinateur	Sébastien Tixeuil	–	Professeur, Sorbonne Université, France
Directrice	Lélia Blin	–	Maîtresse de conférence HDR, LIP6, Université d'Évry, France
Directeur	Franck Petit	–	Professeur, Sorbonne Université, France

– 2023 –

Memory-Optimization for Self-Stabilizing Distributed Algorithms

Gabriel Le Bouder

in fulfillment of the requirements for the degree of
Doctor in the subject of
Computer Science

January 6th, 2023

Referees	Stéphane Devismes	Professor - University of Picardie Jules Verne, France	
	Christian Scheideler	Professor - Paderborn University, Germany	
Committee	Stéphane Devismes	Professor - University of Picardie Jules Verne, France.	Referee
	Christian Scheideler	Professor - Paderborn University, Germany.	Referee
	Nicolas Hanusse	Research Director - CNRS, LaBRI, France.	Examinor
	Alessia Milani	Professor - Aix-Marseille Université, France.	Examinor
	Sébastien Tixeuil	Professor - Sorbonne Université, France.	Examinor
	Lélia Blin	Associate Professor HDR - LIP6, Université d'Évry, France.	Director
	Franck Petit	Professor - Sorbonne Université, France.	Director

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Résumé

L'auto-stabilisation est un paradigme adapté aux systèmes distribués, particulièrement susceptibles de subir des fautes transitoires. Des erreurs de corruption de mémoire, de messages, la rupture d'un lien de communication peuvent plonger le système dans un état incohérent. Un protocole est auto-stabilisant si, quel que soit l'état initial du système, il garantit un retour à un fonctionnement normal en temps fini.

Plusieurs contraintes s'appliquent aux algorithmes conçus pour les systèmes distribués. L'asynchronie en est un exemple emblématique. Avec le développement de réseaux d'objets connectés, censés être autonomes, il devient également central de concevoir des algorithmes ayant un faible coût en termes de consommation énergétique et peu exigeants en termes de ressources.

Une des manières d'appréhender ces problèmes est de chercher à réduire la taille des messages échangés entre les différents nœuds du réseau. Cette thèse se concentre sur l'optimisation de la mémoire nécessaire à la communication pour les algorithmes distribués auto-stabilisants.

Nous établissons dans cette thèse plusieurs résultats négatifs, démontrant l'impossibilité de résoudre certains problèmes sans une certaine taille minimale pour les messages échangés, en établissant une impossibilité d'utiliser jusqu'au bout l'existence d'identifiants uniques dans le réseau en dessous de cette taille minimale. Ces résultats sont génériques et peuvent s'appliquer à de nombreux problèmes distribués. Dans un second temps, nous proposons des algorithmes particulièrement efficaces en mémoire pour la résolution de deux problèmes fondamentaux des systèmes distribués: la détection de terminaison, et la circulation perpétuelle de jeton.

Abstract

Self-stabilization is a suitable paradigm for distributed systems, particularly prone to transient faults. Errors such as memory or messages corruption, break of a communication link, can put the system in an inconsistent state. A protocol is self-stabilizing if, whatever the initial state of the system, it guarantees that it will return a normal behavior in finite time.

Several constraints concern algorithms designed for distributed systems. Asynchrony is one emblematic example. With the development of networks of connected, autonomous devices, it also becomes crucial to design algorithms with a low energy consumption, and not requiring much in terms of resources.

One way to address these problems is to aim at reducing the size of the messages exchanged between the nodes of the network. This thesis focuses on the memory optimization of the communication for self-stabilizing distributed algorithms.

We establish in this thesis several negative results, which prove the impossibility to solve some problems under a certain limit on the size of the exchanged messages, by showing an impossibility to fully use the presence of unique identifiers in the network below that minimal size. Those results are generic, and may apply to numerous distributed problems. Secondly, we propose particularly efficient algorithms in terms of memory for two fundamental problems in distributed systems: the termination detection, and the token circulation.

Acknowledgments

Bien que signée de ma main seule, cette thèse n'a été menée à son terme que grâce à l'implication de bien des personnes autour de moi.

Je remercie en premier lieu mes encadrants Lélia et Franck, qui m'ont transmis leur intérêt pour la recherche, m'ont accompagné sur les plans scientifique et moral dans cette longue et difficile tâche, toujours avec la préoccupation de maintenir une ambiance de travail saine et respectueuse.

Je tiens également à remercier toutes les autres personnes avec qui j'ai collaboré durant ces trois ans, et notamment Swan, Laurent et Colette.

Merci aussi à ceux qui, dans mon équipe, ont fait de notre couloir un lieu de vie riche d'échanges, de débats et de vie, Aymeric, Baptiste, Célia, Étienne, Francis, Ilyas, Jonhatan, Jonhatan, Julien, Reda, Saalik, et Vincent.

Une thèse requiert un investissement qui déborde des portes du bureau. Je remercie Carmen qui m'a aidé, supporté, accompagné, dans les bons et les mauvais moments, et sans qui ces trois ans auraient été bien plus difficiles.

Je remercie aussi les copaines de Bordeaux, de Cachan, d'Ivry, de Vitry et d'ailleurs, qui m'ont permis de respirer quand j'en avais besoin.

Je tiens également à remercier ma famille, et en particulier mon père, qui m'a soutenu dans les moments les plus compliqués.

Enfin, je ne peux ne pas citer ma mère, qui a participé à faire de moi ce que je suis aujourd'hui.

Contents

List of Figures

Chapter 1

Introduction

> The Matrix is everywhere. It is all around us. Even now, in this very room. You can see it when you look out your window or when you turn on your television. You can feel it when you go to work... when you go to church... when you pay your taxes.
>
> Morpheus

The continuous development of communication technologies has deeply changed every aspect of our society. Most users see this evolution through the increasing role and size of Internet, social networks, instant messaging, *etc.* It also brought ruptures in the core of the economy. Six out of the eight biggest companies in the world, in terms of capitalization, are directly linked to computing technology, produce both devices and services, the stock market itself also relies on those technologies.

Communication technology is in constant change, and already widely diverse. Data collection from connected objects relies on the structure of the Internet of Things (IoT), GPS technology relies on a very precise synchronization in satellite networks, phone internet is based on Wi-Fi networks.

With the development of communication protocols involving small, autonomous devices which communicate via WiFi, it is crucial to design algorithms which are efficient in terms of energy, memory, CPU...

All these communication technologies can be described as **distributed systems**. A distributed system, which we also call **network**, is a system composed of several autonomous computing units, and of communication links between those units. Each device computes autonomously, depending on the information it has, and can send and receive information to other devices through the communication links. Each unit is directly linked to a non-empty subset of the other units, its neighbors, such that there exists a communication path between any two computing units. This definition includes computer networks, sensor networks, swarm robots, parallel computers....

Although very diverse, distributed systems share some characteristics which differentiate them from central systems. In distributed systems, the different computing units of a distributed system do not necessarily compute at the same speed, may run different computations which do not take the same time, may run on different materials which do not have the same performances. This results in an **asynchrony** inherent to distributed systems. In particular, the individual clocks of each device may desynchronize, which invalidates the possibility to rely on a global notion of time.

We must also consider that the communication between devices takes place through communication links of various qualities, length, with different speeds. This makes it impossible to define a reliable order on the operations made by separate devices: an execution on one system may not be linearizable. Therefore, one cannot expect any synchronization between the different devices of a distributed system. This asynchrony has, as a direct consequence, **non-determinism** on the evolution of the system. Even if the algorithm con-

sidered is deterministic, the order in which the nodes compute, send and receive messages may be different in two separate executions.

Distributed systems are more prone to **faults** than centralized systems. Such systems cover a huge quantity of diverse devices and depend on a large variety of communication technologies, which make the existence of communication or unit errors very likely within time. Therefore, algorithms for distributed systems should be fault-resilient.

In distributed systems, the computing units are autonomous, which means that they all execute their own code, and update their own variables. By definition, nodes have **no global view on the state of the system**. Therefore, the computations and decisions made by the individual devices in order to make the system behave correctly only depend on the local memory of the device, and on the messages it received from its direct neighbors. Due to asynchrony, this information may be outdated, and due to faults, they can even be erroneous.

Wireless networks are simpler to deploy than wired networks, and therefore are more and more spread, and larger and larger within time. The IoT, sensor networks, or swarm robots are research and technological areas in constant development. All those networks share some properties which make it a specific challenge to design algorithms for them. In wired networks, the messages are exchanged through physical, wired, communication links. Due to commutation technology, everything behaves like if any communication link was shared by exactly two devices. In such situations, identifying the communication link, the port on which it is physically connected, allows devices to know which of their neighbors sent them some message. Reciprocally, devices can without difficulty send a message to exactly one of their neighbors.

Things are very different for wireless networks. In such networks, any message sent by one device is identically received by all of its neighbors, through wireless communication (WiFi, Bluetooth, radio...). Therefore, sending a message to one specific neighbor is impossible, and **addressing a message to one specific neighbor is a challenge**, since the same message will be received by all the other nodes in the neighborhood. Not only does this complicate the resolution of some tasks, but in addition this can lead to an increase in the number of sent messages in the network, and even to loops of propagation of information.

One way to solve this issue is to rely on **unique identifiers** possessed by nodes, such as MAC addresses, or IP addresses. If all nodes share their unique identifier to all of their neighbors, then it becomes possible to address a message to one specific neighbor by prefixing the message with the recipient identifier. This technique has several flaws. The first one is that it does not apply to networks in which nodes do not have a unique, permanent, identifier. Such networks are said anonymous. A lot of operations are much more complicated to achieve in anonymous networks. Even in networks in which devices have a globally unique identifier, it might be desirable to avoid such solutions. Indeed, this solution requires the transmission if identifiers, by WiFi, to the entire neighborhood, which rises **privacy and/or security issues**. Finally, prefixing all messages by the identifier of the recipient has a non-negligible cost in terms of **size of the message** communicated by the devices.

Sensor networks, robot networks, also have in common the fact that the devices involved may be distant from any source of energy for a pretty long period of time. In order to keep them able to perform their task, algorithms for such networks should be as little talkative as possible. Indeed, in distributed systems, devices constantly exchange information to check local consistency, and perform the task they are designed for. To **save battery**, and extend the life expectancy of the devices, it becomes crucial to make the messages as short as possible. For this reason, designing algorithms which do not require the exchange of the node identifiers is an interesting challenge. One other interest of such algorithms is that

they do not require a lot of memory, since nodes do not have to store the identifiers of their neighbors, and thus are suitable for networks in which the devices have a low storage capacity.

Problems addressed in the field of distributed computing can be very diverse.

Structure construction problems designate all the problems which aim at providing the network a structure which satisfies an intended property. This structure can be spanning the whole network (maximal independent set, tree, ...), involving only one node (leader election). Building structures is often a first step to solve a more complicated problem.

Mutual exclusion problems designate all the problems which aim at assuring that one or several resources in the network (a printer, a computing server, a database...) is not accessed by several users at the same time. This exclusion can be local: no two neighbors can access the resource at the same time, or global: no two devices of the networks can access the resource at the same time.

Global observation problems designate all the problems which aim at gathering on one or several nodes information relative to the global state of the system. The particular information can be the termination of a computation as well as detecting some property on the topology of the network, or the presence of a deadlock.

Nowadays distributed systems involve more and more components, and more and more non-reliable components (watches, fridges, printers...). With this increase of the number of devices, the probability that **faults occur** increase as well. These faults can be caused by the environment, by attacks, or even by the device itself which may be poorly coded, or unadapted to the conditions in which it operates. Furthermore, the effects of such faults are also pretty diverse. It can lead to a temporary or permanent disappearing of one node, or of some communication links. I can also be simply the transmission of some erroneous messages. In the worst case, an attacker takes permanent control of one or several nodes, which hinder a normal behavior of the system.

Two main paradigms are suitable for faulty distributed systems.

Robustness [GLM06, CDPR20] is a paradigm which focuses on the nature of the solution. A solution is robust if, even when some node, or some communication links crash, the solution remains valid. Designing robust solutions to problems allows ignoring the future faults in the system. Such solutions are generally expensive, and hard to build when they exist, which is not always the case. They are mainly used for critical systems, such as nuclear plants, passenger transport (planes, autonomous cars), *etc.*

Self-stabilization, introduced by Dijkstra [Dij74], is a more general paradigm, which focuses on the ability to autonomously recover a correct configuration after some faults occur. This paradigm is suitable for transient faults, which happen rarely enough for the system to have enough time to converge between faults. An algorithm is self-stabilizing if, whatever the initial configuration of the system, it returns to a correct behavior in finite time. Several variants of self-stabilization are presented in the literature.

In this thesis, we focus on the space complexity of some distributed problems in the framework of self-stabilization for wireless networks. Small space complexity is desirable for distributed algorithms in general, and especially for wireless distributed systems. We address the question of lower bounds for the space complexity of some problems, an underdeveloped field of research. We consider both anonymous and identified networks, and introduce memory efficient self-stabilizing algorithms in both environments.

In Chapter 2 we detail the computational model used in this thesis. We formally define distributed systems, algorithms, distributed problems, and stabilization hypotheses.

In Chapter 3, we present results from [BFB21]. We study the cost of using the presence identifiers in identified networks. Most algorithms requiring unique identifiers use $\Omega(\log n)$

bits per node, to store and send the identifiers to their neighbors. Recently, algorithms solving problems which require unique identifiers were presented, and only use $O(\log \log n)$ bits per node. We prove that this complexity is asymptotically optimal, in the sense that under $O(\log \log n)$ bits per node, algorithms cannot use the presence of unique identifiers in the network, and behave as if the network was anonymous. It is a very general result: it applies to all distributed algorithms, stabilizing or not, in various specific models. We apply this result to obtain lower bounds on the space complexity of actual problems, by establishing a $\Omega(\log \log n)$ lower bound for the Leader Election problem, one of the most studied problems in the field of self-stabilization. Our lower bound is established on one very simple, and widely studied, class of graphs: non-prime rings.

These results have been awarded *Best Student Paper* at [BFB21], and extended abstracts were also presented in [BFB19, BFB22].

In Chapter 4, we present results from [BJBP22b]. We address one fundamental problem of distributed systems: the Termination Detection problem. We design a snap-stabilizing solution for this problem, with very low requirements on the network. It works in anonymous settings, is adapted to wireless networks, and works on any topology. Furthermore, our solution only requires $\Theta(\log D)$ bits per node, which is the lowest complexity achieved so far, and had never been achieved by any snap-stabilizing algorithm before. To design our algorithm, we rely on an unspecified unison algorithm. To be so generic, we established a theoretical analysis of properties of unison algorithms, from the specification of Unison problem itself.

A French-language version of these results was also presented in [BJBP22a].

In Chapter 5, we address the problem of fair token circulation. We design a self-stabilizing algorithm which solves this problem on a generic class of graphs, Destination Oriented Acyclic Graphs, and is also adapted to wireless networks. Finally, this solution requires $\Theta(\log \log n)$ bits per node, which is asymptotically optimal, and works under the weaker assumption on the synchrony of the network.

This result will soon be submitted to an international conference.

In the last chapter we summarize our contributions, and broaden their scope by opening directions in which our techniques and results might be extended.

Chapter 2

Model

Each of these lives is the right one!
Every path is the right path.
Everything could have been anything else and it would have just as much meaning.

Nemo Nobody

Contents

In this chapter we introduce the formal framework in which this thesis lies. We introduce the computational model, the communication model, and other objects that are considered all along this thesis.

2.1 Preliminaries

In this section we define some mathematical objects and properties that will be useful later.

Relation Let V be a set. A *binary relation* R on V is a subset of $V \times V$. We write uRv as an equivalent of $(u, v) \in R$. A binary relation is *reflexive* if $\forall v \in V, vRv$. A binary relation is *transitive* if $\forall u, v, w \in V, uRv \wedge vRw \Rightarrow uRw$. The *transitive closure* of a relation R, denoted R^+ is the smallest transitive relation which contains R. We have

$$uR^+v \iff \exists k > 0, \exists v_0, \ldots, v_k : (u = v_0 \wedge v = v_k \wedge \forall i \in [0, k-1], v_i R v_{i+1}).$$

The *reflexive transitive closure* of a relation R, denoted R^* is the smallest reflexive and transitive relation which contains R. We have

$$\begin{aligned} uR^*v &\iff \exists k \geq 0, \exists v_0, \ldots, v_k : (u = v_0 \wedge v = v_k \wedge \forall i \in [0, k-1], v_i R v_{i+1}) \\ &\iff (uR^+v) \vee (u = v). \end{aligned}$$

A binary relation R is *acyclic* if $\forall u \in V, \neg(uR^+u)$. A binary relation R is *rooted* if $\exists r \in V : \forall u \in V, uR^*r$. Remark that if a binary relation R is both rooted and acyclic, then there exists one unique $r \in V : \forall u \in V, uR^*r$. This element r is said the *root* of R.

Order A binary relation R is *antisymmetric* if $\forall u, v \in V, uRv \wedge vRu \Rightarrow u = v$. A binary relation R is an *order* if it is reflexive, transitive, and antisymmetric. In the following, orders are denoted with the symbols $\preceq$, or $\leq$. Consider an order $\preceq$, we define the *strict order* associated to $\preceq$, and denote $\prec$, the relation such that $\forall u, v \in V, u \prec v \iff u \preceq v \wedge u \neq v$.

An order $\preceq$ is *well-founded* if there does not exist any infinite sequence $v_0 v_1 \ldots$ such that $\forall i \in \mathbb{N}, v_{i+1} \prec v_i$. If $\preceq$ is a well-founded order, then $(V, \preceq)$ is a *well-founded set*.

2.2 Distributed System

A distributed system is a set of autonomous computing units, which can communicate with each other in order to complete a global task. Computing units can be computers, network devices, a core of a multicore process, *etc.* We suppose a fully decentralized system, where the different computing units do not share any resource.

2.2.1 Characteristics of Distributed Systems

No Global Time Usually, the speed of computation as well as the latency of the different communication links are not homogeneous in distributed systems. Since the processes cannot rely on a global clock, then we cannot suppose any synchronization between distant processes.

No Global Knowledge There also does not exist any shared memory through which nodes could safely and centrally communicate. The computation is local to each node, and is made without any global knowledge of the state of the system. Nodes can only exchange pieces of information through the existing communication links, point to point. Due to asynchrony, it might be non-trivial to maintain consistency in this communication model. Note that communication links are symmetric: if one process can communicate with one other process, then the opposite is true as well.

2.2.2 Model of Distributed Systems: Networks

Networks A distributed system, or *network*, is a non-oriented connected graph $G = (V, E)$. The processes of the distributed system are represented by the *nodes* of G, which

are the elements of V. The communication links of the distributed system are represented by the (non-oriented) *edges* of G, which are the elements of $E \subseteq \mathcal{P}_2(V)$ (pairs of elements of V).

There exist works focusing on *oriented*, or *directed*, networks. In oriented networks, the communication links are not necessarily symmetric: it might happen that one node can send information to one of its neighbor, without reciprocity. This is modeled by oriented edges, elements of $E \subseteq V \times V$. In this thesis we always consider bi-directional communication links, but sometimes refer to results established in oriented networks.

We call *size* of G and denote by n the number of nodes in the network. Two nodes u and v are neighbors in G if and only if $\{u, v\}$ is an edge of G. The set of all the neighbors of v is denoted N_v. We also denote by $N[v]$ the set of extended neighbors of v, $N[v] = N_v \cup \{v\}$.

Graphs Properties We call *degree* of a node v and denote Δ_v the number of neighbors of v. We call *degree* of G and denote $\Delta(G)$, or simply Δ if no confusion can be made, the maximum degree of the nodes of G.

A path of G is a sequence $p = (v_1, v_2, \ldots, v_k)$ of nodes of G such that $\forall i \in [1, k-1], \{v_i, v_{i+1}\} \in E$. We call *source* of p the node v_1, *end* of p the node v_k, and *length* of p the number of edges in p, here $k-1$. A path $p = (v_1, v_2, \ldots, v_k)$ of G is *elementary* if $\forall i \neq j \in [1, k], v_i \neq v_j$. If the source and the end of a path p are equal, then p is also said a *cycle*. A cycle $c = (v_1, v_2, \ldots, v_k)$ is an *elementary cycle* if the path $p = (v_1, v_2, \ldots, v_{k-1})$ is elementary.

The *distance* between two nodes u and v in G is the length of the shortest path of G with source u and end v. We denote by $\texttt{dist}(u, v)$ the distance between u and v. We call *diameter* of G, and denote $D(G)$, or simply D if no confusion can be made, the maximal distance between two nodes of G.

2.2.3 Local Knowledge

Port Numbering In distributed systems, nodes receive information from several neighbors, each neighbor corresponds to one specific communication link. Although one connected device does not necessarily know which entity is located at the endpoint of each communication link, it can nevertheless distinguish the different communication links from another. Practically, this distinction is made by identifying the port number associated to the communication, the frequency used by the device, *etc.* We do not suppose any global consistency in the attribution of the port numbers in the network.

In our model, a node v has access to locally unique port numbers associated with its adjacent edges. We denote by $\texttt{port}_v(u)$ the port number associated by v to the edge leading to its neighbor u. Typically, $\texttt{port}_v(u)$ is an integer in $[1, \Delta_v]$.

Identifiers, Semi-Uniform In some distributed networks, devices come with a globally unique identifier, which can be a MAC address, an IP address, *etc.* This unicity can be useful to solve problems that require symmetry breaking, for example. Due to NATing, or to false MAC-addresses, it might happen that some, but few, devices share the same identifier. In most cases, this homonymy does not cause any problem, but in some situations, this can lead to, sometimes unsolvable, issues.

Although most distributed systems are based on devices with unique identifiers, several reasons justify the interest to design distributed algorithms that do not require such identifiers. For privacy reasons, it might be crucial to design algorithms that are not based on sharing unique identifiers in the entire network. Furthermore, for this precise reason, there

exist actual networks where identifiers do exist, but do not have any consistency through time due to the use of VPN for example.

Independently of the presence of identifiers, there sometime exists one particular device in the network that has a specific role. This specific device can be the DHCP server, for a local network, or a master DNS, in one DNS zone. Some problems are easier to address if the network has a distinguished device, that can perform tasks that the other devices cannot.

A network G is *identified* if $\forall v \in V$, v has a local constant, its identifier, denoted by $\texttt{ID}_v$, such that $\forall u \neq v \in V, \texttt{ID}_u \neq \texttt{ID}_v$. For all $k \in \mathbb{N}^*$, G is *k-homonymous* if $\forall v \in V$, v has a local constant, its identifier, denoted by $\texttt{ID}_v$, and such that at least k nodes have the same identifier. Remark that according to this definition, any k-homonymous network is also k'-homonymous, for any $k' \leq k$. A network G is *anonymous* if all nodes have the same identifier or, equivalently, if there is no identifier at all. Remark that anonymous networks may be seen as n-homonymous networks.

Given a graph G, an integer $k \in \mathbb{N}^*$, and a set of identifiers $S_{\texttt{ID}}$, we use the following notations:

- $G^0[S_{\texttt{ID}}]$ represents any identified network with topology G and with unique identifiers taken in $S_{\texttt{ID}}$.
- $G^k[S_{\texttt{ID}}]$ represents any k-homonymous network with topology G and with identifiers taken in $S_{\texttt{ID}}$
- $G^n[\texttt{ID}]$ represents the anonymous network with topology G in which all nodes have identifier $\texttt{ID}$.

A network is *semi-uniform* if $\forall v \in V$, v has a local constant $\texttt{distinguish}_v$ such that $\exists w \in V : \texttt{distinguish}_w = 1$ and $\forall u \neq w, \texttt{distinguish}_u = 0$. Otherwise, the network is *uniform*.

2.2.4 Knowledge about the Topology of the System

Depending on the distributed system, some global characteristics (and especially its size, diameter, and degree) can be persistent through time. Although this does not apply to mobile networks, for example, this can be a relevant hypothesis for sensor networks, which contain devices that are not meant to be moved. In such situations, the knowledge of the values of those parameters can be helpful to efficiently solve tasks in the system. Depending on the application, the exact value of the parameter is not always necessary: an approximation or an upper bound can be sufficient. For example, an upper bound on the diameter of the network is enough to guarantee the propagation of information through the entire network.

In some cases, the nodes of the network have a local constant that stores n, Δ, D, or a combination of those parameters. In such case, the parameter is said *given*. If only an upper bound or an approximation of the value is given to the nodes, then the parameter is said *approximated*.

2.2.5 Common Topologies

In the general case, distributed systems have very diverse topologies. Thus, we aim to design distributed algorithms which behave correctly on any topology. If no precision is made, saying that a distributed algorithm completes a certain task means that it completes this task on all topologies.

Yet, it might be interesting to reason on specific, simpler topologies for at least two reasons. The first one is that some distributed systems actually respect some specific topologies, and it might be possible to find more efficient algorithms by supposing some specific topology. The other reason is that some problems can be difficult to address in the first place, and working on a simpler case can be an option to find a way to a generic solution.

We call topology a class of all the networks that respect a particular property. Here we give some common topologies that will be used in our thesis. We present examples of graph for each topology in Figure 2.1.

- G is a ring if there exists an elementary cycle of length n in G.

  Rings are a typical example of a tool-topology. Although it is one very simple topology, numerous problems of distributed computing (the presence of cycles, symmetric configurations) can be tackled in this simple topology at first.

- G is a tree if there is no elementary cycle in G.

  Trees are both a simple topology, suited to first address some problems, and a real-world topology. Indeed, minimizing the number of communication links in distributed systems is pretty useful to reduce the number of messages exchanged, to avoid conflicts...

- G is a star if there exists one node r such that all the other nodes have only r as a neighbor.

  Stars are a particular case of trees, which corresponds to highly centralized networks.

- G is a complete graph if there exists an edge between any pair of nodes.

- We also consider Directed Acyclic Graphs (DAG) and Destination Oriented Directed Acyclic Graph (DODAG), which are both particular cases of oriented graphs. Recall that we consider non-oriented networks, in the sense that the communication is always bi-directional. The orientation we are talking about is therefore more a hierarchy relation between neighbors than an orientation of the communication links. In practice, devices have a local table that allows them to know whether one specific neighbor is a parent or a child of theirs in the network. Similarly, we rely on port numbers to grant an orientation to the network.

  Suppose that the space of port numbers can be split into two disjoint sets $\texttt{port}^+$ and $\texttt{port}^-$. Let us denote by $u \to v$ the relation $\texttt{port}_u(v) \in \texttt{port}^+$.

  G is a Directed Acyclic Graph (DAG) if the relation $\to$ is acyclic. Note that this implies: $\texttt{port}_u(v) \in \texttt{port}^+ \iff \texttt{port}_v(u) \in \texttt{port}^-$.

  DAGs corresponds to hierarchical distributed systems, where devices have initiators, and followers.

  G is a Destination Oriented Directed Acyclic Graph (DODAG, or $\texttt{D}^o$) if the relation $\to$ is acyclic and rooted. We call *root* of the $\texttt{D}^o$ the root of the relation $\to$.

  DODAGs are DAGs with only exactly one node that has no ancestor, which is a common ancestor of all the other nodes. It corresponds to centralized, hierarchical networks, such as DNSs networks.

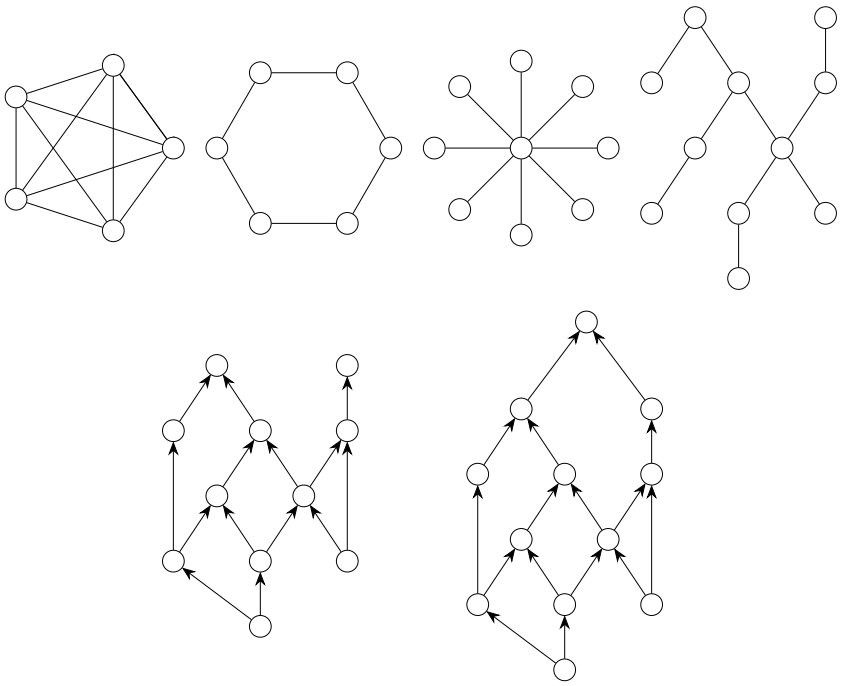

Figure 2.1: Common topologies, from left to right starting from the first row: complete graph, ring, star, tree, DAG, DODAD

2.3 Communication Model, Algorithm

2.3.1 Memory

During the execution of a distributed algorithm, the devices of the system exchange information through the communication links. Every time one device receives some information, it can update its variables, depending on its personal information, and on the one it received. After that, the device can send an updated information to its neighbors, that depends on its new state and on other local information. Thus, a device possesses two distinct types of information. The information that is inherent to its material, persistent, and the information it received from the other devices of the system, that is meant to be updated, and transmitted.

In our model, nodes have two types of memory: the *immutable* memory and the *mutable* memory. The immutable memory is not corruptible, and cannot be updated by the node itself. It typically contains the code of the algorithm, the identifier of the node, its port numbers, and constant `distinguish`. The mutable memory, also called *register*, is corruptible, and can be updated by the node itself. It typically contains the variables of the algorithm that is executed in the network.

The set of all variables in the registers of node v is called the *state* of v, and is denoted $\texttt{state}_v$. We also denote by $\texttt{local}_v$ the set of all local constants of node v, stored in its immutable memory: its identifier, port numbers, `distinguish`...

We call a *configuration* of G the set of all the states of the nodes of V. Configurations are denoted by the letter γ. We also denote by Γ the set of all possible configurations of G. The value of variable `var` on node v in configuration γ is denoted $\texttt{var}_v^\gamma$.

We call execution, and denote by the letter ϵ, any non-empty sequence of configurations: $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$.

2.3.2 Communication

State Model The communication between the devices of a distributed system highly depends on the material, of the network, and can be studied at each layer of the OSI model (ADSL/Bluetooth, Wi-Fi/Ethernet, IPv4/IPv6, TCP/UDP, *etc*). For the sake of genericity, we model communication between nodes by abstracting the sending of messages.

We consider the *state model*, denoted $\mathcal{S}$, introduced by Dijkstra [Dij74]. In the state model, each node v has read/write access to its register, and has a read-only access to the registers of its neighbors. Thus, to transmit information to its neighbors, a node simply writes that information in its mutable memory.

Knowledge of the Port Numbers In wired distributed systems, the communication between neighbors is made through physical links, each link connecting exactly two devices. In such systems, one node can naturally send different information to its neighbors. On the other hand, in wireless networks, such as sensor networks or Wi-Fi antennas, communication is multidirectional: any information emitted by one device is identically received by all of its neighbors. In such networks, it can be pretty difficult to send information to one, and only one, neighbor.

In our model, the different communication links are identified by port numbers. To embrace that diversity in the capacity to communicate, two variants of the state model are studied in the literature: the *port-known state model* and the *port-unknown state model*.

In the port-known state model, denoted $\mathcal{S}_{\mathsf{PK}}$, nodes know the port number it is assigned by its different neighbors. Namely, v has a table that contains the value $\mathtt{port}_u(v)$ for each of its neighbors u. Thus, if one neighbor u of v wants to send information to one of its neighbors, it can prefix this information with the port number it assigned to that neighbor. Since all of its neighbors know their assigned port number, they can decide whether the information is intended to them or not. The model $\mathcal{S}_{\mathsf{PK}}$ corresponds to wired distributed systems.

In the port-unknown state model, denoted $\mathcal{S}_{\mathsf{PU}}$, nodes do not have any knowledge *a priori* on which port number they were assigned by their different neighbors. This makes it non-trivial to send information to one specific neighbor. If necessary, then this difficulty must be overcome by the algorithm. The model $\mathcal{S}_{\mathsf{PU}}$ corresponds to wireless distributed systems.

The positive results and algorithms presented in Chapters 4 and 5 are all valid in the most challenging model: the port-unknown state model $\mathcal{S}_{\mathsf{PU}}$. The negative results (lower bound and impossibility results), presented in Chapter 3 are all valid in the least challenging model: the port-known state model $\mathcal{S}_{\mathsf{PK}}$.

In Chapter 5, the problem we address requires that nodes communicate information to exactly one of their neighbors. Let us give an idea on how this difficulty might be overcome. If the network is identified, then sending information to one neighbor can be done by prefixing the information by the identifier of the designated neighbor. This requires that each node permanently writes its identifier in its mutable memory, so that its neighbors have access to it (recall that the local constants of v, such as $\mathtt{ID}_v$, are not readable by the neighbors of v). This technique has one flaw: it has a non-negligible cost in terms of memory used, since storing the identifier in the mutable memory requires at least $O(\log n)$ bits on each node. In Chapter 5 we exponentially reduce the size of the memory.

Note that in anonymous networks, sending information to one specific neighbor is generally unfeasible in $\mathcal{S}_{\mathsf{PU}}$.

2.3.3 Deterministic Algorithm

Rules A distributed algorithm consists of a set of rules of the following form:

$$< label > : < guard > \rightarrow < action >$$

where

- $< label >$ is the name of the rule,
- $< guard >$ is a predicate that involves the variables of the node and of its neighbors, and the local constants of the node (its identifier, port numbers...),
- $< action >$ is a set of **deterministic** instructions that modify the state of the node.

If at least one action involves randomization, then the algorithm is said *non-deterministic*, or *randomized*. In this work, we only consider deterministic algorithms.

We denote by $R_{\mathcal{A}}$ the set of the rules of $\mathcal{A}$. If the guard of a rule is evaluated to true on node v, then this rule is said *enabled* on v. An action can be executed by node v only if the rule in which it appears is enabled. A node is *enabled* if one of its rules is enabled, and is *disabled* otherwise. We denote by $\mathcal{A}^e(\gamma)$ the set of the enabled nodes in γ. A configuration γ is *terminal* if no node is enabled in γ, *i.e.* if $\mathcal{A}^e(\gamma) = \emptyset$.

Network-Specificities of Algorithms If at least one rule of an algorithm $\mathcal{A}$ refers to the local constant `distinguish`, then it is designed to be efficient on semi-uniform networks. Algorithm $\mathcal{A}$ is said *semi-uniform*. Otherwise, it is said *uniform*. If no rule of an algorithm $\mathcal{A}$ refers to the local identifier of the node, then it is designed to be efficient on anonymous networks. Algorithm $\mathcal{A}$ is said *anonymous*. Otherwise, it is said *ID-based*.

Computation When activated, an enabled node v atomically executes the three following actions.

- It reads the state of all of its neighbors $u \in N_v$
- It executes the action of one of the rules it is enabled, according to the state of its neighbors, its own state, and its local constants.
- It writes in its mutable memory its new state.

If several nodes are activated at the same moment, they all atomically execute those three actions.

Permanent Communication As defined above, it might seem that nodes read the state of their neighbors only when activated. This is actually misleading. In a distributed system, the devices constantly exchange information, in order to detect any update, any breaking in the system. In our model, this is true as well, but hidden by the notion of enabled nodes.

A node is activated only if it enabled, which depends on the state of its neighbors. Thus, to be able to know whether it is enabled or not, a node necessarily has to constantly read the state of its neighbors, and evaluate all the guards of its rules with the updated information it read.

2.4 Scheduler

2.4.1 Asynchrony

Although devices constantly exchange information, there is no guarantee that all the nodes compute and communicate at the same speed. Most certainly, some devices are more reactive than others, some communication links are more congested than others, *etc.* Thus, it would be a huge simplification supposing that enabled nodes are immediately activated: distributed systems are inherently asynchronous.

This asynchrony is modeled by the existence of an *adversary*, called *daemon* or *scheduler*. At each step, the scheduler selects a subset of the enabled nodes, and activates all of them. Each activated node updates its state according to the action of one of its enabled rules. If several rules are enabled on the same activated node, then it non-deterministically pick one of them.

Formally, a schedule is a map $\mathcal{D}$ that takes as input a non-empty sequence of configurations $(\gamma_0, \gamma_1, \ldots, \gamma_k)$ and a subset of the nodes of the graph $V^e \subseteq V$, which represents the set of enabled nodes. $D(\gamma_0, \gamma_1, \ldots, \gamma_k, V^e) = \emptyset$ if and only if $V^e = \emptyset$. If $V^e \neq \emptyset$, then $D(\gamma_0, \gamma_1, \ldots, \gamma_k, V^e)$ is a non-empty subset of V^e.

Let us denote by $\gamma \underset{\mathcal{A}}{\rightarrow} \gamma'$ a *computing step* of algorithm $\mathcal{A}$, where γ' is obtained from γ after the activation of one or several enabled nodes and the simultaneous execution of their action.

Given a schedule $\mathcal{D}$, an *execution* of $\mathcal{A}$ under $\mathcal{D}$, is a finite or infinite execution $\epsilon = \gamma_0 \underset{\mathcal{A}}{\rightarrow} \gamma_1 \underset{\mathcal{A}}{\rightarrow} \cdots$ such that $\forall i$, the set of activated nodes between γ_i and γ_{i+1} is $\mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_i, \mathcal{A}^e(\gamma_i))$. An execution is maximal if it is infinite, or if it is finite and the last configuration of ϵ is terminal. If $\epsilon = \gamma_0 \underset{\mathcal{A}}{\rightarrow} \gamma_1 \underset{\mathcal{A}}{\rightarrow} \cdots$ is an execution, then $\forall i \geq 0$, $\epsilon' = \gamma_i \underset{\mathcal{A}}{\rightarrow} \gamma_{i+1} \underset{\mathcal{A}}{\rightarrow} \cdots$ is a *sub execution* of ϵ. A sub execution can be both finite or infinite. The first configuration of an execution, γ_0, is called the *initial configuration*.

2.4.2 Schedules and Schedulers

In practice, we don't know the precise schedule that will determine the execution of our algorithm. Furthermore, it would be very restrictive to design an algorithm for one particular schedule. What we actually do is suppose that the execution respects some properties of fairness, for example. We call *scheduler* a class of schedules which all share common properties. Since we do not use in practice the notion of schedule, we also denote by $\mathcal{D}$ the schedulers. Saying that an algorithm $\mathcal{A}$ respects some property on a certain scheduler $\mathcal{D}$ means that it respects that property on every schedule of $\mathcal{D}$.

2.4.3 Common Schedulers

Dubois and al. in [DT11] presented a vast overview of schedulers. In this section, we only present the schedulers that are invoked in this thesis. Let us consider a schedule $\mathcal{D}$ and a infinite sequence of configurations $\gamma_0, \gamma_1, \ldots$.

- The schedule $\mathcal{D}$ is *strongly fair* if the following is true for any sequence of configurations $\gamma_0, \gamma_1, \cdots$, for any $i \geq 0$ and for any node $v \in V$: if v is enabled in an infinite number of configurations γ_j with $j \geq i$ then v is eventually activated by $\mathcal{D}$. More formally, if there exists an infinite number of sets $V_j \ni v$ with $j \geq i$, then there exists $k \geq i : v \in \mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_k, V_k)$.

- The schedule $\mathcal{D}$ is *weakly fair* if the following is true for any sequence of configurations $\gamma_0, \gamma_1, \cdots$, for any $i \geq 0$ and for any node $v \in V$: if v is enabled in all the configurations $\gamma_i, \gamma_{i+1}, \ldots$, then v is eventually activated by $\mathcal{D}$. More formally, if $\forall j \geq i, v \in V_j$, then there exists $k \geq i : v \in \mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_k, V_k)$.
- The schedule $\mathcal{D}$ is *central* if it activates only one of the enabled nodes at each computing step. In other words, if for any $i \geq 0$, for any $V^e \subseteq V$, $|\mathcal{D}(\gamma_0, \gamma_1, \ldots, \gamma_i, V^e)| \leq 1$.
- The schedule $\mathcal{D}$ is *synchronous* if it activates all the enabled nodes at each computing step. Remark the there exists exactly one synchronous schedule, and that it is strongly fair.

Remark that if $\mathcal{D}$ is strongly fair, then it is also weakly fair. By commodity, we say that all schedules are *unfair*. By commodity, we also say that all schedules are *distributed*, by opposition to central schedules.

Those properties of schedules directly transfer to schedulers. For example, we can consider the central strongly fair scheduler, which contains all the schedules that are both central and strongly fair.

2.4.4 Relation Between Schedulers

A scheduler $\mathcal{D}_1$ is stronger (*resp.* weaker) than a scheduler $\mathcal{D}_2$ if $\mathcal{D}_1$ can simulate $\mathcal{D}_2$ (*resp.* if $\mathcal{D}_1$ can be simulated by $\mathcal{D}_2$). More formally, if $\mathcal{D}_1 \supseteq \mathcal{D}_2$ (*resp.* if $\mathcal{D}_1 \subseteq \mathcal{D}_2$). Intuitively, the more possible executions there are, the stronger the adversary.

Note that not all schedulers can be compared. For example, the synchronous scheduler and the central scheduler, both weaker than the distributed scheduler, cannot be compared with each other.

2.5 Problems Specification

2.5.1 Specification

Let R and Q be two boolean predicates over configurations. We note $\gamma \in R$ when R is evaluated to `true` in γ, and $\gamma \notin R$ otherwise. We denote by `true` the predicate that always evaluates to `true`. Given an algorithm $\mathcal{A}$, the predicate R is *closed* for $\mathcal{A}$ if for every computing step $\gamma \underset{\mathcal{A}}{\rightarrow} \gamma'$, such that $\gamma \in R$, then $\gamma' \in R$. Algorithm $\mathcal{A}$ *converges* to predicate Q from predicate R under the scheduler $\mathcal{D}$ if Q is closed and if for any execution $\epsilon = (\gamma_0 \underset{\mathcal{A}}{\rightarrow} \gamma_1 \underset{\mathcal{A}}{\rightarrow} \cdots)$ under $\mathcal{D}$ such that $\gamma_0 \in R$, there exists $i \geq 0$ such that $\gamma_i \in Q$. This is noted $R \triangleright_{\mathcal{A}\mathcal{D}} Q$. If no confusion can be made, we simply write $R \triangleright_{\mathcal{A}} Q$ or even $R \triangleright Q$. We say that R is an *attractor* if $\texttt{true} \triangleright R$. When it is clear in the context, we indifferently use a predicate and the set of configurations it describes. For example, we can write Γ instead of `true`.

The specification SP_P of a problem P is a predicate over the executions and the network, which describes a specific behavior of the system. An algorithm $\mathcal{A}$ solves a problem P under a certain scheduler if every execution of $\mathcal{A}$ under that scheduler satisfies the specification of P.

The specification of a problem might be described through the use of *local predicates*. Intuitively, a local predicate is a boolean function evaluated by one node, with the same information as when the node evaluates the code of the algorithm: its own local constants, its own state, and the states of its neighbors.

Definition 2.1 (*Local Predicate*)

We call local predicate any boolean predicate which takes as inputs a set of local constants, and a set of one or several states of nodes.

If P is a local predicate and $v \in V$, we denote $P^\gamma(v) = P(\texttt{local}_v, \texttt{state}^\gamma_{N[v]})$.

2.5.2 Examples of Problems

We shortly present some common problems, and especially those that are studied in this thesis.

- The *Leader Election* problem, denoted $\mathcal{LE}$, is a fundamental problem that consists in determining one single, elected, node in the network. Solving this problem is basically making semi-uniform a uniform network, and simplify dealing with the issues related to concurrency, by allowing only one node to make critical decisions. We specifically address $\mathcal{LE}$ in Chapter 3.

- The *Spanning Tree Construction* problem, denoted $\mathcal{ST}$, consists in building a spanning tree structure in the network. Having a spanning tree structure in the network is particularly useful for minimizing the number of exchanged messages when propagating information in the whole network.

- The *Vertex Coloring* problem, or simply *Coloring* problem, denoted $\Delta - \mathcal{C}$, consists in associating to each node a color, usually an integer between 1 and Δ, such that no two neighbors have the same color. Coloring a graph is a suitable tool to simulate $\mathcal{S}_{\mathsf{PK}}$ in $\mathcal{S}_{\mathsf{PU}}$ with an additional cost of $\Omega(\Delta)$ bits of memory per node. Indeed, nodes of a vertex-colored network can address information to one of their neighbors by prefixing the message by the color of the intended neighbor. Coloring a network also captures a local mutual exclusion problem, which corresponds to situations where neighbors share a resource or a service which cannot be used by several devices at a time, to the allocation of frequency bands for WiFi antennas...

- The *Maximal Independent Set* problem, denoted $\mathcal{MIS}$, consists in selecting a subset of all the nodes, such that no two neighbors are selected, and such that the obtained subset is maximal: all the nodes that are not selected have at least one neighbor which is. A maximal independent set is a structure that is notably adapted to time-varying graphs, since it is relatively simple to maintain, and offers a lite but complete coverage.

- The *Unison* problem, denoted $\mathcal{U}$, requires the presence of a variable clock on each node, and consists in increasing all of these variables infinitely often, while maintaining each pairs of neighboring clocks with a difference at most one. Unison guarantees a minimal form of synchrony in an asynchronous network. Notably, a correct unison guarantees that even under an unfair scheduler, all nodes are regularly activated. We specifically address $\mathcal{U}$ in Chapter 4 and especially in Section 4.3.

- The *Termination Detection* problem, denoted $\mathcal{TD}$, consists in detecting when another algorithm executed on the same network has converged. In numerous situations, network protocols are built as a pile of interacting algorithms. If one of the layers is critical, then it might be necessary to be able to guarantee that the service it provides is actually satisfied before using it. This problem boils down to checking whether the algorithm of the critical layer has terminated. We specifically address $\mathcal{TD}$ in Chapter 4.

- The *Token Circulation* problem (or *Fair Token Circulation* problem), denoted $\mathcal{TC}$, is a fundamental problem that consists in guaranteeing that a unique token travels the network perpetually, visiting every node repeatedly. This problem models the objective of perpetually and fairly allocating a resource or a service to the devices of a distributed system, while insuring global mutual-exclusion. We specifically address $\mathcal{TC}$ in Chapter 5.

2.5.3 Classes of Problems

Terminating and Non-Terminating Problems Some problems such as $\mathcal{TC}$, or $\mathcal{U}$, require a perpetual execution of the algorithm. In particular, no finite execution can satisfy the specification of such problems. Those problems are said *non-terminating*.

On the contrary, there are problems, such as $\mathcal{LE}$, $\mathcal{ST}$, or $\mathcal{MIS}$ which only aim to provide the network a particular configuration. For those problems, the specification is of the form $SP_P(\epsilon) \equiv \forall \gamma \in \epsilon, P(\gamma)$, where P is a predicate over the configurations and the network. Those problems are said *terminating*. Note that some algorithms may require infinite executions to solve terminating problems.

Request/Answer Problems Some problems, such as $\mathcal{TD}$, depend on an external activation, a *request*. When the request is made, then the algorithm starts an execution, which ends when an *answer* is emitted, as a response to the request. Those problems generally suppose that one variable of the algorithm is shared by another application. This variable is dedicated to the communication of the request and the answer between the two algorithms. Such problems are called *request/answer* problems, or *request-based* problems.

2.6 Stabilization

Distributed systems are prone to transient faults. Most commonly, some messages can be lost, or corrupted. If such a failure occurs, the system may reach an incorrect configuration. Therefore, algorithms designed to perform in such environments must be fault-tolerant. Several paradigms were introduced in the literature to capture that notion. We only present the variants that we use in this work.

Distributed Algorithms Some distributed algorithms are not effective in faulty environments. Namely, they suppose that the initial configuration of the system is non faulty, typically that the variables of the nodes are properly initiated to a blank value. Such algorithms converge from this initial configuration, but not necessarily from a faulty, inconsistent configuration. Those algorithms, with no fault-recovering properties, are simply called *distributed algorithms*. In this thesis, we only consider algorithms which can handle faults, but we sometime compare ourselves to the simpler case of distributed algorithms.

Silent Self-stabilization There exist problems, such as the leader election or the spanning tree construction, for which the specification tolerates constant executions. Indeed, as soon as one single leader is elected, there is no *a priori* need to keep computing. An algorithm $\mathcal{A}$ is *silent* [DGS99] for problem P under scheduler $\mathcal{D}$ if every maximal execution of $\mathcal{A}$ is finite. Silent self-stabilization is a paradigm particularly adapted to terminating problems.

Definition 2.2 (*Silent*)

A distributed algorithm $\mathcal{A}$ is silent if all the maximal executions of $\mathcal{A}$ are finite.

Self-stabilization For most situations, the concept of *self-stabilization* introduced by Dijkstra [Dij74] is a suitable paradigm. An algorithm $\mathcal{A}$ is *self-stabilizing* for one problem P under scheduler $\mathcal{D}$ if there exists a predicate R such that $\Gamma \triangleright_{\mathcal{AD}} R$ and such that any execution of $\mathcal{A}$ under $\mathcal{D}$ starting from a configuration $\gamma \in R$ satisfies SP_P. In other words, a self-stabilizing algorithm converges to a legitimate execution whatever the initial configuration of the system. Self-stabilization is a paradigm suited to distributed systems prone to relatively rare transient failures.

Definition 2.3 (*Self-stabilization*)

Algorithm $\mathcal{A}$ is a self-stabilizing algorithm for problem P under a certain scheduler $\mathcal{D}$ if there exists a predicate R such that $\Gamma \triangleright_{\mathcal{AD}} R$ and such that any execution of $\mathcal{A}$ starting from a configuration $\gamma \in R$ satisfies SP_P.

An execution of $\mathcal{A}$ has stabilized *once R is valid. The stabilization time of $\mathcal{A}$ is the maximal number of rounds in executions of $\mathcal{A}$ starting from any configuration, before $\mathcal{A}$ stabilized.*

Snap-stabilization Although self-stabilization is a suitable paradigm in most situations, there exist problems for which this condition is not sufficient. If we consider problems that depend on an output, typically observation problems, then self-stabilization is not adapted. Indeed, self-stabilization ensures that execution ultimately converges to one predicate R from which executions are correct. Yet, nothing guarantees that no incorrect output is provided before the convergence to R. Such incorrect outputs might be detrimental to the system, which makes self-stabilization non-suitable in this situation. An algorithm $\mathcal{A}$ is *snap-stabilizing* [BDPV07] for one problem P under scheduler $\mathcal{D}$ if any execution of $\mathcal{A}$ under $\mathcal{D}$ satisfies SP_P. In other words, if as soon as there are no more faults, then the system immediately behaves correctly. Snap-stabilization is a paradigm especially suitable for request-based problems.

Definition 2.4 (*Snap-Stabilization*)

A distributed algorithm $\mathcal{A}$ is snap-stabilizing for a specification SP if any maximal execution of $\mathcal{A}$ starting from any configuration satisfies SP.

2.7 Efficiency of an Algorithm

We have already seen above that numerous criteria can be studied to discuss the genericity of an algorithm. Those criteria are the presence of (unique or not) identifiers, the uniformity of the network, the (precise or bounded) knowledge of some parameters of the network (size, degree, diameter...), the topologies on which the algorithm works (rings, trees, all graphs...), the knowledge of nodes of the port number affected to them by their neighbors ($\mathcal{S}_{\mathsf{PU}}$ or $\mathcal{S}_{\mathsf{PK}}$), the determinism of the algorithm, the hypothesis made on the scheduler (fairness, centrality...), and the capacity to handle transient faults.

In addition to those criteria, the complexity of the execution of an algorithm is one other crucial parameter to take into account. Algorithms executed on a central system are evaluated based on two criteria: the memory required and the convergence time. Algorithms executed on distributed systems are also evaluated on these criteria.

In most cases, the limiting parameters do not come from the computing units, but from the communication links of the network, bandwidth, latency, *etc.* Thus, the relevant quantities of a distributed algorithm are not the complexity of the algorithm on one node, but the global complexity of the execution of the algorithm in the network. Although

this does not bring much trouble when examining the spatial complexity, it requires some nuance when we come to time complexity.

Spatial and time complexities are expressed as functions, whose parameters are parameters of the network. The most common parameters of complexity functions are the size, diameter, and degree of the network (n, D, and Δ).

2.7.1 Comparison Functions

What is interesting when studying spatial and time complexities is their asymptotic behavior. To express this, we use the following standard notations:

$$
\begin{array}{lll}
f(x) \in o(g(x)) & \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 & \\
f(x) \in O(g(x)) & \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty & \\
f(x) \in \Omega(g(x)) & \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0 & \iff \neg\big(f(x) \in o(g(x))\big) \\
f(x) \in \omega(g(x)) & \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty & \iff \neg\big(f(x) \in O(g(x))\big) \\
f(x) \in \Theta(g(x)) & \iff 0 < \lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty & \iff f(x) \in O(g(x)) \wedge f(x) \in \Omega(g(x))
\end{array}
$$

2.7.2 Spatial Complexity

The relevant quantity for the spatial efficiency of an algorithm executed on a distributed system is the size of the messages necessary to complete the task. In the state model, the messages are abstracted by the ability for nodes to directly read the mutable memory of their neighbors. Thus, the size unit of the messages which are exchanged on the network is the size of the mutable memory the algorithm requires, on each node. Note that the identifier of the node is part of the immutable memory, and thus its size is not taken into account in spatial complexity, unless the algorithm explicitly writes it into the mutable memory to share it with the neighborhood.

We call spatial complexity of an algorithm $\mathcal{A}$, and denote $\mathbb{S}_{\mathcal{A}}$, the number of bits of mutable memory required by $\mathcal{A}$ on each node.

2.7.3 Time Complexity

To estimate the time efficiency of a distributed algorithm, focusing on the time complexity of the execution of one action on one node is not relevant for two reasons. The first one is that this time is most often negligible when compared to the communication time between the devices of the system. The second is that what takes time in the execution of a distributed algorithm is the number of communications necessary for the convergence of the system. We define the time complexity of an algorithm $\mathcal{A}$, denoted $\mathbb{T}_{\mathcal{A}}$, as the number of time units it requires before resolving the problem.

Time Units Due to asynchrony, some nodes whose action is necessary for the algorithm, can be inactivated during a long time, and block the execution of the algorithm. To embrace such situations, there are two definitions of what a time unit is.

The notion of *step* corresponds to the number of separated activation of nodes by the scheduler: the number of computing steps.

On the other hand, the notion of *round* [BDPV07] captures the execution rate of the slowest processor. Let us state this more formally. A node v is *neutralized* in the computing step $\gamma \underset{\mathcal{A}}{\rightarrow} \gamma'$ if v is enabled in γ and not enabled in γ', but does not execute any action between these two configurations. Neutralization occurs when some neighbors of v changed

their state between γ and γ', and this change makes the guards of all actions of v false. Let ϵ be an execution.

The first round of an execution ϵ, noted ϵ', is the minimal prefix of ϵ in which every node that is enabled in the initial configuration either executes an action or becomes neutralized. Let ϵ'' be the suffix of ϵ starting from the last configuration of ϵ'. The second round of ϵ is the first round of ϵ'', and so forth.

Time of Resolution The notion of resolving a problem depends on the stabilization paradigm considered.

If we consider self-stabilization, then the resolution corresponds to the convergence of the algorithm: the moment where the specification of the problem is satisfied. We call this quantity the *time of convergence*.

If we consider silent self-stabilization, then the resolution corresponds to the termination of the system: when it reaches a terminal configuration. We call this quantity the *time of termination*.

Snap-stabilizing algorithms are, by definition, self-stabilizing algorithms whose time of convergence is 0, which makes this quantity non-relevant. In most cases, snap-stabilizing algorithms are designed to solve request/answer problems. In such situations, a suitable parameter is the *time of response*, which is the number of time units between the moment a request is made, and the moment the algorithm answers to this request. This notion can be refined depending on the problem considered.

Chapter 3

Lower Bound for Spatial Complexity

> You don't have any concept of what it is to hit the bottom
>
> Tyler

Contents

In this chapter, we introduce a generic $O(\log \log n)$ lower bound for the space complexity of distributed, deterministic algorithms, and especially for the leader election problem.

A preliminary version of these works was presented in [BFB21].

3.1 Introduction

It is known that the presence of unique identifiers at the nodes of a network is very helpful, and sometimes necessary, to break symmetric configurations. Some fundamental problems, such as Leader Election or Spanning Tree Construction, are known to be unsolvable by deterministic algorithms in generic anonymous networks [Ang80, Dij82]. On the other hand, numerous distributed, deterministic, self-stabilizing algorithms exist for those problems in identified networks [BGJ99, DGS99, DLV11a, DLV11b, BT18]. Although it is known that identifiers are required to solve such problems, the question of how compact the use of those identifiers can be made remains. Most algorithms for identified networks are such that nodes directly share their identifier with their neighbors by writing it into their mutable memory, and thus have a spatial complexity in $\Omega(\log n)$ bits per node. Recently, a self-stabilizing algorithm was presented to address Leader Election, which requires only $O(\log\log n+\log\Delta)$ bits per node [BT20]. This indicates that $O(\log\log n + \log\Delta)$ bits per node are sufficient to use the identifiers. On the other hand, it is known that Leader Election cannot be solved with constant space complexity [BGJ99]. This indicates that one cannot fully use the power proper to identified networks with $O(1)$ bits per node. However, the question of how many memory is necessary for algorithms to use the identifiers was still open. We prove a $\Omega(\log\log n)$ bits per node lower bound for that problem. More precisely, on identified, bounded-degree graphs, algorithms that use $o(\log\log n)$ bits of memory per nodes behave exactly the same way as if they were executed on anonymous networks. Since a large variety of problems, and especially Leader Election, are unsolvable in bounded-degree anonymous networks, we can conclude a $\Omega(\log\log n)$ bits per node lower bound for Leader Election.

3.1.1 Motivation

During the execution of a self-stabilizing algorithm, the nodes exchange information along the links of the network, and this information is stored locally at every node. Specifically, processes in a distributed system have two types of memory: the *persistent* memory, and the *mutable* memory. The persistent memory is used to store the identity of the process (e.g., its IP address), its port numbers, and the code of the algorithm executed on the process. Importantly, this section of the memory is not write enabled during the execution of the algorithm. As a consequence it is less likely to be corruptible, and most work in self-stabilization assumes that this part of the memory is not subject to failures. The mutable memory is used to store the variables used by the algorithm, and is subject to failures, that is, to the corruption of these variables. The space complexity of a self-stabilizing algorithm is the total size of all the variables used by the algorithm.

Preserving small space complexity is very much desirable, for several reasons. First, it is expected that self-stabilizing algorithms offer some form of universality, in the sense that they are executable on several types of networks. Networks of sensors as used in IoT, as well as networks of robots as used in swarm robotics, have the property to involve nodes with limited memory capacity, and distributed algorithms of large space complexity may not be executable on these types of networks. Second, a small space complexity is the guarantee to consume a small bandwidth when nodes exchange information, thus reducing the overhead due to link congestion [ANT12]. In fact, a self-stabilizing algorithm is never terminating, in the sense that it keeps running in the background in case a failure occurs, for helping the system to return to a correct configuration. Therefore, nodes may be perpetually exchanging information, even after stabilization, and even when no faults occur. Limiting the amount exchanged information, and thus, in particular, the size of the variables, is therefore of the utmost importance for optimizing time, and even energy. Last but not least, increasing robustness against variable corruption can be achieved by data replication [HP00]. This

is, however, doable only if the variables are reasonably small. Said otherwise, for a given memory capacity, the smaller the space complexity the larger the robustness thanks to data replication.

3.1.2 Related Work

Space complexity of self-stabilizing algorithms has been extensively studied for *silent* algorithms, that is, algorithms that guarantee that the content of the variables of every node does not change once the algorithm has reached a correct configuration. For silent algorithms, Dolev and al. [DGS99], proved that finding the centers of a graph, electing a leader, and constructing a spanning tree require registers of $\Omega(\log n)$ bits per node. Silent algorithms have later been related to a concept known as *proof-labeling scheme* (PLS) [KKP10]. Any lower bound on the size of the proofs in a PLS for a predicate P on graphs implies a lower bound on the size of the registers for silent self-stabilizing algorithms solving P. A typical example is the $\Omega(\log^2 n)$-bit lower bound on the size of any PLS for minimum-weight spanning trees (MST) [KK07] , which implies the same bound for constructing an MST in a silent self-stabilizing manner [BF15]. Thanks to the tight connection between silent self-stabilizing algorithms and proof-labeling schemes, the space complexity of a vast collection of problems is known, for silent algorithms. (See [FF16] for more information on proof-labeling schemes.)

On the other hand, to our knowledge, the only lower bound on the space complexity for general self-stabilizing algorithms (without the requirement of being silent) that corresponds to our setting has been established by Beauquier, Gradinariu and Johnen [BGJ99] who proved that registers of constant size are not sufficient for leader election algorithms. Interestingly, the same paper also contains several other space complexity lower bounds for models different from ours – e.g., anonymous networks, or harsher forms of asynchrony. Although there are very few lower bounds for the model we consider, there exist several impossibility results for specific topologies. In the case of anonymous networks, where nodes do not have a unique identifier, Angluin [Ang80] proved that there does not exist any self-stabilizing algorithm for strict leader election (*i.e.* such that there is never more than one leader), even under a central scheduler and even allowing randomization, due to the general impossibility to break symmetry in such networks. Later, Dijkstra [Dij82] proved that there does not exist any self-stabilizing algorithm for leader election on the anonymous ring, unless the size of the ring is a prime number. A slightly more powerful model than anonymous networks is homonymous networks, where identifiers may be shared by several nodes. In [FKK+04], the authors investigate the problem on solving leader election on homonymous rings. They prove that leader election is feasible if and only if there is no non-trivial symmetry in the distribution of the identifiers around the ring. In [DGFTT14] the authors propose a necessary and sufficient condition on the number of distinct labels in bidirectional homonymous rings to solve terminating leader election. They show that there is a solution if and only if the number of distinct identifiers is greater than the highest divisor of n. Later, [ADD+20] generalizes the previous by establishing impossibility results for leader election in some unidirectional homonymous rings. Note that the impossibility results of [DGFTT14] and [ADD+20] are established for general distributed algorithms, without any self-stabilization requirements.

The literature dealing with upper bounds is far richer. In particular, [BT18] recently presented a self-stabilizing leader election algorithm using registers of $O(\log\log n)$ bits per node in n-node rings. This algorithm was later generalized to networks with maximum degree Δ, using registers of $O(\log\log n + \log\Delta)$ bits per node [BT20]. It is worth noticing that spanning tree construction and $(\Delta+1)$-coloring have the same space complexity $O(\log\log n + \log\Delta)$ bits per node [BT20]. Prior to these works, the best upper bound was

a space complexity $O(\log n)$ bits per node [BT18], and it has then been conjectured that, by some iteration of the technique enabling to reduce the space complexity from $O(\log n)$ bits per node to $O(\log \log n)$ bits per node, one could go all the way down to a space complexity of $O(\log^* n)$ bits per node. Arguments in favor of this conjecture were that such successive exponential improvements have been observed several times in distributed computing. A prominent example is the time complexity of minimum spanning tree construction in the congested clique model [LPPP05, HPP+15, GP16, JN18]. Complexities $O(\log^* n)$ are not unknown in the self-stabilizing framework [AO94], and it seemed at first that the technique in [DGS99] could indeed be iterated (in a similar fashion as in [BKN17]). Our result shows that this is not the case, and somewhat closes the question of the space complexity of leader election.

3.1.3 Contributions

In this chapter, we establish several lower bounds of $\Omega(\log \log n)$ bits per node for the space complexity of deterministic distributed algorithms that require unique identifiers. Namely, we establish a link between executions of algorithms in identified networks, and executions of algorithms in weaker types of networks.

We first show that, as soon as identifiers are taken in a polynomial range $[1, n^c]$, with $c \in \mathbb{R}, c > 1$, then algorithms that use $o(\log \log n)$ bits per node are not more powerful than anonymous algorithms. More precisely, let $\mathcal{A}$ be an algorithm in a network with unique node identifiers, and let us assume that $\mathcal{A}$ has space complexity $o(\log \log n)$ bits per node. We show that there exist graphs and assignments of identifiers to the nodes of these graphs such that, in these graphs and for these identifier-assignments, $\mathcal{A}$ has the same behavior as an algorithm executed in these graphs but where all nodes share the same identifier (i.e. in the anonymous version of these graphs).

Be aware that a space complexity in $o(\log n)$ does not prevent $\mathcal{A}$ from exchanging identifiers between nodes, but they must be transferred as a series of smaller pieces of information that are pipelined along a link, each of size $o(\log n)$ bits. Yet, a node cannot store the identifier of even just one of its neighbors.

We then slightly modify our proof to obtain a second, similar, theorem. We show that, for any $k \in \mathbb{N}, k \geq 2$, as soon as identifiers are taken in a linear range $[1, n \times c]$, with $c \in \mathbb{R}, c > 1$, then algorithms that use $o(\log \log n)$ bits per node are not more powerful than k-homonymous algorithms. More precisely, let $\mathcal{A}$ be an algorithm in a network with unique node identifiers, and let us assume that $\mathcal{A}$ has space complexity $o(\log \log n)$ bits per node. We show that, with spacial complexity $o(\log \log n)$ bits per node, there exist graphs and assignments of identifiers to the nodes of these graphs such that, in these graphs and for these identifier-assignments, $\mathcal{A}$ has the same behavior as an algorithm executed in these graphs but where identifiers can be shared by several nodes (i.e. in the k-homonymous version of these graphs).

Both results have a very broad scope of application. Indeed, we make almost no hypothesis on the model in which $\mathcal{A}$ is executed. We do not make any assumption on the communication model ($\mathcal{S}_{\mathsf{PU}}$ or $\mathcal{S}_{\mathsf{PK}}$), neither on the knowledge of the topology (precise or bounded knowledge of n, Δ, D), neither on uniformity of the network, neither on the scheduler under which it is executed, and a weak assumption is made on the topology on which the algorithm is executed. Practically, one can fix any parametrization for the model, our theorem guarantees an equivalence between executions of $\mathcal{A}$ in identified and anonymous or homonymous networks, under the same chosen model.

What our result is really about is the power of identifiers in a scenario where very little space/communication is used. Remember that the Naor-Stockmeyer order-invariance theorem [NS95] states that in the LOCAL model, for local problems, constant-time algorithms

that use the exact values of the identifiers are not more powerful than the order-invariant algorithm that only uses the relative ordering of the identifiers. In some sense our paper and [NS95] have the same take-home message, in two different contexts: if you do not have enough resources, you cannot use the (full) power of the identifiers.

Note that both previous results establish an equivalence between executions on identified networks, and executions on weaker networks. Yet it is unclear whether this extends to the resolution of some problems. Indeed, although we prove that the behavior of the algorithm cannot fully use the power of identifiers, it may be that the specification predicates uses the identifiers. In such a case, it could be that in an identified network, the specification is valid thanks to the identifier, while in the exact same configuration of an anonymous, or k-homonymous, network, it is not. Fortunately, in most cases, the specification of the problem is simple enough for this issue to be easily overcome.

As a third result, we show how the previous can be adapted to prove lower bound of $\Omega(\log\log n)$ bits per node for a problem, by addressing the Leader Election problem. This improves the only lower bound known so far (see [BGJ99]), which states that leader election has non-constant space complexity, i.e., complexity $\omega(1)$. More importantly, our bound matches the best known upper bound on the space complexity of leader election, which is $O(\log\log n)$ bits per node in bounded degree networks [BT20], and in particular invalidates the folklore conjecture stating that leader election is solvable using only $O(\log^* n)$ bits of mutable memory per node.

The technique used to adapt the previous results to prove a lower bound for $\mathcal{LE}$ is very generic, and works for basically any problem that requires minimal symmetry breaking. In Chapter 5 we reuse the previous to prove the space-optimality of our token-circulation algorithm.

Chapter Outline The remainder of this chapter is organized as follows. We first formally describe the particular models in which we work, provide some definition, and the specification of the leader election problem in Section 3.2. Then, in Section 3.3 we show on a simple example the reasoning used in our proofs. Finally, in Sections 3.4, 3.5, and 3.6 we formally prove our three theorems.

3.2 Model

3.2.1 Indistinguishability

The proofs of Sections 3.4, 3.5 and 3.6 are based on the notion of indistinguishability. In this work, we only consider indistinguishability for networks that have the same underlying graph. Two networks that have the same underlying graph are indistinguishable for an algorithm $\mathcal{A}$ if the computing steps of $\mathcal{A}$ are exactly the same on both networks.

Definition 3.1 (*Indistinguishability*)

Let G_1 and G_2 be two networks with identical topology, i.e. whose underlying graph is the same, and let $\mathcal{A}$ be a distributed algorithm.

G_1 and G_2 are indistinguishable for $\mathcal{A}$ if for any two configurations γ and γ', $\gamma \to \gamma'$ is a valid computing step of $\mathcal{A}$ on G_1 if and only if it is a valid computing step of $\mathcal{A}$ on G_2.

Remark that, by transitivity, if G_1 and G_2 are indistinguishable for $\mathcal{A}$ then, for any scheduler $\mathcal{D}$, the executions of $\mathcal{A}$ on G_1 under $\mathcal{D}$ are the same than the executions of $\mathcal{A}$ on G_2 under $\mathcal{D}$.

3.2.2 Degree-limited

The proofs of Sections 3.4 and 3.5 are only effective on graphs with relatively small degree. Let us give a proper definition of how to asymptotically define this notion of small degree:

Definition 3.2 (Δ-*limited*)

Let $\mathcal{G} = (G_i)_{i\in\mathbb{N}}$ be a class of graphs which contains graphs of arbitrary large size. Let us call $\Delta_{\mathcal{G}}(n)$ the function that associates to each integer n, the maximal degree of graphs of size n in $\mathcal{G}$. If no graph of size n exists in $\mathcal{G}$ we define $\Delta_{\mathcal{G}}(n) = 0$. If there is no bound on the degree of graphs of size n in $\mathcal{G}$ we define $\Delta_{\mathcal{G}}(n) = \infty$.

We say that $\mathcal{G}$ is Δ-limited by function f if $\Delta_{\mathcal{G}}(n) \in o(f(n))$.

3.2.3 Model for Sections 3.4 and 3.5

Our two first results are highly generic, in the sense that they can be adapted to a large variety of situations. We prove that whatever are the precise state model $\mathcal{S}_{\mathsf{PU}}$ or $\mathcal{S}_{\mathsf{PK}}$, the scheduler, the knowledge of the parameters of the network, the uniformity of the network, or the presence of faults, executions of some deterministic algorithm with small memory on large, identified networks are also executions of the same algorithm on an anonymous or homonymous network. We only make 2 hypothesis, on the range in which are taken the identifiers, and on the degree of the networks that are involved.

Range for identifiers We need to consider identifiers between 1 and $n \times c$, for $c > 1$, at least. This is not an artifact of our proofs: it is actually necessary for the results to hold. Indeed, if the identifier range is $[1, n]$, then an algorithm can attribute specific rules to each node of the network, and being certain that no set of rules will be missing. For example, even in a uniform network, an algorithm may use the node with identifier 1 as a designated node, and thus making the network semi-uniform-like. This operation could not be done in the anonymous version of the network. Since semi-uniform algorithms can achieve space complexity below our lower bound [DJPV00, Joh97], this excludes the possibility to extend our results to identifier range $[1, n]$. Even without the possibility of having a trivially designated node, one can take advantage of small identifier range: there actually exists a leader election algorithm for the ring which requires constant memory if the identifiers are taken in $[1, n + c]$ for a constant c [BGJ99]. As soon as the identifiers are taken in a linear range $[1, n \times c]$, no such trick can be made to attribute a specific role to one node.

Limited-degree graphs Our proofs only apply on networks with relatively small degree. Namely, it works with graphs with degree $o(\log n)$. This may seem very restrictive, but practically it is not. Indeed, most problems that cannot be solved on general anonymous networks can neither be solved on anonymous networks with small degree. Therefore, proving an equivalence between the executions of algorithms on identified and anonymous, small-degree, networks, is generally sufficient to prove that some problem cannot be solved with $o(\log \log n)$ bits per node.

3.2.4 Model for Section 3.6

In Section 3.6 we prove the following theorem:

Theorem 3.1

Let $c \in \mathbb{R}, c > 1$. Every deterministic distributed algorithm solving leader election in the

state model under a central strongly fair scheduler or under the synchronous scheduler requires registers of size at least $\Omega(\log \log n)$ bits per node in n-node composite uniform rings with unique identifiers in $[1, n \times c]$.

Our lower bound is actually optimal not only in terms of size, but also in terms of the assumptions we make on the setting. More precisely, our theorem has four restrictions: it works for deterministic algorithms only, on rings with a non prime number of nodes, rings that are uniform, and with identifiers in a large enough range. In particular, we do not make any real assumption on the scheduler, since the central strongly fair scheduler and the synchronous scheduler are the weakest schedulers in the literature, nor on the knowledge on the topology, nor on the variant of the state model.

We will now discuss why those four limitations are actually necessary.

Determinism Randomization is a common tool for symmetry breaking, and our problem is one example. Namely, [IL94] proved that using randomization, one can solve leader election using constant memory, which implies that our result cannot be generalized in that direction.

Topology Our proof is established on composite rings, *i.e.* rings whose size n is not a prime number. Therefore, it applies to any algorithm that solves Leader Election on, at least, composite rings. Note that our result does not invalidate the possibility to design constant-size algorithms for Leader Election in some particular topologies. In stars, exactly one node has more than one neighbor. Therefore, stars can be seen as particular semi-uniform networks: the center of the star is a trivial designated leader. More interestingly, our result does not apply to algorithms that solve leader election on prime rings, *i.e.* rings whose size n is a prime number. There is no hope to extend our result in that direction, since [ILS95] builds a constant memory algorithm for leader election on anonymous prime rings, under a central scheduler.

Remark that proving an impossibility result for the sub-class of composite rings is particularly interesting since a large number of works have focused on solving the problem of the leader election in rings [ILS95, BGJ99, DGFTT14, BT18, ADD+20]. We prove that even under that pretty simple topology, one cannot design an algorithm that uses $o(\log \log n)$ bits per node.

Identifier Range Since Leader Election is impossible in general homonymous networks, the equivalence result of Section 3.5 is sufficient to establish our lower bound. Thus, we only need to suppose that identifiers are taken in a linear range, $[1, n \times c]$, for $c \in \mathbb{R}, c > 1$ to obtain our impossibility result. Due to [BGJ99] constant-memory algorithm for Leader Election when identifiers are taken in $[1, n + c]$, for $c \in \mathbb{N}$, there is no hope to extend our theorem to asymptotically smaller range of identifiers.

Uniformity Our proof is only valid in uniform networks. There is no hope to extend our results to semi-uniform networks since $\mathcal{LE}$ is trivial in anonymous, semi-uniform networks. Indeed, the distinguished node of the network is a trivial designated leader.

3.2.5 Leader Election Problem

In Section 3.6, we focus on one of the arguably most important problems in the context of distributed computing, namely *leader election*. The objective is to maintain a unique leader

in the network, and to enable the network to return to a configuration with a unique leader in case there are either zero or more than one leader.

One first step to define the specification of $\mathcal{LE}$ is to express what being a leader means. Some papers suppose a variable $\texttt{leader}_v$ on each node v, that can only take two values, 1 and 0, 1 being the value for the leader and 0 for the others. In this case, solving $\mathcal{LE}$ boils down to reaching a configuration where exactly one node v has its variable $\texttt{leader}_v$ set to 1.

Although pretty intuitive, this definition is a bit restrictive. Indeed, it could be that nodes do not store in a variable the information whether they are the leader or not. A more general possibility is that each time it is necessary, nodes compute a value which indicates whether they are the leader or not. This computation may depend on their local variables, their state, and the state of their neighbors, according to the state model. To be as generic as possible, we consider the second option, which implies the use of a local predicate.

We can now define the specification of $\mathcal{LE}$ by referring to the unicity of the leader. A configuration $\gamma \in \Gamma$ is a correct configuration for the leader election problem if one single node is elected, and an execution satisfies the specification of $\mathcal{LE}$ if any configuration it contains is a correct configuration for the leader election problem. More formally:

Specification 3.1 (*Leader Election*)

$\mathcal{LE}$ (Leader Election) in $G = (V, E)$ is specified by the existence of a local predicate $E_{\mathcal{LE}}$.

A configuration γ is correct if

$$P_{\mathcal{LE}}(\gamma) \equiv \exists! v \in V : E^{\gamma}_{\mathcal{LE}}(v) = \texttt{true}.$$

An execution $\epsilon = \gamma_0 \to \cdots$ is correct if all the configurations γ_i are correct:

$$\begin{aligned} SP_{\mathcal{LE}}(\epsilon) \equiv &\quad \forall i \geq 0, P_{\mathcal{LE}}(\gamma_i) \\ \equiv &\quad \forall i \geq 0, \exists! v \in V : E^{\gamma_i}_{\mathcal{LE}}(v) = \texttt{true}. \end{aligned}$$

Remark that, once again, our definition is extremely general. Notably, we do not suppose that the leader remains leader during the entire execution, the leadership can switch from one node to another.

Since we prove a lower bound for $\mathcal{LE}$, being as permissive as possible only strengthens our result.

3.3 Intuition of the proofs

3.3.1 Challenge of lower bounds for non-silent algorithms

Almost all lower bounds for self-stabilization are for silent algorithms, which are required to stay in the same configuration once they have stabilized. These lower bounds are then about a static data structure, the stabilized solution. The question boils down to establishing how much memory is needed to locally certify the global correctness of the solution, and this is well-studied [Feu19].

When we do not require that the algorithm should converge to one correct configuration, and stay there, there is no static structure on which we can reason. It is then unclear how we can establish lower bounds. One way is to think about invariants. Consider a property that we can assume to hold in the initial configuration, and that is preserved by the computation

(if it follows some memory requirement hypothesis). If no correct output configuration has this property, then we can never reach a correct output configuration.

In our proof, the property that will be preserved is that every configuration is symmetric. This can clearly be assumed for the original configuration, and we show that basically if the memory is limited then this is preserved at each step. As the specification we use for leader election is that the leader should output 1, and the other nodes should output 0, then it is not possible that a proper output configuration is symmetric.

3.3.2 Intuition on a minimal example

Equivalence with anonymous and homonymous networks Let us now give some intuition about why algorithms for identified networks with small memory are effective in k-homonymous or anonymous networks as well. The code of an algorithm $\mathcal{A}$ for identified networks may refer to the identifier of the node that is running it. For example, a rule of the algorithm could be:

- if the states of the current node and of its left and right neighbors are respectively x, y, and z, then: if the identifier is odd the new state is a, otherwise it is b.

Now suppose you have fixed an identifier, and you look at the rules for this fixed identifier. In our example, if the identifier is 7, the rule becomes:

- if the states of the current node and of its left and right neighbors are respectively x, y, and z, then: the new state is a.

This transformation can be done for any rule, thus, for an identifier i, we can get an algorithm $\mathcal{A}_i$ specific to this identifier. When we run $\mathcal{A}$ on every node, we can consider that every node, with some identifier i is running $\mathcal{A}_i$. Note that $\mathcal{A}_i$ does not refer to the identifier in its code.

The key observation is the following. If the amount of memory an algorithm can use is very limited, then there is very limited number of different behaviors a node can have, especially if the code does not refer to the identifier. Let us illustrate this point by studying an extreme example: a ring on which states have only one bit. In this case the number of input configurations for a node is the set of views (x, y, z) as above, with $x, y, z \in \{0, 1\}$. That is there are $2^3 = 8$ different inputs, thus the algorithm can be described with 8 different rules. Since the output of the function is the new state, the output is also a single bit. Therefore, there are at most $2^8 = 256$ different sets of rules, that is 256 different possible behaviors for a node. In other words, in this extreme case, each specific algorithm $\mathcal{A}_i$ is equal to one of the behaviors of this list of 256 elements. This implies that, if we take a ring with 257 nodes, there exist two nodes with two distinct identifiers i and j, such that the specific algorithms $\mathcal{A}_i$ and $\mathcal{A}_j$ are equal.

But the idea above can be strengthened to get our theorem. The key is to use the hypothesis that the identifiers are taken from a large enough range. As we have a pretty large palette of identifiers, we can always find, not only 2, but n distinct identifiers in $[1, n \times c]$ that can be grouped such that the specific algorithm of all the nodes of the same group correspond to the exact same behavior. In this case, it is as if the network was k-homonymous, where k is the size of each group. If the identifiers are taken in $[1, n^c]$, we can even find n distinct identifiers, such that all the specific algorithms $\mathcal{A}_i$ correspond to the exact same behavior. In this case it is as if the network was anonymous.

Application: Lower Bound for $\mathcal{LE}$ As soon as the network is homonymous-like, we can start an execution from a symmetric configuration. If the scheduler always activates

all homonymous nodes consecutively, then the execution contains an infinity of symmetric configurations, and thus never stabilizes to a proper leader election execution. This is presented in Figure 3.1.

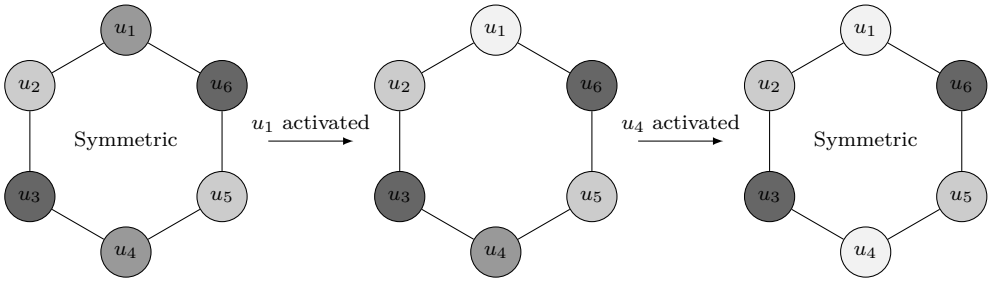

Figure 3.1: Example of two computing steps, with $n = 6$, $k = 2$, and $\mathcal{A}_{ID(u_1)} = \mathcal{A}_{ID(u_4)}$, $\mathcal{A}_{ID(u_2)} = \mathcal{A}_{ID(u_5)}$, $\mathcal{A}_{ID(u_3)} = \mathcal{A}_{ID(u_6)}$. Identical states are represented by identical shades of gray. If the states of the nodes are symmetric in the initial configuration, and if the behavior functions are symmetrically placed around the ring, then an execution can contain an infinity of symmetric configurations.

Note that the larger the memory is, the more different behaviors there are, and the smaller the set of identical specific algorithms we can find. This trade-off implies that the construction works as long as the memory is in $o(\log \log n)$.

3.4 Equivalence with Anonymous Networks

3.4.1 Statement of the Theorem

In this section, we prove the following theorem:

Theorem 3.2

Let $c \in \mathbb{R}, c > 1$. Let us consider $\mathcal{G} = (G_i)_{i \in \mathbb{N}}$ a class of graphs which contains graphs of arbitrary large size, such that $\mathcal{G}$ is Δ-limited by $\log n$.

Let $\mathcal{A}$ be a distributed algorithm that uses registers of size $o(\frac{\log \log n}{\Delta})$ on all identified networks $G_i^0[1, n^c]$, i.e. on all graphs G_i with unique identifiers in $[1, n^c]$.

For any large-enough graph $G_i \in \mathcal{G}$, there exist n identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ in $[1, n^c]$ such that $G_i^0[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n]$ and $G_i^n[\mathtt{ID}_1]$ are indistinguishable for $\mathcal{A}$.

In other words, there exist n identifiers in $[1, n^c]$ such that in G_i, $\mathcal{A}$ cannot distinguish those identifiers: it runs exactly as if the network was anonymous.

3.4.2 Proof in $\mathcal{S}_{\mathsf{PU}}$ on uniform networks

In this section we prove a weaker version of Theorem 3.2: we only consider the port-unknown state model and uniform networks. We show in Section 3.4.3 how this proof can be generalized to $\mathcal{S}_{\mathsf{PK}}$ and to semi-uniform networks.

Consider a graph G_i with maximal degree Δ, and an algorithm for identified networks using $f(n)$ bits of memory per node on graphs of $\mathcal{G}$. An algorithm can be seen as the function that describes the behavior of the algorithm. In the state model, the predicates of the different rules may consider the identifier of the node itself, its state, the degree of the

node, and the states of the neighbors of the node. The degree can actually be a parameter on which the algorithm relies. Algorithms, especially in anonymous networks, may have different behavior depending on the degree of the node. Depending on the value of the different parameters, an action will be executed, which produces an update of the state of the node.

Consequently, we model an algorithm by a function, which takes as an input, an identifier, a state for the node, the degree of the node, and a state for all of the neighbors of the node. This function produces as an output a new state for the node. Formally:

$$\begin{array}{rccccccccc} \mathcal{A}: & [n^c] & \times & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

The idea is that, to get its output, node v will first feed its identifier $\texttt{ID}_v$, its own state $S(v)$, then its own degree Δ_v and finally the states of its Δ_v first neighbors as the Δ_v next fields. The other fields are left blank. The output of the function is the output of the algorithm: the new state of the node.

Note that this corresponds to the port-unknown state model: the function does not have any information that allows distinguishing the actions taken by one node that is pointed by its port-number, and one node that is not pointed by its port number, both having the same neighboring.

Now we can consider that for every identifier i, we have an algorithm of the form:

$$\begin{array}{rccccccc} \mathcal{A}_i: & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (\text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

Thus a specific algorithm $\mathcal{A}_i$ boils down to a function of the form:

$$\mathcal{A}_i : \left(\{0,1\}^{f(n)}\right)^{\Delta+1} \times [\Delta] \to \{0,1\}^{f(n)}.$$

Let us call such a function a *behavior*, and let $\mathcal{B}_n$ be the set of all behaviors.

Lemma 3.1 counts the maximum number of distinct behaviors that can exist.

Lemma 3.1

$$|\mathcal{B}_n| \leq \left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}$$

Proof: The inputs are basically an integer between 1 and Δ, and binary strings of length $f(n)$, and there are up to $\Delta+1$ such strings. There are at most $2^{(\Delta+1)f(n)}$ possibilities for the strings, and Δ possibilities for the degree. Similarly the number of possible outputs is $2^{f(n)}$. Thus the number of functions in $\mathcal{B}_n$ is at most $\left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}$. ■

Lemma 3.1 implies that the smaller f, the fewer different behaviors. Let us make this more concrete with Lemma 3.2.

Lemma 3.2

If $f(n) \in o(\frac{\log\log n}{\Delta})$, then for every large enough n, $n^{c-1} > |\mathcal{B}_n|$.

Proof: Consider the expressions of n^{c-1} and $|\mathcal{B}_n|$ after applying the logarithm twice:

$$\begin{aligned} \log\log n^{c-1} &= \log((c-1)\log n) \\ &= \log(c-1) + \log\log n \\ &\sim \log\log n \end{aligned}$$

$$\begin{aligned}\log\log|\mathcal{B}_n| &\leq \log\log\left[\left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}\right]\\ &\leq \log\left[f(n)2^{(\Delta+1)f(n)}\Delta\right]\\ &\leq (\Delta+1)f(n)+\log f(n)+\log\Delta\\ &\in o(\log\log n)\end{aligned}$$

As the dominating term in the second expression is of order $f(n) \in o(\log\log n)$, asymptotically the first expression is larger. As $\log\log(\cdot)$ is an increasing positive function for large values, this implies that asymptotically $n^{c-1} > |\mathcal{B}_n|$. ■

Recall that our goal is to find n different identifiers that all correspond to the same behavior function. The next lemma shows that this is feasible as soon as $n^{c-1} = o(|\mathcal{B}_n|)$.

Lemma 3.3

If $n^{c-1} > |\mathcal{B}_n|$ then there exist n distinct identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ in $[1, n^c]$ that all have the same behavior, i.e. such that $\mathcal{A}_{\mathtt{ID}_1} = \mathcal{A}_{\mathtt{ID}_2} = \cdots = \mathcal{A}_{\mathtt{ID}_n}$

Proof: Let $\varphi : [n^c] \to \mathcal{B}_n$ be the function that associates to each identifier its corresponding behavior function: $\varphi(\mathtt{ID}) = \mathcal{A}_{\mathtt{ID}}$. The number of pre-images a behavior function $\mathcal{A}_i$ has is $|\varphi^{-1}(\mathcal{A}_i)|$.

Remark that since φ is a function defined on a set of n^c elements, we have:

$$\sum_{\mathcal{A}_i \in \mathcal{B}_n} |\varphi^{-1}(\mathcal{A}_i)| = n^c.$$

Let us count the average number of pre-image a behavior function has by φ, which is the total number of pre-images divided by the total number of behavior functions. Therefore, this number is:

$$\frac{1}{|\mathcal{B}_n|}\sum_{\mathcal{A}_i \in \mathcal{B}_n} |\varphi^{-1}(\mathcal{A}_i)| = \frac{1}{|\mathcal{B}_n|}\cdot n^c$$

By hypothesis, this average number is greater than n, and thus by pigeonhole principle there must exist at least one behavior function $\mathcal{A}_i$ that has at least n pre-image by φ.

In other words, there exist n distinct identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ such that $\mathcal{A}_{\mathtt{ID}_1} = \mathcal{A}_{\mathtt{ID}_2} = \cdots = \mathcal{A}_{\mathtt{ID}_n}$. ■

Let us now consider a large enough graph $G_i \in \mathcal{G}$. By Lemmas 3.2 and 3.3, there are n identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ such that $\mathcal{A}_{\mathtt{ID}_1} = \mathcal{A}_{\mathtt{ID}_2} = \cdots = \mathcal{A}_{\mathtt{ID}_n}$. Consequently $G_i^0[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ and $G_i^n[\mathtt{ID}_1]$ are indistinguishable by $\mathcal{A}$. This completes the proof of Theorem 3.2.

3.4.3 Generalization to $\mathcal{S}_{\mathsf{PK}}$ and to semi-uniform networks

Generalization to $\mathcal{S}_{\mathsf{PK}}$ In the previous section, we supposed the model $\mathcal{S}_{\mathsf{PU}}$. If we now consider the model $\mathcal{S}_{\mathsf{PK}}$, then nodes know which port was assigned to them by each of their neighbors. Namely, each node v has, as a local variable, a table that associates to each of its Δ_v neighbors which port it was assigned. This table, with length Δ_v and entries between 1 and Δ should be part of the inputs of the function that models $\mathcal{A}$.

There are two options to deal with this situation. The first one is to slightly modify the hypothesis that $\mathcal{G}$ is Δ-limited by $\log n$. If we suppose that $\mathcal{G}$ is Δ-limited by $(\log n)\cdot(\log\log n)$ then adding a field $\Delta\dot{\log}\Delta$ in the parameters of $\mathcal{A}$ does not invalidate all the math we made, especially the proof of Lemma 3.2.

The other option is to remark that the port number assignment can be chosen symmetrically. In such circumstances, all nodes have the same table as an input, and it then becomes irrelevant as a parameter of the function that models $\mathcal{A}$.

Generalization to semi-uniform networks In the previous section, we supposed uniform networks. If we suppose now that the network is semi-uniform, then variable `distinguish` may be used by the algorithm, and thus should be a parameter of the function that models $\mathcal{A}$.

But actually, to prove indistinguishability with anonymous, semi-uniform, networks, it is sufficient to find $n-1$ distinct identifiers that all correspond to the same behavior function and to map them to the $n-1$ undistinguished nodes. Indeed, an anonymous, semi-uniform, network permits that the distinguished node has a behavior different from the other nodes. Thus, we can simply reason on the $n-1$ nodes with variable `distinguish` set to 0, and exactly as before, variable `distinguish` becomes irrelevant as a parameter of the function that models $\mathcal{A}$.

3.5 Equivalence with Homonymous Networks

3.5.1 Statement of the Theorem

In this section, we prove the following theorem:

Theorem 3.3

Let $c \in \mathbb{R}, c > 1$ and let $k \in \mathbb{N}, k \geq 2$. Let us consider $\mathcal{G} = (G_i)_{i\in\mathbb{N}}$ a class of graphs which contains graphs of arbitrary large size, such that $\mathcal{G}$ is Δ-limited by $\log n$.

Let $\mathcal{A}$ be a distributed algorithm that uses registers of size $o(\frac{\log\log n}{\Delta})$ on all identified networks $G_i^0[1, n \times c]$, i.e. on all graphs G_i with unique identifiers in $[1, n \times c]$.

For any large-enough graph $G_i \in \mathcal{G}$, there exist n identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ in $[1, n\times c]$ such that $G_i^0[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n]$ and $G_i^k[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_{n/k}]$ are indistinguishable for $\mathcal{A}$.

In other words, there exist n identifiers in $[1, n\times c]$ such that in G_i, $\mathcal{A}$ cannot distinguish those identifiers: it runs exactly as if the network was homonymous.

3.5.2 Proof when k divides n

Similarly to how we proceeded in Section 3.4, we suppose in this section the $\mathcal{S}_{\mathsf{PU}}$ model and uniform graphs. The remarks made in Section 3.4.3 apply without any difficulty to the following proof to establish the general version of Theorem 3.3.

In this section we prove a weaker version of Theorem 3.3: we suppose that the size of the networks we consider are divisible by the integer k. We show in Section 3.5.3 how this proof can be generalized to all networks.

The proof of Theorem 3.3 is basically the same as the proof of Theorem 3.2, and follows the same scheme. The major difference is that since the range in which the identifiers are taken is smaller, it is harder to find distinct identifiers that cannot be distinguished by $\mathcal{A}$. Actually, we cannot guarantee to find n such identifiers anymore. Fortunately, we can find n distinct identifiers that can be grouped k by k so that in each group all the identifiers have the same corresponding behavior function, and thus we obtain the equivalence with homonymous networks.

Consider a graph G_i with maximal degree Δ, and an algorithm for identified networks using $f(n)$ bits of memory per node on graphs of $\mathcal{G}$. Once again, the algorithm can be seen as the function which takes as an input, an identifier, a state for the node, the degree of

the node, and a state for all of the neighbors of the node:

$$\begin{array}{rccccccccc} \mathcal{A}: & [n\times c] & \times & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

We can still consider that for every identifier i, we have an algorithm of the form:

$$\begin{array}{rccccccccc} \mathcal{A}_i: & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (\text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

Lemma 3.1 remains valid:

$$|\mathcal{B}_n| \leq \left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}$$

Since there are fewer identifiers than there were in the previous, we must be tighter when dominating $|\mathcal{B}_n|$ by a function of n than we were in Lemma 3.2.

Lemma 3.4

If $f(n) \in o(\frac{\log\log n}{\Delta})$, then for every n large enough, we have $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$.

Proof: Consider the expressions of $\frac{n(c-1)}{k-1}$ and $|\mathcal{B}_n|$ after applying the logarithm twice:

$$\begin{aligned} \log\log \tfrac{n(c-1)}{k-1} &= \log(\log n + \log(c-1) - \log(k-1)) \\ &\sim \log\log n \end{aligned}$$

And similarly to what was established for Lemma 3.2:

$$\log\log|\mathcal{B}_n| \in o(\log\log n)$$

As $\log\log(\cdot)$ is an increasing positive function for large values, this completes the proof of the lemma. ■

Recall that our goal is to find n different identifiers that can be grouped by sets of size k such that, in every group, all the identifiers have the same corresponding behavior. If we consider φ the function that associates each identifier to the corresponding behavior, then it boils down to finding n distinct identifiers that can be grouped by k, such that in each group, all the identifiers have the same image by φ. For example, if $k = 2$, if the identifiers are taken in $\{1, 2, 3, 4, 5, 6, 7\}$, and if there are three behaviors $\mathcal{B} = \{b_1, b_2, b_3\}$, we can have $\varphi(1) = \varphi(2) = \varphi(3) = b_1$, $\varphi(4) = b_2$, and $\varphi(5) = \varphi(6) = \varphi(7) = b_3$. In that case, we can form two sets of size $k = 2$ such that all the elements of one set have the same image by φ. We can for example consider $\{1, 3\}$ and $\{6, 7\}$.

Let us give a definition that formalizes that.

Definition 3.3 (*k-group number of a function*)

Let $\varphi : A \to B$ be a function, and let $k \in \mathbb{N}$. We define the k-group number of φ, and denote $t_k(\varphi)$ the maximum number of disjoint sets of size k of elements of A, $S_1, S_2, \ldots, S_{t_k(\varphi)}$ such that all the elements of the same set S_i have the same image by φ.

The precise value of $t_k(\varphi)$ depends on the specificities of φ. Nevertheless, we can have a pretty good estimation of $t_k(\varphi)$ by comparing the respective sizes of A and B. Indeed, the larger A is, the more chances we have to find sets of elements of A that satisfy some property. On the contrary, the larger B is, the more possible images there are, and the harder it is to find elements of A that are mapped to the same element of B.

Lemma 3.5 establishes a generic lower bound on $t_k(\varphi)$.

Lemma 3.5

Let $\varphi : A \to B$ be a function, and let $k \in \mathbb{N}^*$. We have $t_k(\varphi) \geq \frac{|A|-(k-1)|B|}{k}$.

Proof: Let us first consider $b \in B$, and denote $t_k^b(\varphi)$ the number of disjoint sets of size k of elements of A with image b that can be formed. Intuitively, we have $t_k^b(\varphi) = \lfloor \frac{|\varphi^{-1}(b)|}{k} \rfloor$. By definition, we also have

$$t_k(\varphi) = \sum_{b \in B} t_k^b(\varphi) = \sum_{b \in B} \lfloor \frac{|\varphi^{-1}(b)|}{k} \rfloor.$$

Recall the inequality, true for any integer m: $\lfloor \frac{m}{k} \rfloor \geq \frac{m-(k-1)}{k}$. Thus, we deduce:

$$t_k(\varphi) \geq \sum_{b \in B} \frac{1}{k}(|\varphi^{-1}(b)|-(k-1)) \geq \frac{1}{k} \sum_{b \in B}(|\varphi^{-1}(b)|-(k-1)) \geq \frac{1}{k}\big(\sum_{b \in B}(|\varphi^{-1}(b)|)-(k-1)|B|\big)$$

Since φ is a function from A to B we have $\sum_{b \in B}(|\varphi^{-1}(b)|) = |A|$ and thus we conclude $t_k(\varphi) \geq \frac{|A|-(k-1)|B|}{k}$. ■

The next lemma shows that, if $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$, we can find n/k disjoint sets of k identifiers taken in $[1, n \times c]$ such that all the identifiers of each set have the same corresponding behavior.

Lemma 3.6

Let $\varphi_n : [1, n \times c] \to \mathcal{B}_n$ be the function that associates each identifier `ID` *to its corresponding behavior $\mathcal{A}_{\texttt{ID}}$. If $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$, then $t_k(\varphi_n) \geq n/k$.*

Proof: According to Lemma 3.5 we have $t_k(\varphi_n) \geq \frac{1}{k}(nc - (k-1)|\mathcal{B}_n|)$, and by hypothesis this means that $t_k(\varphi_n) \geq \frac{1}{k}(nc - n(c-1))$, and thus $t_k(\varphi_n) \geq n/k$. ■

Let us now consider a large enough graph $G_i \in \mathcal{G}$. By Lemmas 3.4 and 3.6 we can consider $S_1, S_2, \ldots, S_{n/k}$, disjoint sets of k identifiers, such that all the identifiers of the same set have the same image by φ_n. For each set S_i, let us pick one specific identifier, $\texttt{ID}_i$.

Consider now $G_i^k[\texttt{ID}_1, \ldots, \texttt{ID}_{n/k}]$ a k-homonymous network with topology G and k occurrences of each $\texttt{ID}_i$, on the first hand, and $G_i^0[S_1 \cup S_2 \cup \cdots \cup S_k]$ an identified network similar to G_h, where for each i, occurrences of identifier $\texttt{ID}_i$ are substituted by one of each value of S_i. By construction, algorithm $\mathcal{A}$ cannot distinguish identifiers from the same set S_i, and thus $G_i^k[\texttt{ID}_1, \ldots, \texttt{ID}_{n/k}]$ and $G_i^0[S_1 \cup S_2 \cup \cdots \cup S_k]$ are indistinguishable for $\mathcal{A}$. This completes the proof of Theorem 3.3.

3.5.3 Proof for any n

In the previous section, we supposed that n was a multiple of k, which simplifies the distribution of groups of k identifiers to the n nodes of the network.

If this hypothesis falls, then for a fixed value k we can still find n/k disjoint sets of k identifiers such that in each set, all the identifiers have the same corresponding behavior. Yet this does not cover all the nodes anymore, there are still $n \mod k$ nodes that do not have an identifier this way.

Finding one new set of k identifiers that all correspond to the same behavior, and associating some of them to the nodes that do not have an identifier will not work, since a

k-homonymous network must have *at least* k nodes with the same identifier, and $n \mod k$ is less than k.

To solve that issue, let us rather prove that there are at least $n \mod k$ (in practice, at least $k-1$) disjoint sets of $k+1$ identifiers such that in each set, all the identifiers have the same corresponding function. This way, we can construct a k-homonymous network, in which all identifiers are shared by at least k, sometimes $k+1$, nodes.

One problem may arise: it might be that some of those sets of $k+1$ identifiers intersects 2 of the sets of size k that we have. Indeed, suppose that $2k$ identifiers correspond to the same behavior. This allows us to define 2 sets of size k, but if $k+1$ such identifiers are taken by one of the new sets, then we cannot finish the identifier assignments. Hopefully, we cannot lose more that 2 sets of size k due to a set of size $k+1$. If this happens, we can reorganize the different incomplete sets to form new ones.

To avoid such situation, it is sufficient to find not n/k, but $n/k+k$ sets of size k. Thus, even if we lose 2 sets of size k for every set of size $k+1$, we end with k sets of size $k+1$, and $(n/k-k)$ sets of size k, which is sufficient to assign a unique identifier to each node of the graph. Since n is supposed large enough, this additional k will be negligible.

Similarly to Lemma 3.4, we can prove that for any large enough n, we have $|\mathcal{B}_n| \leq \frac{nc-k^2+1}{k}$, on the first hand, and $|\mathcal{B}_n| \leq \frac{n(c-1)-k^2}{k-1}$ on the other hand.

Now, similarly to Lemma 3.6, we use these inequalities to prove that for any large enough n, $t_{k+1}(\varphi_n) \geq k-1$, on the first hand, and $t_k(\varphi_n) \geq \frac{n}{k}+k$, on the other hand.

Finally, we can consider $S_1, S_2, \ldots, S_{n \mod k}$ disjoint sets of $k+1$ identifiers such that all the identifiers of the same set have the same image by φ_n, and $\tilde{S}_1, \tilde{S}_2, \ldots, \tilde{S}_{\lfloor n/k \rfloor - (n \mod k)}$ disjoint sets, disjoint from the S_i, such that all the identifiers of the same set have the same image by φ_n.

The elements of these $\lfloor n/k \rfloor$ jointly contain $(n \mod k)(k+1)+k(\lfloor n/k \rfloor-(n \mod k)) = n \mod k + k\lfloor n/k \rfloor = n$ elements. Thus, similarly to how we concluded the proof of Theorem 3.3, we can consider a large enough graph $G \in \mathcal{G}$ on the one hand, and one specific identifier $\texttt{ID}_i$ for each set on the other hand. By construction, $G^k[\texttt{ID}_1, \ldots, \texttt{ID}_{\lfloor n/k \rfloor}]$ is k-homonymous, and is indistinguishable from the identified network $G^0[S_1 \cup \cdots \cup S_{n \mod k} \cup \tilde{S}_1 \cup \cdots \cup \tilde{S}_{\lfloor n/k \rfloor - (n \mod k)}]$, which completes the proof.

3.6 Lower Bound for $\mathcal{LE}$

3.6.1 Statement of the Theorem

In this section we establish a lower bound of $\Omega(\log\log n)$ bits per node for the space complexity of distributed algorithms that solve leader election in composite, uniform, rings.

Theorem 3.1

Let $c \in \mathbb{R}, c > 1$. Every deterministic distributed algorithm solving leader election in the state model under a central strongly fair scheduler or under the synchronous scheduler requires registers of size at least $\Omega(\log\log n)$ bits per node in n-node composite uniform rings with unique identifiers in $[1, n \times c]$.

This bound improves the only lower bound known so far [BGJ99], from $\omega(1)$ to $\Omega(\log\log n)$, and it is tight, as it matches the upper bound of [BT18], obtained in the same model and under a more challenging scheduler, the weakly fair distributed scheduler. In particular, it invalidates the folklore conjecture stating that the aforementioned problems are solvable using only $O(\log^* n)$ memory.

Link with Dijkstra's impossibility result In a celebrated paper [Dij82], Dijkstra established, among other things, that one cannot break symmetry within anonymous composite rings. This result holds under a central strongly fair scheduler. Theorem 3.1 follows the same idea as Dijkstra's. The core of the proof, and the statement of Theorem 3.2 is about proving that an algorithm with too little memory cannot fully use the power of identifiers. The second part of the proof of Theorem 3.1 is basically a generalization of [Dij82]: we prove that one cannot break symmetry in homonymous composite rings.

Recall that assuming that the ring is composite is essential, due to the constant memory algorithm on anonymous prime rings of [ILS95].

3.6.2 Limits of Theorems 3.2 and 3.3

It is known that $\mathcal{LE}$ cannot be solved in general, homonymous rings [DGFTT14, ADD$^+$20]. Therefore, the proof of Theorem 3.1 will rely on Theorem 3.3 and on its proof. Yet, there are two issues that must be overcome to adapt Theorem 3.3 to the impossibility to solve $\mathcal{LE}$ on composite rings.

3.6.2.1 k-Homonymy

In the previous section, we supposed that k was a fixed integer, independent of n. In this section, in order to prove the impossibility of solving $\mathcal{LE}$, we must make sure to have a symmetric network. This is feasible only if k is a divisor of n. In the worst case, we have $k = \sqrt{n}$. We will have to adapt some of our proofs so that everything remain valid.

3.6.2.2 Indistinguishability

We proved in Section 3.5 that for any algorithm $\mathcal{A}$ which uses few memory, there exist identified and homonymous networks that are indistinguishable for $\mathcal{A}$. It is very tempting to conclude that any problem that cannot be solved in homonymous networks, can neither be solved with few memory. Since it is known that $\mathcal{LE}$ cannot be solved in homonymous networks, we would conclude immediately. Unfortunately, things are not that straightforward.

Indeed, we only proved that networks were indistinguishable for $\mathcal{A}$. We did not prove that they were indistinguishable for $SP_{\mathcal{LE}}$. It could be that two identical executions, one in an identical network, and one in a homonymous network, do not have the same status from the specification point of view, and that the execution in the identified network solves $\mathcal{LE}$, while it does not in the homonym network.

Let us give one example. Suppose that each node has in its memory a string of bits, and that this string of bits corresponds to the identifier of one of the node. In a homonymous network, this does not help to solve $\mathcal{LE}$. However, in the identified network, $\mathcal{LE}$ is now trivially solved, although all nodes are in the exact same state. This is because the specification of the $\mathcal{LE}$, and especially $E_{\mathcal{LE}}$, can rely on the identifiers.

The example above requires that each node stores an identifier, which requires at least $\log n$ bits, and thus, it is contradictory with the hypothesis on the memory. This is actually not a coincidence: with few memory, one predicate, not more than one algorithm, can fully use the uniqueness of the identifiers.

We will follow that idea to prove that the reasoning we made on $\mathcal{A}$ in Sections 3.4 and 3.5 can be extended to the predicate that determines $\mathcal{LE}$.

3.6.3 Proof

In this section we prove Theorem 3.1 by focusing on the case were the scheduler is a central strongly fair scheduler. Although the synchronous scheduler cannot be said stronger than the central strongly fair scheduler, the construction we propose fits for the synchronous scheduler too.

Consider a ring of size n, and an algorithm for identified networks $\mathcal{A}$ using $f(n)$ bits of memory per node to solve leader election in composite rings.

Similarly to what we did in the previous sections, we can consider the function that describes the behavior of the algorithm. Since all nodes have exactly two neighbors, the expression of $\mathcal{A}$ as a function is much simpler than in the previous section:

$$\begin{array}{rccccccccc} \mathcal{A}: & [n\times c] & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \to & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & \text{new-state} \end{array}$$

Note that in general, we consider non-directed rings thus the nodes do not have a consistent global definition for right and left. As we are dealing with a lower bound with a worst-case on the port numbering, assuming such a consistent orientation only makes the result stronger.

Unfortunately, as explained above, this will not be sufficient to establish Theorem 3.1.

Let us suppose that $\mathcal{A}$ solves $\mathcal{LE}$. By definition there exists a boolean predicate $E_{\mathcal{LE}}$ associated to $\mathcal{A}$, that characterizes how $\mathcal{A}$ solves the specification of $\mathcal{LE}$. By definition, this predicate takes the same input as $\mathcal{A}$: the local variables of the node on which it is evaluated, and the state of the node itself and of its neighbors. Therefore, we can consider the function $(\mathcal{A}, E_{\mathcal{LE}})$, which takes the same inputs as $\mathcal{A}$, or $E_{\mathcal{LE}}$, and returns a tuple, the result computed by $\mathcal{A}$, which is a new state for the node, and the result computed by $E_{\mathcal{LE}}$, which is a boolean value that indicates whether the node is elected or not. Formally:

$$\begin{array}{rccccccccc} (\mathcal{A}, E_{\mathcal{LE}}): & [n\times c] & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \to & \{0,1\}^{f(n)}\times\{0,1\} \\ & (ID & , & \text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & (\text{new-state}, \text{elected}) \end{array}$$

Now, we can consider that for every identifier i, we have a tuple of the form:

$$\begin{array}{rccccccc} (\mathcal{A}, E_{\mathcal{LE}})_i: & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \to & \{0,1\}^{f(n)}\times\{0,1\} \\ & (\text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & (\text{new-state}, \text{elected}) \end{array}$$

Thus a specific tuple $(\mathcal{A}, E_{\mathcal{LE}})_i$ boils down to a function of the form: $\{0,1\}^{3f(n)} \to \{0,1\}^{f(n)+1}$. Let us call such a function a *behavior*, and let $\mathcal{B}_n$ be the set of all behaviors.

Similarly to what was done in Lemma 3.1 we can count how many distinct behaviors can exist:

$$|\mathcal{B}_n| = 2^{(f(n)+1)\times 2^{3f(n)}}$$

Remark that from now on, the issue coming from the non-indistinguishability for the predicate is basically solved: behaviors now include the predicate, and the increase in number is negligible in front of what we tolerate for the algorithm.

However, the issue coming from which the possibility for k to grow with n has not been addressed yet. Recall that our goal is to find n different identifiers which can be grouped by sets of size k (k being a divisor of n), such that in every group, all the identifiers have the same corresponding behavior. In the worst case, all rings have size p^2 where p is prime. To build a symmetric configuration in this case, and formally establish impossibility, we need to find p disjoint sets of p identifiers that all have the same corresponding behavior, where p is the square root of the size of the network. Therefore, k cannot be supposed a constant anymore.

Let us establish a variant of Lemma 3.4:

Lemma 3.7

If $f(n) \in o(\log\log n)$, then for every n large enough, for every $k \leq \sqrt{n}$, we have $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$.

Proof: Remark first that $\frac{n(c-1)}{k-1} \geq \sqrt{n}(c-1)$. Now consider the expressions of $\sqrt{n}(c-1)$ and $|\mathcal{B}_n|$ after applying the logarithm twice:

$$\begin{aligned}\log\log(\sqrt{n}(c-1)) &= \log(\tfrac{1}{2}\log n + \log(c-1)) \\ &\sim \log\log n\end{aligned}$$

$$\begin{aligned}\log\log(|\mathcal{B}_n|) &= \log\left((f(n)+1)\times 2^{3f(n)}\right) = \log(f(n)+1) + 3f(n) \\ &\in o(\log\log n)\end{aligned}$$

As $\log\log(\cdot)$ is an increasing positive function for large values, this completes the proof of the lemma. ■

The next lemma shows that if $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$, then for every $k \leq \sqrt{n}$ we can find n/k disjoint sets of k identifiers taken in $[1, n \times c]$ such that all the identifiers of each set have the same corresponding behavior.

Lemma 3.8

Let $\varphi_n : [1, n \times c] \to \mathcal{B}_n$ be the function that associates each identifier i to its corresponding behavior $\mathcal{A}_i$, and let $k \leq \sqrt{n}$. If $\frac{n(c-1)}{k-1} > |\mathcal{B}_n|$, then $t_k(\varphi_n) \geq n/k$.

Proof: According to Lemma 3.5 we have $t_k(\varphi_n) \geq \frac{1}{k}(nc - (k-1)|\mathcal{B}_n|)$, and by hypothesis this means that $t_k(\varphi_n) \geq \frac{1}{k}(nc - n(c-1))$, and thus $t_k(\varphi_n) \geq n/k$. ■

Combining the three lemmas we get that, if $f(n) \in o(\log\log n)$, then for any large enough n, for any $k \leq \sqrt{n}$, we can find n different identifiers in $[1, n \times c]$, which can be grouped in sets of size k such that all identifiers of the same set have the exact same behavior.

Now, consider a large enough composite ring of size n, and let k be a divisor of n such that $1 < k \leq \sqrt{n}$. Let us consider n identifiers which can be grouped as explained above, and let us place those identifiers on the ring such that the identifiers of the same set are placed each n/k node, similarly to what is presented on Figure 3.1.

For our proof, we need to start in a symmetric configuration, where all nodes with their identifier taken in the same set have the same state. Since our theorem does not suppose a faulty environment, we must consider as an initial configuration, a correctly initialized configuration, where all nodes have the same, clean, state. This perfectly fits what is required for the proof.

Let γ_0 be the initial, correctly initialized, configuration of the system. We cannot have $P_{\mathcal{LE}}(\gamma_0)$. Indeed, the predicate $E_{\mathcal{LE}}$ has the same behavior on all the nodes with identifiers taken in the same set. Therefore, the number of nodes v such that $E^{\gamma_0}_{\mathcal{LE}}(v) = 1$ is a multiple of k.

Since γ_0 does not satisfy $P_{\mathcal{LE}}$, at least one node is enabled. Note that all nodes with identifiers from the same set see the same states for themselves, and for both their neighbors. Thus, since all of them have the same behavior, if one is enabled, then the others are too, and if activated they will execute the same rule.

Now, the central scheduler activates one of them, and for the $k-1$ next steps, it activates each of the others. Since those nodes are placed at a distance at least 2 from each other, the action executed by one node has no incidence on the state of the other nodes, nor on the

states of their neighbors, and thus the other nodes take the exact same step as the first one. Thus, after those k computing steps, the ring is once again in a symmetric configuration γ_k. One again, we have $\neg P_{\mathcal{LE}}(\gamma_k)$.

We can iterate this argument forever, as long as the scheduler consecutively activates nodes from the same set, which a central strongly fair scheduler can do. In other words, there exists an execution of $\mathcal{A}$ such that the system is infinitely often in symmetric configurations, which do not satisfy $P_{\mathcal{LE}}$. Therefore, the network never stabilizes in an execution that satisfies leader election. This proves Theorem 3.1.

If we now consider an execution under the synchronous scheduler, where all the enabled nodes are atomically activated at each computing step, then all the k computing steps happen at once, and therefore all the configurations are symmetric, which also invalidates the specification of $\mathcal{LE}$.

Generalization of Theorem 3.3 Theorem 3.3 establishes an equivalence between algorithms with small memory, and algorithms for k-homonymous networks, for any fixed value of $k \in \mathbb{N}$. On the other hand, the technique used in the proof of Theorem 3.1 guarantees that for any $k \leq \sqrt{n}$, we can build a k-homonymous network on which the algorithm behaves exactly the same way as in an identified network.

The second is more general, since k can depend on n. Since the proofs of Theorem 3.1 and 3.3 are basically the same, we can adapt the given proof of Theorem 3.3 to make the result a bit more general.

We state the new theorem as a corollary of Theorem 3.3 and of the proof of Theorem 3.1.

Corollary 3.1

Let $c \in \mathbb{R}, c > 1$. Let us consider $\mathcal{G} = (G_i)_{i \in \mathbb{N}}$ a class of graphs which contains graphs of arbitrary large size, such that $\mathcal{G}$ is Δ-limited by $\log n$.

Let $\mathcal{A}$ be a distributed algorithm that uses registers of size $o(\frac{\log \log n}{\Delta})$ on all identified networks $G_i^0[1, n \times c]$, i.e. on all graphs G_i with unique identifiers in $[1, n \times c]$.

For any large-enough graph $G_i \in \mathcal{G}$, for any divisor k of n such that $k \leq \sqrt{n}$, there exist n identifiers $\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n$ in $[1, n \times c]$ such that $G_i^0[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_n]$ and $G_i^k[\mathtt{ID}_1, \mathtt{ID}_2, \ldots, \mathtt{ID}_{n/k}]$ are indistinguishable for $\mathcal{A}$.

3.7 Conclusion

In this chapter, we establish two generic results that establish a strong link between algorithms that use $o(\log \log n)$ bits per node, and algorithms executed in anonymous or homonymous networks. Since various problems are unsolvable in anonymous, or homonymous networks, these theorems are a powerful tool to establish sub-logarithmic lower bounds.

However, the specification of the problem must be taken into account too, to conclude an impossibility result from an indistinguishability result. We showed in Section 3.6 how this logical link can be done, by addressing the leader election problem. Specifically, we prove a $\Omega(\log \log n)$ bits per node lower bound for the leader election problem on the ring.

This bound matches the upper-bound $O(\log \log n)$ bits per node on rings [BT18]. Yet, for arbitrary graphs, [BT20] requires an additional space in $O(\log \Delta)$ bits per node. An interesting problem would be to find whether this additional term in $O(\log \Delta)$ is necessary for graphs where the degree is not bounded.

Chapter 4

Silent Anonymous Snap-Stabilizing Termination Detection

You are terminated

Terminator

Contents

A preliminary version of these works was presented in [BJBP22b].

4.1 Introduction

Termination Detection [Fra80] is a fundamental and widely studied problem of distributed systems. It belongs to the category of global system observation mechanisms that processes of the distributed system may need in the accomplishment of a global computation, for example, detecting the presence of a deadlock, taking a snapshot of the global system state, or maintaining a logical distributed clock. The distributed nature of systems makes these problems difficult to solve. They are subject to specific distributed algorithms dedicated to control the global state of other distributed algorithms. Regarding the termination detection (TD, for short) problem, any node of the distributed system may need to detect whether a computation has globally terminated. More precisely, upon a (local) request (for instance, from an application) a node initiates an instance of TD over the whole system to find out whether another distributed algorithm is terminated.

4.1.1 Motivation

It is known that TD cannot be achieved in the context of self-stabilization, except in the specific case where the TD algorithm is snap-stabilizing. Indeed, with the self- but not snap-stabilizing approach, assume an application where nodes decide to use local variables (supposed to be updated by a global detection mechanism) whether the stabilization phase is over or not. Then it is always possible to build an arbitrary configuration in which the same nodes have access to exactly the same local information. As a consequence, the two situations are indistinguishable, leading to a wrong decision in the second one.

Nevertheless, a self-stabilizing algorithm (at least) guarantees that by repeating global detection instances, the variables provide the correct answer at some, but in an unpredictable time. Indeed, let ssTD be a self-stabilizing TD algorithm. By initiating ssTD, it is possible that ssTD returns a wrong answer during the stabilization phase, *e.g.*, ssTD returns "*yes*" (meaning that an observed algorithm $\mathcal{A}$ terminated), while $\mathcal{A}$ actually did not terminate. In other words, ssTD can compute incorrect (or unsafe) answers several (but a finite number of) times before at the end computing true/correct answers. Self-stabilization ensures that by repeating instances of ssTD, the answer is eventually correct forever.

In [CDD+16], it is shown that the termination detection can be achieved using only one detection instance *i.e.*, at the very first request to know whether an observed algorithm is terminated or not, the returned answer is correct, even if the request was initiated during the stabilization phase. This corresponds to the paradigm of snap-stabilization.

It is important to notice that snap-stabilizing algorithms do not hide better the effects of transient faults than the self-stabilizing algorithms that do not respect this property. However, while a self-stabilizing algorithm guarantees only a finite, yet generally unbounded, number of incorrect answers after the faults cease, a snap-stabilizing algorithm offers correct answers from the first request (after the faults cease).

Note that the snap-stabilizing solutions in [CDD+16] assume an identified network, where each node has a unique identifier. Even if most of existing distributed systems are identified, developing algorithms that do not use process identifiers definitely makes sense in several aspects. Such algorithms are said to be *anonymous* algorithms—or, algorithms for *anonymous* systems.

Anonymity often makes the resolution of some problems harder by far. That makes anonymous approaches interesting from a computational point of view. As presented in the previous chapter, Leader Election is an iconic example of that difficulty. Although the problem is quite trivial to solve deterministically by using the total order provided by process identifiers—just choose the maximum or minimum identifier as the leader, it

cannot be solved at all in systems lacking properties (as process identifiers) that break possible symmetries [Ang80, Dij82].

Also, anonymous solutions *a priori* require less memory. Notably, they do not need process identifiers, and thus do not require any register to store the information related to the identifiers. Anonymous approaches are also very attractive from a practical point of view. Indeed, they provide solutions that preserve user privacy, they work for systems with homonyms, where nodes can change their names or be replaced during the algorithm lifetime. They are also very suitable for networks made of units with weak capabilities such as wireless sensor networks, body-area networks, *etc.*

4.1.2 Related Work

The question of detecting the stability of a self-stabilizing algorithm in an anonymous system was first addressed in [LS92]. In this paper, the authors introduce the notion of *observer*: a local node that can detect correctness of a given algorithm, but cannot influence it.

Assuming the central scheduler, where only one node takes a step at each time and a prime-size uniform ring, the authors propose a deterministic distributed algorithm for the observer that detects stability in $\Theta(n^2)$ steps from the time the ring is stabilized. Located at each node, the proposed observer is not subject to any type of corruption *i.e.*, it is not self-stabilizing. In [BPR05], the authors also propose a non-self-stabilizing observer. They propose an observer for synchronous rooted systems, where all enabled nodes take steps simultaneously and a unique node is distinguished from the others. In a synchronous and non-self-stabilizing system, the same authors remove the constraint of having a distinguished node by introducing randomization [BPR06].

The first deterministic algorithm that solves the problem addressed in [LS92] that is also self-stabilizing is proposed in [CDD+16]. By contrast with the above results [LS92, BPR05, BPR06] that are not self-stabilizing, the results in [CDD+16] show the necessity to achieve snap-stabilization and not only self-stabilization. Indeed, only snap-stabilization offers the desirable property of returning the right answer to the request of knowing whether a self-stabilizing algorithm achieved stability, even during the stabilization phase of the observed algorithm. As mentioned earlier, the solution in [CDD+16] requires a named network.

In anonymous networks of arbitrary size, *unison* [CFG92, BPV04, DJ19, EK21] offers a nice support to implement deterministic solutions [BLP08]. The asynchronous unison consists in maintaining a local *logical clock* (sometimes referred to as *counter*), one for each node, such that: (i) the clock value of each node does not differ by more than 1 with any of its neighbors, and (ii) the clock value of each node is increased by 1 infinitely often. The unison principle is a strong tool to synchronize the whole system by implementing a synchronization barrier. To the best of our knowledge, this principle forms the basis of all known deterministic solutions for anonymous networks, even non-self-stabilizing, which solve global (*a.k.a.*, *total* [Tel88]) problems, *i.e.* problems involving all nodes of the network before a decision can be taken. TD obviously belongs to this class of algorithms. The phase algorithm in [Tel88] and the TD algorithm in [SSP85] are typical examples of algorithms that use an underlying unison mechanism. Both algorithms require that nodes know (an upper bound on) the network diameter D, *i.e.* the maximum distance between two nodes of the network. As far as we know, the question of the necessity of this knowledge to be able to deterministically solve total problems remains open.

In [God19], the author proposes a snap-stabilizing TD algorithm to characterize tasks that are solvable with snap-stabilizing algorithms in anonymous networks. This algorithm combines the synchronization technique in [SSP85] and the self-stabilizing enumeration algorithm in [God02]. The former actually uses a unison mechanism and requires that

nodes know (an upper bound of) D. The latter works on a particular class of graphs (so called, non-ambiguous graphs [Maz97]) and implements a renaming mechanism that uses an *a priori* exponential memory size.

Many self-stabilizing algorithms aim at being *silent*, *i.e.*, after some calculations, the communication registers used by the algorithm remain fixed as long as no request is made [DGS99, GH99]. By communication variable, we mean variables shared between neighboring nodes. Silence is trivially a desired property with algorithms that terminate by building distributed fixed structures—*e.g.*, spanning trees, coloring, Maximal Independent Set, *etc.* It is also very desirable for so-called "long-lived" algorithms—*e.g.*, mutual exclusion, unison, routing, *etc.*—to reduce communication operations and bandwidth. For instance, it is quite easy to design a self-stabilizing silent unison [GH99, BPV04] by modifying the above Condition (ii) as follows: (ii') the clock value of each node is increased by 1, provided that at least one node decides to do it.

4.1.3 Contribution

In this chapter, we address asynchronous anonymous networks. We focus on *terminating and silent* self-stabilizing algorithms *i.e.*, self-stabilizing algorithms that converge in finite time to a desired global configuration from which no values of the communication variables are changed thereafter. We present a generic, deterministic, silent algorithm that detects whether an observed terminating silent self-stabilizing algorithm has converged to a configuration that satisfies an intended predicate. Our solution uses similar techniques as in [BLP08] to achieve snap-stabilization, namely it is based on an underlying unison algorithm. However, in this chapter, the latter can be any (asynchronous) unison in the literature, *e.g.*, [CFG92, BPV04, DJ19, EK21].

As all existing deterministic anonymous total algorithms in the literature (*e.g.*, [SSP85, Tel88, BLP08, God19]), our algorithm requires that nodes know (an upper bound on) the diameter D of the network. It works under the weakest scheduling assumptions *a.k.a*, the unfair scheduler. Built over any asynchronous self-stabilizing underlying unison $\mathcal{U}$, our solution adds only $O(\log D)$ bits per node, where D is the diameter of the network. Since there exists no unison algorithm with better space complexity — the best space complexity for the asynchronous unison is obtained in [EK21] with $O(\log D)$ bits —, the extra space of our solution is negligible *w.r.t.* the space complexity of the underlying unison algorithm.

Time complexities are given in terms of *rounds* that captures the execution rate of the slowest node in any computation [CDD+16, Dol00]. The response time computes the number of rounds between the time when a request is triggered and the time when the answer to that request is returned. The response time of our solution is in $O(\max(k, k', D))$ rounds, where k and k' are the stabilization time complexities of $\mathcal{A}$ and $\mathcal{U}$, respectively. In other words, once both $\mathcal{A}$ and $\mathcal{U}$ are stabilized, our solution provides an answer in optimal time, *i.e.*, $O(D)$ rounds.

Chapter Outline The remainder of the chapter is organized as follows. We first formally describe specific notations and definitions. Then, in Section 4.3, we formally define the unison problem, and establish some properties of self-stabilizing unison algorithms. In Section 4.4, we present and formally describe our snap-stabilizing algorithm for termination detection. Section 4.5 contains the proofs of our claims and theorems, and in particular we establish that our algorithm is snap-stabilizing for the termination detection problem. We make some concluding remarks in Section 4.6.

4.2 Model

4.2.1 Computational Hypothesis

In this chapter, we consider the port-unknown state model $\mathcal{S}_{\mathsf{PU}}$, where nodes do not know the port number they were assigned by their neighbors. We also consider anonymous, uniform networks. The conjugation of these three parameters is the weakest assumption one can make on the network.

Nevertheless, we suppose that nodes all know an upper bound on the diameter of the network D. We do not suppose that all nodes have the exact same lower-bound. However, the time and space complexity of our algorithm depend on the highest value of that lower-bound in the network. To avoid unnecessary complications, we suppose in the following that all nodes have the same value D as an upper bound on the diameter of the network.

Our algorithm for termination detection relies on a self-stabilizing unison algorithm $\mathcal{U}$. This algorithm is used as a black box, we only require that it eventually stabilizes in an execution satisfying the Unison requirements. On the other hand, the termination detection itself must be snap-stabilizing: it must not provide any incorrect output if no fault occurs during its execution. Classically, our algorithm can only detect the termination of a terminating, and thus silent, algorithm. The termination detection algorithm is itself silent too.

Finally, our termination detection algorithm works under any scheduler. Hence, it nevertheless requires that the observed algorithm terminates, which may be possible only under some specific scheduling assumptions.

4.2.2 Unison Algorithms

Algorithms solving Unison guarantee that all the nodes in the system have a variable clock that increases infinitely often, and such that any pair of neighboring nodes have a difference of at most one between their clocks. Let us denote by $\mathcal{C}$ the set of values that clock can take.

Since we consider nodes with finite memory, $\mathcal{C}$ must be finite. Therefore, to be infinitely increased, it is necessary that after some increase, the clock of some node reaches some value it has already taken. Therefore, in a valid execution, the values taken by clock must cycle.

Without loss of generality, we suppose that the values taken by clock contain some set $\mathbb{Z}/m\mathbb{Z}$, the modulo-m integer, for some value of m (which may depend on some parameters of the network).

To achieve maximal genericity, we consider that $\mathcal{C}$ can contain other values than the classical values in $\mathbb{Z}/m\mathbb{Z}$. Typically, $\mathcal{C}$ can contain control values such as $\perp$, nil, etc. Notice that if the value of $\mathsf{clock} \in \mathbb{Z}/m\mathbb{Z}$, then the usual arithmetic operations on clock are modulo$-m$ operations. In the following, we consider the operations made on clock, specific to each algorithm, only when $\mathsf{clock} \in \mathbb{Z}/m\mathbb{Z}$.

Let us formalize this in the following specification:

Specification 4.1 (*Unison*)

$\mathcal{U}$ (Unison) in $G = (V, E)$ is specified by the existence of a variable clock_v on each node $v \in V$ which takes values in $\mathcal{C} \supseteq \mathbb{Z}/m\mathbb{Z}$, for some integer m.

A node v is non-faulty in γ if

$$P^{\gamma}(v) \equiv \begin{cases} \forall u \in N[v], \mathsf{clock}_u^{\gamma} \in \mathbb{Z}/m\mathbb{Z} \\ \wedge \\ \mathsf{clock}_u^{\gamma} \in \{\mathsf{clock}_v^{\gamma} - 1, \mathsf{clock}_v^{\gamma}, \mathsf{clock}_v^{\gamma} + 1\} \end{cases}$$

A configuration γ is correct if all nodes are non-faulty:

$$P_{\mathcal{U}}^{\gamma} \equiv \forall v \in V, P^{\gamma}(v)$$

An execution $\epsilon = \gamma_0 \rightarrow \cdots$ is correct if all the configurations γ_i are correct (safety property) and if all the nodes have their variable clock increased infinitely many times (liveness property):

$$SP_{\mathcal{U}}(\epsilon) \equiv \begin{cases} \forall i \geq 0, P_{\mathcal{U}}^{\gamma_i} \\ \wedge \\ \forall v \in V, \forall i \geq 0, \exists j > i : \mathsf{clock}_v^{\gamma_j} = \mathsf{clock}_v^{\gamma_i} + 1 \end{cases}$$

An algorithm $\mathcal{U}$ is a self-stabilizing algorithm for Unison if $\Gamma \triangleright_{\mathcal{U}} P_{\mathcal{U}}$ and if any execution that starts in $P_{\mathcal{U}}$ satisfies $SP_{\mathcal{U}}$.

4.2.3 Termination Detection Algorithms

Request-Based Algorithm A request-based algorithm RB is an algorithm that interacts with an external application *App*, typically an external user or another algorithm. This interaction takes place through one shared variable, req, that can be updated by the application. The variable req has four values: $\mathsf{idl}, \mathsf{on}, \mathsf{wk}, \mathsf{off}$. The value idl means that no request is in progress on node v for the application. Node v is then said *idle*. Value on means that a request is initiated, but the computation is not launched. Once the requested computation is running, req is set to wk. The fourth value, off indicates that the requested computation is done, but the result has not been communicated to the application yet. Variable req is updated through four methods. Two of them, *ask* and *get*, are part of *App*. The two others, $\mathbb{R}_{start}$ and $\mathbb{R}_{stop}$, are part of RB—see Figure 4.1.

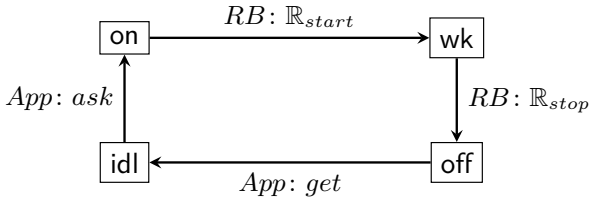

Figure 4.1: Diagram for req_v

Snap-stabilization is a suitable paradigm for request-based algorithms. Indeed, a snap-stabilizing request-based algorithm ensures that if the application executes *ask* on one node v, then the following execution of *get* on the same node v will mark the end of a correct computation. We define the response time of a request-based algorithm as the maximal

number of rounds between any computing step in which the application executes ask on any node v, and the following computing step in which the same node v updates req_v to off.

Simulation Strictly speaking, a termination detection algorithm $\mathcal{T}$ runs on the same network as $\mathcal{A}$ the algorithm it observes. $\mathcal{T}$ has a read-only access to the variables of $\mathcal{A}$, and $\mathcal{A}$ is executed exactly the same way as if it was not observed. To model this behavior in a didactic way, we reverse this paradigm. Rather than having $\mathcal{T}$ observe an independent execution, and to define triggers for each action taken by $\mathcal{A}$, We consider that $\mathcal{T}$ is in charge of the execution of $\mathcal{A}$. Both points of view are equivalent, and only differ in the writing conventions.

Yet, since $\mathcal{T}$ executes the code of $\mathcal{A}$, we must assure that the behavior of $\mathcal{A}$ executed by $\mathcal{T}$ actually corresponds to real executions of $\mathcal{A}$. We introduce the notion of simulation to formalize that constraint:

Definition 4.1 (*Simulation*)

An algorithm $\mathcal{T}$ is said to simulate an algorithm $\mathcal{A}$ if the variables of $\mathcal{A}$ are a subset of the variable of $\mathcal{T}$, and if any (possibly infinite) execution of $\mathcal{T}$, $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ corresponds to a legitimate execution of $\mathcal{A}$, $\epsilon_{\mathcal{A}} = (\gamma_{0|\mathcal{A}} \underset{\mathcal{T}}{\rightarrow} \gamma_{1|\mathcal{A}} \underset{\mathcal{T}}{\rightarrow} \cdots)$ on the subset of the variables of $\mathcal{A}$, with possibly empty computing steps $\gamma_{i|\mathcal{A}} = \gamma_{i+1|\mathcal{A}}$.

ϵ is called a simulation of $\mathcal{A}$.

The possible empty computing steps correspond to the fact that $\mathcal{T}$ might execute actions without any activation of rules of $\mathcal{A}$.

Remark that our algorithm $\mathcal{T}$ will not only simulate $\mathcal{A}$, but the unison algorithm $\mathcal{U}$ as well.

Termination Detection In this chapter, we consider the Snap-Stabilizing Termination Detection problem. An algorithm solves the termination detection problem on $\mathcal{A}$ if it simulates $\mathcal{A}$, and if when a request is emitted on some node u, (req is set from idl to on), then u ultimately answers (*i.e.*, req is set to off), and when it answers, algorithm $\mathcal{A}$ has terminated.

Let us state that more formally:

Specification 4.2 (*Termination Detection*)

$\mathcal{TD}$ (Termination Detection) of a silent distributed algorithm $\mathcal{A}$ in $G = (V, E)$ is specified by the presence of a variable $\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}, \mathsf{off}, \mathsf{idl}\}$ on each node $v \in V$.

An execution $\epsilon = \gamma_0 \rightarrow \cdots$ is correct if:

1. *ϵ is a simulation of $\mathcal{A}$,*
2. *$\mathcal{A}$ terminates in ϵ*
3. *$\forall t \geq 0, \forall v \in V$, if $\mathsf{req}_v^{\gamma_t} = \mathsf{on}$ then*
   - *3-i $\exists t' > t : \mathsf{req}_v = \mathsf{off}$, and*
   - *3-ii $\forall t' > t : \mathsf{req}_v = \mathsf{off}$, $\mathcal{A}$ has terminated in $\gamma_{t'}$.*

Remark that according to Definition 4.1, an execution that does not take any computing step of $\mathcal{A}$ is considered as a simulation of $\mathcal{A}$. This does not fit our requirements, since we

need that $\mathcal{A}$ terminates in any execution of $\mathcal{T}$. For that reason, we add another requirement which forces $\mathcal{A}$ to terminate in executions of $\mathcal{T}$.

4.2.4 Algorithm-Specific Notations

In the following, we will simultaneously consider three algorithms: the observed algorithm $\mathcal{A}$, a unison algorithm $\mathcal{U}$, and a termination detection algorithm $\mathcal{T}$. To avoid confusions, we specify the algorithm to which our notations correspond by the following: The set of rules of algorithm X is denoted by R_X. The subset of the variables of one node v specific to one algorithm X is denoted by $\texttt{state}_{v|X}$.

4.3 Properties of Unison Algorithms

4.3.1 Preliminaries: Silent Unison Algorithms

The termination algorithm that we propose is silent. Yet, it simulates a unison algorithm, whose liveness property is the opposite of silence. Let us explain how to combine those two properties.

Any self-stabilizing unison algorithm can be made *silent*, with as consequence the loss of the liveness property. To do so, we can prevent the clock to increase if there is no request for it and the safety property is achieved. Essentially, if $P(v)$ and $\forall u \in N_v, \mathsf{clock}_u \in \{\mathsf{clock}_v - 1, \mathsf{clock}_v\}$, then v is not enabled. If the initial configuration of the system is correct (*i.e.* if $P_{\mathcal{U}}^{\gamma_0}$), then the system reaches a terminal configuration as soon as all the clock reach the highest initial value.

Yet, one may want to keep a silent self-stabilizing unison algorithm running as long as some external condition is not satisfied. This *request* notion may be extended with the concept of *local request* [BPV04]. Silent self-stabilizing unison algorithms are equipped with one additional predicate $\mathsf{LocReq}(v)$ that depends on external parameters. When this predicate is evaluated to `true` on one node, then it takes precedence over the termination condition, and forces the system to keep running with respect to $P_{\mathcal{U}}$. In the following, we consider a silent self-stabilizing unison algorithm.

4.3.2 Rules of Unison Algorithms

In this section, we aim at proving global properties of unison algorithms. Since those properties must be true for any unison algorithm, the only tool on which we can reason is the specification of $\mathcal{U}$, $SP_{\mathcal{U}}$. The main tool on which we reason is the fact that the predicate $P_{\mathcal{U}}$ is closed. This means that once a correct configuration is reached, the system never becomes incorrect.

But individual nodes do not have any information on the global correctness of the system. Before taking an action or not, the only information on which one node can rely is its local correctness. In other words, the specification of unison guarantees that a node v such that $P(v)$ never executes an action that, isolated, could invalidate the correctness of the whole system. Let us first recall the definition of $P(v)$:

$$P(v) \equiv \begin{cases} \forall u \in N[v], \mathsf{clock}_u \in \mathbb{Z}/m\mathbb{Z} \\ \wedge \\ \mathsf{clock}_u \in \{\mathsf{clock}_v - 1, \mathsf{clock}_v, \mathsf{clock}_v + 1\} \end{cases}$$

Let us consider a unison algorithm $\mathcal{U}$. It will be useful for us to separate the rules of $\mathcal{U}$ into three distinct sets, depending on their guards. Let $R_{\mathcal{U}}$ be the set of the rules of $\mathcal{U}$.

Let us define the following predicates:

$$\begin{aligned} P^+(v) \equiv &\quad P(v) \wedge \forall u \in N_v, \mathsf{clock}_u \neq \mathsf{clock}_v - 1 \\ P^-(v) \equiv &\quad P(v) \wedge \exists u \in N_v : \mathsf{clock}_u = \mathsf{clock}_v - 1. \end{aligned}$$

Note that $P^+(v)$ and $P^-(v)$ are mutually exclusive, and, joined, equate $P(v)$. We suppose without loss of generality that the guard of all the rules in $R_{\mathcal{U}}$ contains $\neg P(v)$, $P^+(v)$, or $P^-(v)$. If not, we make three copies of each rule, and add $\neg P(v)$ to the guard of the first copy, $P^+(v)$ to the guard of the second copy, and $P^-(v)$ to the guard of the third copy.

Since $\mathcal{U}$ is a self-stabilizing algorithm, any rule that includes $P(v)$ in its guard guarantees that its action will not invalidate $P(v)$. Thus, actions that include $P^+(v)$ either increment clock_v by one or do not update it. In the same way, actions that include $P^-(v)$ cannot update clock_v, for it could lead to a difference of 2 between v and its neighbor u such that $\mathsf{clock}_u = \mathsf{clock}_v - 1$.

Among all the rules that include $P^+(v)$, we denote by $R_{\mathcal{U}_N}$, and call the *normal rules* the rules whose action increases by 1 the variable clock_v. These are the rules by which $\mathcal{U}$ makes progress. We denote by $R_{\mathcal{U}_T}$ and call the *transparent rules* all the other rules that include $P(v)$ (either $P^-(v)$ or $P^+(v)$), which do not update clock_v by definition. Transparent rules do not necessarily exist in all unison algorithms, but cannot be avoided *a priori*. Finally, we denote by $R_{\mathcal{U}_C}$ and call the *convergence rules* all the other rules of $R_{\mathcal{U}}$, rules that include $\neg P(v)$, and that enable convergence. By definition, $\mathcal{U}$ has stabilized if and only if no rule of $R_{\mathcal{U}_C}$ is enabled.

We define the *sluggishness* of $\mathcal{U}$, and denote $\mathcal{S}(\mathcal{U})$ as the maximal number of consecutive transparent rules a node can execute between two executions of normal rules, after stabilization. Sluggishness depicts how much the transparent rules might slow down clock increment. Sluggishness of unison algorithms presented in [CFG92, BPV04, BPV08, DJ19, EK21] is 0.

4.3.3 Tools on Executions of Unison Algorithms

In this subsection we introduce logical tools that will be useful to reason on generic unison algorithms. Definitions 4.2 to 4.5 were introduced in [BLP08]. Definition 4.6 is original work and was designed specifically for our proof.

Our goal, in this section, is to prove that following the combination of the safety property and of the liveness property of the unison specification. Our goal, in this section, is to prove that, due to the combination of the safety property and of the liveness property of the unison specification, the activations of nodes during the execution of a unison algorithm reflect, somehow, long-distance interactions between nodes.

To illustrate this idea, assume that one node v, anywhere in the network, is blocked, *i.e.* never increases its clock. Therefore, its neighbors will eventually be blocked at 1 over the value of clock_v. But then, the neighbors of v's neighbors will eventually be blocked at 2 over the value of clock_v. Step by step, we can see that the entire network will eventually be blocked, which means that one node may have an influence in the entire network. If at some point, v restarts increasing its clock, then its neighbors will be able to increase their own variable too. And after that, the neighbors of v's neighbors, and so on.

In order to formalize this notion of interactions, we must provide some definitions.

An event corresponds to a computing step in which one node executed a particular rule.

Definition 4.2 (*Event*)

Let $\epsilon = (\gamma_0 \xrightarrow[u]{} \gamma_1 \cdots)$ be a finite or infinite execution.

An event is a pair $(v, t+1)$ such that v is activated in $\gamma_t \underset{u}{\rightarrow} \gamma_{t+1}$. We say that v executes a rule at time $t+1$. By convention, $(v, 0)$ is an event for all $v \in V$.

An event (v, t) is said to be external if the guard of the executed rule by v depends on at least one shared register of a neighbor of v.

An event (v, t) is a normal (resp. transparent, resp. convergence) event if v executes a normal (resp. transparent, resp. convergence) rule at time t.

The causal relation $\rightsquigarrow$ allows us to causally link two events that are not necessarily globally consecutive in an execution, due to activation of distant nodes, buy that are locally consecutive.

Definition 4.3 (*Causal relation* $\rightsquigarrow$)

The causal relation is the smallest relation $\rightsquigarrow$ on the set of events such that the following two conditions hold:

1. *Let (v, t) and (v, t') be two events such that t' is the greatest integer such that $t' < t$. Then $(v, t') \rightsquigarrow (v, t)$;*
2. *Let (v, t) and (w, t') be two events such that (v, t) is an external event, $w \in N_v$, t' is the greatest integer such that $t' < t$. Then $(w, t') \rightsquigarrow (v, t)$.*

The dependence relation $\rightsquigarrow^N$ allows us to formalize the notion that a node increases its clock immediately after one other node increased its variable clock. Note that any activation of a convergence rule on the node that increased its clock first beaks the dependence relation.

We also denote by $\preceq^N$ the transitive closure of $\rightsquigarrow^N$.

Definition 4.4 (*Dependence Relation* $\rightsquigarrow^N$)

Let (v, t_0) and (w, t') be two events.

We say that (w, t') normally depends on (v, t_0), and denote $(v, t_0) \rightsquigarrow^N (w, t')$ if there exists $k \geq 0$ and $t_1, \ldots t_k$ such that $(v, t_0) \rightsquigarrow (v, t_1) \rightsquigarrow \cdots \rightsquigarrow (v, t_k) \rightsquigarrow (w, t')$ and $\forall i > 0, (v, t_i)$ is a transparent event, and (w, t') is a normal event.

We denote $(v, t) \preceq^N (w, t')$ if there is a path w.r.t *to $\rightsquigarrow^N$ from (v, t) to (w, t').*

An N-sequence corresponds to consecutive clock increase on one node, without any execution of a convergence rule.

Definition 4.5 (*N-sequence*)

When a node v consecutively executes k normal rules, possibly intercut with transparent rules,

$$\rightsquigarrow^N (v, t_0) \rightsquigarrow^N (v, t_1) \rightsquigarrow^N \cdots \rightsquigarrow^N (v, t_{k-1})$$

it executes an N-sequence of length k.

A causal pyramid is the extension of the notion of dependence relation to interconnected interactions of neighbors to neighbors. Intuitively, if one node v executes numerous, say N, normal actions, then its neighbors will necessarily execute at least $N-2$ normal actions between the first and last action of v, and the neighbors of v's neighbors at least $N-4$ during the first and last action of their common neighbor with v, and so on. Once again, we consider that any execution of a convergence rule breaks the pyramid. A causal pyramid scheme is presented on Figure 4.2

We denote (i, j) the set of all integers k such that $i < k < j$.

Definition 4.6 (*Causal pyramid*)

Let $p = v_0 v_1 \dots v_k$ be a path of length k in G, and let us consider $d \geq 2k+1$. We say that p is a causal pyramid of length d and of origin t_0^0 if

$$\forall i \in [0,k], \forall j \in [i,(d-1)-i], \exists t_j^i \text{ such that}$$

- $\forall i$, we have $t_i^i < t_{i+1}^{i+1}$ and $t_{(d-1)-(i+1)}^{i+1} < t_{(d-1)-i}^i$
- $\forall i, v_i$ does not execute rules of $R_{\mathcal{U}C}$ in $(t_{i-1}^{i-1}, t_{(d-1)-(i-1)}^{i-1})$
- $\forall i, v_i$ executes an N-sequence $\rightsquigarrow^N (v_i, t_i^i) \rightsquigarrow^N (v_i, t_{i+1}^i) \rightsquigarrow^N \dots \rightsquigarrow^N (v_i, t_{(d-1)-i}^i)$

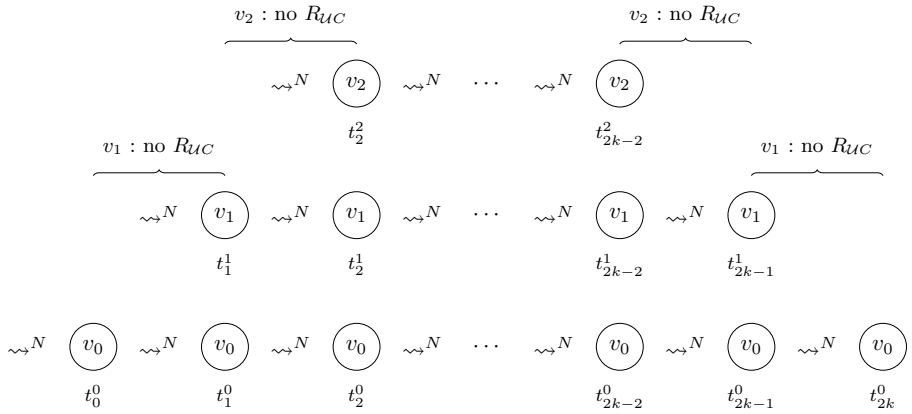

Figure 4.2: Causal Pyramid Scheme: $v_0 v_1 v_2$ is a causal pyramid of length $2k+1$ and of origin t_0^0

Notice that if p is a causal pyramid, then $(v_0, t_0^0) \preceq^N (v_1, t_1^1) \preceq^N \dots \preceq^N (v_k, t_k^k) \preceq^N (v_k, t_{(d-1)-k}^k) \preceq^N \dots \preceq^N (v_0, t_{d-1}^0)$.

4.3.4 Properties of Unison Algorithms

In this section, we extend and adapt some of the results of [BLP08] to any unison algorithm, and establish Theorem 4.1 that will allow us to establish synchronization properties of our termination detection algorithm $\mathcal{T}$.

Lemma 4.1

Let v and w be two neighbors. Suppose that v is a causal pyramid of length 3 and of origin t_0, and that in (t_0, t_2), w does not execute rules of $R_{\mathcal{U}_C}$. Then vw is a causal pyramid of length 3 and of origin t_0.

Proof: Let us denote by p the value of clock_v at time $t_0 - 1$. Since v executes three consecutive normal actions, at time t_0, t_1, and t_2, it increases its clock by three. Furthermore, since v executes a normal action at t_0, we have $P^+(v)$ at $t_0 - 1$ and thus clock_w is equal to p or $p+1$.

If w does not execute any normal action between t_0 and t_2, then when v executes its third normal action, at $t_2 - 1$, we have $\mathsf{clock}_w \in \{p, p+1\}$ and $\mathsf{clock}_v = p+2$, so $\neg P^+(v)$, which is contradictory.

■

The following lemma directly follows by induction of Lemma 4.1 on the length of an N-sequence.

Lemma 4.2

Let v and w be two neighbors. Suppose that v is a causal pyramid of length $d+1, d \geq 2$ and of origin t_0, and that in (t_0, t_d), w does not execute rules of $R_{\mathcal{U}_C}$. Then vw is a causal pyramid of length $d+1$ and of origin t_0.

Proof: Let us denote by p the value of clock_v at time $t_0 - 1$. Since v executes $d+1$ consecutive normal actions, at time $t_0, t_1, \ldots, t_d$, its clock in incremented $d+1$ times. Furthermore, since for any $i \in [0, d]$, v executes a normal action at time t_i, then we have $P^+(v)$ at $t_i - 1$, for any $i \in [0, d]$.

Therefore, for any i, we have at $t_i - 1$, $\mathsf{clock}_w \in \{\mathsf{clock}_v, \mathsf{clock}_v + 1\}$. In other words, for any $i \in [0, d]$, we have at $t_i - 1$, $\mathsf{clock}_w \in \{p + i, p + i + 1\}$. Since w does not execute any convergence action during (t_0, t_d), this is possible only if w increases $d - 1$ times its clock during (t_0, t_d). This means that w executes an N-sequence of length $d - 1$ during the N-sequence of v.

This corresponds to the definition of vw being a causal pyramid of length $d + 1$. ■

Lemma 4.2 establishes a link between the behavior of two neighbors. By induction on the distance between v and any other process, Theorem 4.1 follows.

Theorem 4.1

Let v_0 and v_k be two nodes, and let $p = v_0 v_1 \cdots v_k$ be a path. Suppose that $v_0 \ldots v_{k-1}$ is a causal pyramid of length $d \geq 2k + 1$, and that in $(t_{k-1}^{k-1}, t_{(d-1)-(k-1)}^{k-1})$, v_k does not execute rules of $R_{\mathcal{U}_C}$. Then $v_0 \cdots v_k$ is a causal pyramid of length $2k + 1$.

Proof: We simply apply Lemma 4.2 to v_{k-1}, which is a causal pyramid of length $d-2(k-1) \geq 3$ and of origin t_{k-1}^{k-1}. This implies that v_k executes a N-sequence of length at least $d - 2k$ during $(t_{k-1}^{k-1}, t_{(d-1)-(k-1)}^{k-1})$, and thus $v_0 \cdots v_k$ is a causal pyramid of length $2k + 1$. ■

Corollary 4.1

Let v_0 and v_k be two nodes, and let $p = v_0 v_1 \cdots v_k$ be a path. Suppose that v_0 is a causal pyramid of length $2k+1$, and that $\forall i, v_i$ does not execute rules of $R_{\mathcal{U}_C}$ in (t_0, t_{2k}). Then $v_0 \cdots v_k$ is a causal pyramid of length $2k + 1$.

Proof: We iterate Theorem 4.1 on v_0, then on $v_0 v_1$, and so on. ■

4.4 Algorithm

Let us consider a silent self-stabilizing algorithm $\mathcal{A}$ that solves a problem P under the unfair distributed scheduler. We introduce a generic mechanism that builds an anonymous silent snap-stabilizing request-based algorithm $\mathcal{T}$ that solves the Termination Detection of $\mathcal{A}$.

Based on an anonymous self-stabilizing unison algorithm $\mathcal{U}$, it follows the request-based mechanism described in Section 4.2.3. More specifically, for each node v, the algorithm communicates with the application *App* by means of req_v—refer to Figure 4.1. To know whether $\mathcal{A}$ has terminated or not, *App* triggers a request to $\mathcal{T}$ by executing *App*: *ask* on some idle nodes $v_1, v_2, \ldots, v_k$ of the system, setting the value of req_{v_i}, for $i \in [1, k]$, from idl

to on. Next, $\mathcal{T}$ answers to the request only after algorithm $\mathcal{A}$ has terminated, by setting req_v to off. Then, *App* may execute *App*: *get* that sets req_v to idl.

4.4.1 Scheme of Algorithm $\mathcal{T}$

The main idea of our algorithm is that each node has a variable dedicated to detect the activity of algorithm $\mathcal{A}$. When one node sees an activation of a rule of $\mathcal{A}$ (in our model, when one node simulates a rule of $\mathcal{A}$), it sets its variable to its maximal value, and then decrease it by one at each activation. This variable is also used to propagate the information that $\mathcal{A}$ has been activated to the neighbors of the nodes. Namely, nodes increase their own variable if one of their neighbors has a non-zero value.

When a request is emitted on one node, this node sets another countdown to its maximal value, and decrease it at each computing step. If when this countdown reaches 0, it has not detected any activation of $\mathcal{A}$ from its neighbors, then it can answer that $\mathcal{A}$ has terminated.

This works only if the decrease of both variables is relatively synchronized on the nodes of the system. This is why we need a unison algorithm. Both variables are synchronized with the clock of the unison algorithm. Therefore, thanks the properties established in Section 4.3, we can transfer properties of unison algorithms to our termination detection algorithm.

4.4.2 Variables

The variables of node v in algorithm $\mathcal{T}$ are:

- $\mathtt{state}_{v|\mathcal{A}}$ the set of all variables of node v in algorithm $\mathcal{A}$.
- $\mathtt{state}_{v|\mathcal{U}}$ the set of all variables of node v in algorithm $\mathcal{U}$, including clock_v.
- $\mathsf{da}_v \in [0, 2D+2]$, for detection of activity. This variable is used to store the number of steps since the last time a convergence rule (for $\mathcal{A}$ or $\mathcal{U}$) was executed by node v. Variable da propagates through the whole system with the following rule: if the maximum value of da among the neighbors of v is p, then v cannot set da_v under $p-1$.
- $\mathsf{dt}_v \in [0, 2D+1]$, for detection of termination, is a countdown to 0, initiated at $2D+1$ when the application asks for the termination.
- $\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}, \mathsf{off}, \mathsf{idl}\}$, for request, is the interface between *App* and $\mathcal{T}$. req_v may be updated according to Figure 4.1.

The space complexity of the variables of $\mathcal{T}$ is $O(\log D)$ bits per node, where D is an upper bound of the diameter of the graph. Consequently, the space complexity of the whole system is $O(\mathbb{S}(\mathcal{A}) + \mathbb{S}(U) + \log D)$ bits per node, where $\mathbb{S}(\mathcal{A})$ (resp. $\mathbb{S}(\mathcal{U})$) is the space complexity of algorithm $\mathcal{A}$ (resp. of algorithm $\mathcal{U}$) in bits per node.

4.4.3 Overview of the algorithm

The detail of the rules of algorithm $\mathcal{T}$ is presented in Algorithm 1. Algorithm $\mathcal{T}$ simulates both algorithms $\mathcal{A}$ and $\mathcal{U}$, independently. When an enabled node v is activated, v atomically executes the rule of $\mathcal{A}$ for which it is enabled, if such rule exists, the rule of $\mathcal{U}$ for which it is enabled, if such rule exists, and updates the proper variables of algorithm $\mathcal{T}$.

Since algorithm $\mathcal{T}$ performs two distinct and independent tasks: the simulation of $\mathcal{A}$ and $\mathcal{U}$, on the one hand, the detection of termination, on the other hand, it is natural to divide the set of the rules of $\mathcal{T}$, $R_{\mathcal{T}}$, in two disjoint sets: $\mathcal{R}^S_{\text{simul}}$ and $\mathcal{R}^S_{\text{proper}}$. A node is enabled for $\mathcal{T}$ if it enabled for at least one rule of $\mathcal{R}^S_{\text{simul}}$ or $\mathcal{R}^S_{\text{proper}}$. If an enabled node is activated, then it atomically executes the rule of $\mathcal{R}^S_{\text{simul}}$ for which it is enabled, if such a rule exists, and the rule of $\mathcal{R}^S_{\text{proper}}$ for which it is enabled, if such a rule exists.

$\mathcal{R}^S_{\text{simul}}$ contains the rules which simulate algorithms $\mathcal{A}$ and $\mathcal{U}$, and also updates the variable da. To facilitate reasoning, we distinguish several cases depending on which rules are enabled for $\mathcal{A}$ and $\mathcal{U}$. Figure 4.3 recaps the different possible situations, and the name of the corresponding rule of $\mathcal{R}^S_{\text{simul}}$.

If the activated node v has not yet converged for both $\mathcal{A}$ and $\mathcal{U}$, then it executes a convergence rule for at least one of those algorithms and sets its variable da_v to $2D + 2$. The set of all convergence rules is denoted $\mathbb{R}_{cvg}$, and it contains the following rules: $\mathbb{R}^{\mathcal{U}_C}_{cvg}, \mathbb{R}^{N}_{cvg}, \mathbb{R}^{\mathcal{A}}_{cvg}$. Those rules differ by the precise rules of $\mathcal{A}$ and $\mathcal{U}$ they simulate. Be aware that the rules of $\mathbb{R}_{cvg}$ are not necessarily convergence rules in the sense of unison algorithms. More precisely, $\mathbb{R}_{cvg}$ contains the convergence rules relative to $\mathcal{U}$, but also the convergence rules relative to $\mathcal{A}$.

Otherwise, if w is only enabled for a transparent rule of $\mathcal{U}$, then it executes $\mathbb{R}_{trans} \in \mathcal{R}^S_{\text{simul}}$ and nothing else.

Finally, if v is only enabled for a normal rule of $\mathcal{U}$ then v executes the rule $\mathbb{R}_{wait} \in \mathcal{R}^S_{\text{simul}}$, which sets da_v to one less than the maximum value of da of the closed neighbors of v. As a consequence, as long as one node v has not converged for $\mathcal{A}$, variable da is maintained at $2D + 2$ on v. When activated, the neighbors of v set their variable da to $2D + 1$ (or $2D + 2$ if one rule of $\mathbb{R}_{cvg}$ is activated). After that, if one neighbor of a neighbor of v is activated, then it sets its variable da to at least $2D$, and so on. Thus, if one node is enabled for a convergence rule, then the variable da will propagate at high value along the graph. Yet, this property requires that all the nodes are activated, in a specific order.

	$R_{\mathcal{U}_C}$	$R_{\mathcal{U}_T}$	$R_{\mathcal{U}_N}$	$\neg R_{\mathcal{U}}$
$R_{\mathcal{A}}$	$\mathbb{R}^{\mathcal{U}_C}_{cvg}$	$\mathbb{R}^{\mathcal{A}}_{cvg}$	$\mathbb{R}^{N}_{cvg}$	$\mathbb{R}^{\mathcal{A}}_{cvg}$
$\neg R_{\mathcal{A}}$	$\mathbb{R}^{\mathcal{U}_C}_{cvg}$	$\mathbb{R}_{trans}$	$\mathbb{R}_{wait}$	

Figure 4.3: Rules of $\mathcal{R}^S_{\text{simul}}$ depending on the enabled rules for $\mathcal{A}$ and $\mathcal{U}$. Rules of $\mathbb{R}_{cvg}$ are convergence rules for $\mathcal{U}$ or $\mathcal{A}$.
For $\mathcal{T}$, convergence (*resp.* transparent, *resp.* normal) rules are rules simulating a convergence (*resp.* transparent, *resp.* normal) rule of $\mathcal{U}$. We extend that definition to $\mathbb{R}^{\mathcal{A}}_{cvg}$, which is a transparent rule.

Fortunately, since $\mathcal{U}$ is a unison algorithm, no node, and no subset of nodes, may compute independently of the rest of the system, and decrease its variable da down to 0 regardless of what happens in the whole system. This guarantees that, starting from any configuration, after a node v executes $2D + 1$ computing steps, the variable da_v is under the influence of all the other nodes of the system. Consequently, after $2D + 1$ computing steps of node v, the variable da_v cannot be 0 unless all the nodes in the graph have converged for both $\mathcal{A}$ and $\mathcal{U}$. This property allows us to design our procedure thanks to the variable dt.

Whenever *App* asks one node v for the termination of the algorithm, v executes rule $\mathbb{R}_{start}$, it sets its variable dt_v to $2D+1$ and updates its variable req_v from on to wk. Variable dt_v is decreased by one each time a normal rule of $\mathcal{U}$ is executed, by rule $\mathbb{R}_{cpt}$, and is updated

to $2D+1$ as soon as $\mathsf{da}_v \neq 0$ by rule $\mathbb{R}_{end}$. Since $\mathcal{U}$ is a unison algorithm, in a legitimate execution dt_v reaches 0 (and therefore an answer is produces through rule $\mathbb{R}_{stop}$) only after all nodes have converged for $\mathcal{A}$. As a desired consequence, algorithm $\mathcal{T}$ is snap-stabilizing for the detection of termination for algorithm $\mathcal{A}$.

4.4.4 Predicates

Let $RS \in R_{\mathcal{A}}, R_{\mathcal{U}}, R_{\mathcal{U}_C}, R_{\mathcal{U}_T}, R_{\mathcal{U}_N}$ be the set of the rules of one algorithm, and let v be a node. $\mathsf{Act}(RS, v)$ returns `true` if and only if node u is enabled by a rule of RS.

$$\mathsf{Act}(RS, v) \equiv v \text{ is enabled by one rule of } RS \tag{4.1}$$

As described in Section 4.3.1, we consider a silent self-stabilizing unison algorithm $\mathcal{U}$. In $\mathcal{T}$, the normal rules of $\mathcal{U}$, denoted $R_{\mathcal{U}_N}$, are not enabled on node v unless the following predicate $\mathsf{LocReq}(v)$ is evaluated to `true`.

$$\mathsf{LocReq}(v) \equiv \bigvee \begin{array}{l} (\exists u \in N_v : \mathsf{clock}_u = \mathsf{clock}_v + 1) \\ \mathsf{Act}(R_{\mathcal{A}}, v) \\ (\mathsf{da}_v \neq 0) \\ (\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}\}) \end{array} \tag{4.2}$$

When both algorithm $\mathcal{A}$ and $\mathcal{U}$ have converged on node v, the only simulation rules that v may execute are the normal rules of algorithm $\mathcal{U}$, which permit the liveness of $\mathcal{U}$. This situation is described by the predicate $\mathsf{UnisonOnly}$.

$$\mathsf{UnisonOnly}(v) \equiv \neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_N}, v) \tag{4.3}$$

4.4.5 Actions

Let $RS \in R_{\mathcal{A}}, R_{\mathcal{U}}, R_{\mathcal{U}_C}, R_{\mathcal{U}_T}, R_{\mathcal{U}_N}$ be the set of the rules of one algorithm, and let v be a node. Let $\mathsf{SimulR}(RS, v)$ be a procedure that executes the enabled rule in RS on node v if such rule exists and which does nothing otherwise. Procedure $\mathsf{Simul}(v)$ sequentially executes one rule of Algorithm $\mathcal{A}$ if possible, then one rule of Algorithm $\mathcal{U}$, again if possible. Formally:

$$\mathsf{SimulR}(RS, v) \equiv \begin{cases} v \text{ updates its state executing the enabled rules in } RS & \text{if } \mathsf{Act}(RS, v) \\ v \text{ does nothing} & \text{otherwise} \end{cases} \tag{4.4}$$

$$\mathsf{Simul}(v) \equiv \mathsf{SimulR}(R_{\mathcal{A}}, v); \mathsf{SimulR}(R_{\mathcal{U}}, v) \tag{4.5}$$

Procedure $\mathsf{Propagate\_da}(v)$ updates the variable da_v to one less than the maximal value of da of the closed neighbors of v:

$$\mathsf{Propagate\_da}(v) \equiv \mathsf{da}_v := \max(0, \max_{u \in N[v]}(\mathsf{da}_u - 1)) \tag{4.6}$$

4.5 Correctness of Algorithm $\mathcal{T}$

In this section, we establish that $\mathcal{T}$ is a snap-stabilizing procedure for the detection of the termination of Algorithm $\mathcal{A}$. This proof is divided in four subsections.

Algorithm 1: Algorithm $\mathcal{T}$

During a step, if a rule of set $\mathcal{R}^S_{\text{simul}}$ and a rule of set $\mathcal{R}^S_{\text{proper}}$ are enabled then v executes these 2 rules.

$\mathcal{R}^S_{\textbf{simul}}$ :: **rules to update** da

$\mathbb{R}^{\mathcal{U}_C}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{U}_C}, v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{da}_v := 2D+2$	$\in R_{\mathcal{T}_C}$
$\mathbb{R}^{\mathcal{A}}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\big(\mathsf{Act}(R_{\mathcal{U}_C}, v) \vee \mathsf{Act}(R_{\mathcal{U}_N}, v)\big)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{da}_v := 2D+2$	$\in R_{\mathcal{T}_T}$
$\mathbb{R}_{trans}$	:	$\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_T}, v)$	$\longrightarrow \mathsf{Simul}(v)$	$\in R_{\mathcal{T}_T}$
$\mathbb{R}^{N}_{cvg}$	:	$\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{Act}(R_{\mathcal{U}_N}, v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{da}_v := 2D+2$	$\in R_{\mathcal{T}_N}$
$\mathbb{R}_{wait}$	:	$\mathsf{UnisonOnly}(v)$	$\longrightarrow \mathsf{Simul}(v); \mathsf{Propagate\_da}(v)$	$\in R_{\mathcal{T}_N}$

$\mathcal{R}^S_{\textbf{proper}}$ :: **rules to update** dt **and** req

$\mathbb{R}_{start}$	:	$\mathsf{req}_v = \mathsf{on}$	$\longrightarrow \mathsf{req}_v := \mathsf{wk}; \mathsf{dt}_v := 2D+1$
$\mathbb{R}_{stop}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{dt}_v = 0 \wedge \mathsf{da}_v = 0$	$\longrightarrow \mathsf{req}_v = \mathsf{off}$
$\mathbb{R}_{cpt}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{dt}_v \neq 0 \wedge \mathsf{da}_v = 0$	$\longrightarrow \mathsf{dt}_v := \mathsf{dt}_v - 1$
$\mathbb{R}_{end}$	:	$\mathsf{req}_v = \mathsf{wk} \wedge \mathsf{UnisonOnly}(v) \wedge \mathsf{da}_v \neq 0$	$\longrightarrow \mathsf{dt}_v := 2D+1$

In Subsection 4.5.1, we prove that $\mathcal{T}$ satisfies both Conditions 1 and 2 of Specification 4.2.

In Subsection 4.5.2, we prove that $\mathcal{T}$ satisfies Condition $3-i$ of Specification 4.2.

Finally, in Subsection 4.5.3, we prove that $\mathcal{T}$ satisfies Condition $3-ii$ of Specification 4.2.

We also prove in Subsection 4.5.4 the time complexity of our algorithm.

4.5.1 Simulation properties of $\mathcal{T}$

In this section, we establish Theorem 4.2, which ensures that $\mathcal{T}$ satisfies Condition 1 and Condition 2 of Specification 4.2.

Theorem 4.2

$\mathcal{T}$ is a simulation of both $\mathcal{A}$ and $\mathcal{U}$, and in any execution of $\mathcal{T}$, both $\mathcal{A}$ and $\mathcal{U}$ ultimately converge.

Theorem 4.2 is a consequence of Lemmas 4.4, 4.6, and 4.7.

Before anything else, let us establish Lemma 4.3, that characterizes the terminal configurations of $\mathcal{T}$. These configurations are the one in which $\mathcal{U}$ converged, and in which on all the nodes v of the system, $\mathsf{LocReq}(v)$ is not verified (this includes the termination of $\mathcal{A}$).

Lemma 4.3

The terminal configurations of $\mathcal{T}$ are the configurations such that $\forall v \in V$:

$$\neg\mathsf{Act}(R_{\mathcal{U}}, v) \wedge \neg\mathsf{LocReq}(v)$$

Proof: Let γ be a terminal configuration of $\mathcal{T}$. If $\exists u \in V : \mathsf{Act}(R_{\mathcal{A}}, v) \vee \mathsf{Act}(R_{\mathcal{U}}, v)$, then one rule of $\mathcal{R}^S_{\text{simul}}$ is enabled on node v, so $\forall v \in V, \neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}}, v)$. Furthermore, since $\mathcal{U}$ is a self-stabilizing algorithm, it satisfies the *liveness* property. Thus, unless all the nodes $v \in V$ satisfy $\neg\mathsf{LocReq}(v)$, there exist enabled nodes in the system. Consequently, $\forall v \in V, (\forall u \in N_v, \mathsf{clock}_u \in \{\mathsf{clock}_v - 1, \mathsf{clock}_v\}) \wedge \neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \mathsf{da}_v = 0 \wedge (\mathsf{req}_v \in \{\mathsf{off}, \mathsf{idl}\})$.

Conversely, such configurations are terminal configurations. The rules of $\mathcal{R}^S_{\text{simul}}$ are not enabled because $\forall v \in V, \neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}}, v)$, and the rules of $\mathcal{R}^S_{\text{proper}}$ are not enabled because $\forall v \in V, \mathsf{req}_v \in \{\mathsf{off}, \mathsf{idl}\}$. ■

By the definition of the actions of $\mathcal{T}$, any execution of $\mathcal{T}$ simulates both Algorithm $\mathcal{A}$ and Algorithm $\mathcal{U}$. This is stated in Lemma 4.4.

Lemma 4.4

Any execution of $\mathcal{T}$ is a simulation of both $\mathcal{A}$ and $\mathcal{U}$.

Proof: The only action that updates the variables of $\texttt{state}_{v|\mathcal{A}}$ (resp. of $\texttt{state}_{v|\mathcal{U}}$) is $\mathsf{SimulR}(R_{\mathcal{A}}, v)$ (resp. $\mathsf{SimulR}(R_{\mathcal{U}}, v)$). It is applied only if the activated nodes are enabled for $\mathcal{A}$ (resp. for $\mathcal{U}$). Thus, the execution of $\mathcal{T}$ is a simulation of $\mathcal{A}$ (resp. of $\mathcal{U}$). ■

Since $\mathcal{A}$ is a silent self-stabilizing algorithm, there do not exist infinite executions of $\mathcal{A}$. Since executions of $\mathcal{T}$ are simulations of executions of $\mathcal{A}$, then the number of computing steps of an execution of $\mathcal{T}$ that execute a rule of $\mathcal{A}$ is finite. This is stated in Lemma 4.5, and will be useful to prove the convergence of $\mathcal{U}$.

Lemma 4.5

In any execution of $\mathcal{T}$, a finite number of computing steps execute rules of $\mathcal{A}$.

Proof: Let $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ be an execution of $\mathcal{T}$. Suppose there exists an infinite number of computing steps of ϵ that execute rules of $\mathcal{A}$. Then, $\epsilon_{\mathcal{A}} = (\gamma_{0|\mathcal{A}} \underset{\mathcal{T}}{\rightarrow} \gamma_{1|\mathcal{A}} \underset{\mathcal{T}}{\rightarrow} \cdots)$ is an infinite execution of $\mathcal{A}$. Since $\mathcal{A}$ is a silent self-stabilizing algorithm under the unfair scheduler, it converges in finite time in any execution of $\mathcal{A}$. Thus, such infinite execution does not exist. ■

Since $\mathcal{A}$ is activated only a finite number of times, and since $\mathcal{U}$ is a self-stabilizing algorithm, it indeed ultimately converges in any execution of $\mathcal{T}$. This is stated in Lemma 4.6.

Lemma 4.6

$\mathcal{U}$ converges in any maximal execution of $\mathcal{T}$.

Proof: Let $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ be a maximal execution of $\mathcal{T}$. According to Lemma 4.5, we can assume without loss of generality that in ϵ, no rule of Algorithm $\mathcal{A}$ is executed. Let us reason by contradiction and suppose $\mathcal{U}$ does not converge in ϵ. According to Lemma 4.3, ϵ cannot contain any terminal configuration, so ϵ is an infinite execution. Since $\mathcal{U}$ is a self-stabilizing algorithm under the unfair scheduler, it converges in finite time in any execution of $\mathcal{U}$. If ϵ contains an infinite number of computing steps that execute rules of $\mathcal{U}$, then $\mathcal{U}$ converges. Consequently, there only exists a finite number of such computing steps. We now consider an infinite suffix ϵ' of ϵ that does not contain any rule of $\mathcal{U}$ nor of $\mathcal{A}$. All the rules of $\mathcal{R}^S_{\text{simul}}$ induce a rule of $\mathcal{U}$ or $\mathcal{A}$. Furthermore, the predicates of $\mathbb{R}_{stop}$, $\mathbb{R}_{cpt}$, and $\mathbb{R}_{end}$ all include $\mathsf{UnisonOnly}$ which is the guard of $\mathbb{R}_{wait}$. The only rule executed in ϵ' is $\mathbb{R}_{start}$, but it cannot be executed twice on a node v without v executes the rule $\mathbb{R}_{stop}$ (and several times $\mathbb{R}_{cpt}$) in the meantime. This is contradictory. Consequently, $\mathcal{U}$ converges in ϵ. ■

Finally, since $\mathcal{U}$ is a unison algorithm, all the nodes are regularly activated in any execution, and since $\mathcal{A}$ is a self-stabilizing algorithm, this implies that the convergence of $\mathcal{A}$ ultimately occurs in any maximal execution of $\mathcal{T}$. This is stated in Lemma 4.7.

Lemma 4.7

$\mathcal{A}$ converges in any maximal execution of $\mathcal{T}$.

Proof: Let us consider $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ a maximal execution of $\mathcal{T}$. According to Lemma 4.6, we can assume without loss of generality that in γ_0, $\mathcal{U}$ has already converged. Since $\mathcal{U}$ is a unison algorithm and since $\mathsf{Act}(R_{\mathcal{A}}, v)$ is included in $\mathsf{LocReq}(v)$, the correctness of $\mathcal{U}$ ensures that as long as $\mathcal{A}$ has not converged, all the nodes are regularly activated. Since $\mathcal{A}$ is a self-stabilizing algorithm for the unfair scheduler, this guarantees that $\mathcal{A}$ converges. ■

This completes the proof of Theorem 4.2.

Remark 4.1

Since $\mathcal{T}$ is a simulation of $\mathcal{U}$, we can extend the results of Section 4.3.4 to the executions of $\mathcal{T}$. To do this, we extend to $\mathcal{T}$ the concepts of normal, transparent, and convergence rule, normal dependence relation, and N-sequence. $\mathcal{T}$ has one convergence rule $\mathbb{R}_{cvg}^{\mathcal{U}_C}$, two transparent rules $\mathbb{R}_{cvg}^{\mathcal{A}}$ and $\mathbb{R}_{trans}$, and two normal rules $\mathbb{R}_{cvg}^{N}$ and $\mathbb{R}_{wait}$. Specifically, Theorem 4.1 and Corollary 4.1 remain valid on $\mathcal{T}$.

4.5.2 Termination of $\mathcal{T}$

In this section, we establish that maximal executions of $\mathcal{T}$ satisfy Condition $3-i$ of Specification 4.2 (Theorem 4.4).

We define $\Gamma_{\mathsf{cvg}} \subset \Gamma$ the set of configurations in which $\mathcal{A}$ has stabilized and $\mathcal{U}$ has converged, and Γ_{da} the set of configurations in which, in addition, $\forall v, \mathsf{da}_v = 0$. Lemma 4.8 states that Γ_{da} is an attractor.

Definition 4.7

Let $\Gamma_{\mathsf{da}} \subset \Gamma_{\mathsf{cvg}} \subset \Gamma$ be

$$\Gamma_{\mathsf{cvg}}: \quad \forall v \in V, (\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, v))$$
$$\Gamma_{\mathsf{da}}: \quad \forall v \in V, (\neg\mathsf{Act}(R_{\mathcal{A}}, v) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, v) \wedge \mathsf{da}_v = 0)$$

Lemma 4.8

$$\Gamma \triangleright_{\mathcal{T}} \Gamma_{\mathsf{cvg}} \triangleright_{\mathcal{T}} \Gamma_{\mathsf{da}}$$

Proof: Γ_{cvg} is closed since $\mathcal{A}$ and $\mathcal{U}$ are self-stabilizing algorithms. Theorem 4.2 ensures $\Gamma \triangleright_{\mathcal{T}} \Gamma_{\mathsf{cvg}}$.

Let us prove that Γ_{da} is closed. Let $\gamma \underset{\mathcal{T}}{\rightarrow} \gamma'$ be a computing step such that $\gamma \in \Gamma_{\mathsf{da}}$. Since $\gamma \in \Gamma_{\mathsf{cvg}}$ which is closed, $\gamma' \in \Gamma_{\mathsf{cvg}}$. In γ, no rule of $\mathbb{R}_{cvg}$ is enabled, so only $\mathbb{R}_{wait}$ may update da. But since $\forall v \in V, \mathsf{da}_v^{\gamma} = 0$, then $\mathsf{Propagate\_da}(v)$ has no effect. Thus, $\gamma' \in \Gamma_{\mathsf{cvg}}$.

Let us now prove that $\mathcal{T}$ converges to Γ_{da} from Γ_{cvg}. Let $\gamma_0 \in \Gamma_{\mathsf{cvg}}$ be a configuration, and let us consider a maximal execution $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$. According to Lemma 4.3, if ϵ is finite, then the terminal configuration belongs to Γ_{da}. We suppose now that this execution is infinite. Since Γ_{cvg} is closed, we are only interested in the value of da.

Let us consider $C_i = \max_{v \in V} \mathsf{da}_v^{\gamma_i}$. If $C_0 = 0$, then $\gamma_0 \in \Gamma_{\mathsf{da}}$. Otherwise, since ϵ is infinite, and $\mathcal{U}$ is a unison algorithm, and $\mathbb{R}_{cvg}^{\mathcal{A}}$ is not enabled in Γ_{cvg}, all the nodes execute $\mathbb{R}_{wait}$ infinitely many times. Let i_1 such that before reaching γ_{i_1}, all the nodes have executed $\mathbb{R}_{wait}$. We have $C_{i_1} < C_0$. By induction, there exists a time t such that $C_t = 0$, so we obtain $\gamma_t \in \Gamma_{\mathsf{da}}$. ■

Finally, we define Γ_{dt}^v where $v \in V$, as the set of the configurations of Γ_{da} such that $\mathsf{req}_v \in \{\mathsf{off}, \mathsf{idl}\}$. Theorem 4.3 states that any execution that starts in Γ_{da} eventually reaches a configuration of Γ_{dt}^v.

Definition 4.8

Let $v \in V$. Let $\Gamma^v_{\mathsf{dt}} \subset \Gamma_{\mathsf{da}}$ be the set of configurations such that, $\forall u \in V$:

$$(\neg\mathsf{Act}(R_{\mathcal{A}}, u) \wedge \neg\mathsf{Act}(R_{\mathcal{U}_C}, u) \wedge \mathsf{da}_u = 0) \wedge \mathsf{req}_v \in \{\mathsf{off}, \mathsf{idl}\}$$

Theorem 4.3

Let $v \in V$ be a node and $\epsilon = (\gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ be a maximal execution such that $\gamma_0 \in \Gamma_{\mathsf{da}}$. There exists $i \geq 0 : \gamma_i \in \Gamma^v_{\mathsf{dt}}$.

Proof: Suppose $\gamma_0 \notin \Gamma^v_{\mathsf{dt}}$, *i.e.* $\mathsf{req}^{\gamma_0}_v \in \{\mathsf{on}, \mathsf{wk}\}$.

Case 1: $\mathsf{req}^{\gamma_0}_v = \mathsf{wk}$.

Since Γ_{da} is closed, the only rules of $\mathcal{R}^S_{\text{proper}}$ enableds on v are $\mathbb{R}_{stop}$ and $\mathbb{R}_{cpt}$. As long as $\mathsf{req}_v \in \{\mathsf{on}, \mathsf{wk}\}$, $\mathsf{LocReq}(v)$ is satisfied, and since $\mathcal{U}$ is a unison algorithm, it makes progress, which means that $\mathbb{R}_{wait}$ is regularly activated on all nodes. Thus, as long as $\mathsf{dt}_v \neq 0$, $\mathbb{R}_{cpt}$ is regularly activated by v. Since each activation of $\mathbb{R}_{cpt}$ by v decreases dt_v by 1, it ultimately reaches 0, after what $\mathbb{R}_{stop}$ is activated by v, and then, the system has reached Γ^v_{dt}.

Case 2: $\mathsf{req}^{\gamma_0}_w = \mathsf{on}$.

Then the previous guarantees as well that node w will eventually be activated, through rule $\mathbb{R}_{start}$, after which $\mathsf{req}_w = \mathsf{wk}$, which is the case above. ■

Lemma 4.8 and Theorem 4.3 basically prove that, whatever the initial configuration, the system globally converges to Γ_{da}, and each node is infinitely often in an availability state for the app. This proves that $\mathcal{T}$ satisfies Condition 3-i of Specification 4.2. Let us now formally prove Theorem 4.4:

Theorem 4.4

Let $\epsilon = \gamma_0 \underset{\mathcal{T}}{\rightarrow} \cdots$ be a maximal execution such that $\exists v \in V, t \geq 0 : \mathsf{req}^{\gamma_t}_v = \mathsf{on}$.
There exists $t' > t$ such that $\mathsf{req}^{\gamma_{t'}}_v = \mathsf{off}$

Proof: Let us consider the subexecution $\epsilon_t = \gamma_t \underset{\mathcal{T}}{\rightarrow} \cdots$. According to Lemma 4.8, there exists $t_a \geq t$ such that $\gamma_{t_a} \in \Gamma_{\mathsf{da}}$. Let us consider $\epsilon_a = \gamma_{t_a} \underset{\mathcal{T}}{\rightarrow} \cdots$. According to Theorem 4.3, there exists $t_d \geq t_a$ such that $\mathsf{req}^{\gamma_{t_d}}_v \in \{\mathsf{off}, \mathsf{idl}\}$.

Since $\mathsf{req}^{\gamma_t}_v = \mathsf{on}$ and $\mathsf{req}^{\gamma_{t_d}}_v \in \{\mathsf{off}, \mathsf{idl}\}$, there exists $t' > t$ such that $\mathsf{req}^{\gamma_{t'}}_v = \mathsf{off}$. ■

4.5.3 Snap-stabilization

In this section, we establish that maximal executions of $\mathcal{T}$ satisfy Condition $3 - ii$ of Specification 4.2 (Theorem 4.7).

The activation of $\mathbb{R}_{start}$ by v corresponds to the request by v to detect the termination of Algorithm $\mathcal{A}$. The activation of $\mathbb{R}_{stop}$ by v corresponds to the detection of the termination of Algorithm $\mathcal{A}$ by v.

Lemma 4.9 states that v will eventually execute the rule $\mathbb{R}_{stop}$ along any execution starting by a termination detection request by v (*i.e.* the activation of $\mathbb{R}_{start}$ by v). This lemma allows us to define, for a maximal execution starting by a termination detection request by v, the response time to v's request: f, the time of the first computing step in which v executes the rule $\mathbb{R}_{stop}$. This is stated in Definition 4.9.

Lemma 4.9

Let $v \in V$ and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\rightarrow} \gamma_0$. $\exists i > 0$ such that in $\gamma_{i-1} \underset{\tau}{\rightarrow} \gamma_i$, v executes $\mathbb{R}_{stop}$.

Proof: This is a direct corollary of Theorem 4.4 ■

Definition 4.9 (*Time of Response*)

Let $v \in V$ and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\rightarrow} \gamma_0$.

We denote by $\mathsf{f}(v, \epsilon)$, or simply f the time of response *of v in ϵ, which is the smallest $i > 0$ such that during $\gamma_{i-1} \underset{\tau}{\rightarrow} \gamma_i$, v executes $\mathbb{R}_{stop}$.*

To prove that $\mathcal{T}$ satisfies Condition 3-ii of Specification 4.2, we only have to prove that $\mathcal{A}$ has terminated in γ_f.

Before an answer is provided by $\mathcal{T}$, a lot of computations can be required. Notably, v can execute $\mathbb{R}_{end}$ numerous times, which resets dt_v up to its maximal value. In our reasoning, we will focus on the subexecution that follows the last time v resets dt_v, which is the part where v does not see any activity from $\mathcal{A}$ during a sufficiently long period of time for it to answer. This is presented in Figure 4.4.

Let us define that subexecution:

Definition 4.10 (*Final Descent*)

Let $v \in V$ and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\rightarrow} \gamma_0$, and let f be the time of response of v in ϵ. Let us consider $\epsilon_f = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \cdots \underset{\tau}{\rightarrow} \gamma_{f-1})$.

If v executes $\mathbb{R}_{end}$ along ϵ_f then let us define $\mathsf{cs}_s = \gamma_{s-1} \underset{\tau}{\rightarrow} \gamma_s$, the latest computing step of ϵ_f in which v executes $\mathbb{R}_{end}$.

Otherwise we set $s = 0$ and $\mathsf{cs}_s = \gamma \underset{\tau}{\rightarrow} \gamma_0$.

Remark that cs_s is the latest computing step of ϵ_f in which v executes a rule that sets dt_v at $2D + 1$.

We denote by ϵ_s the final descent *of v in ϵ, $\epsilon_s = \gamma_s \underset{\tau}{\rightarrow} \gamma_{s+1} \underset{\tau}{\rightarrow} \cdots \underset{\tau}{\rightarrow} \gamma_{f-1}$.*

In the sequel of this section, we study the properties of ϵ_s. We establish in Theorem 4.6 that $\gamma_s \in \Gamma_{\mathsf{cvg}}$. As Γ_{cvg} is closed, we also have $\gamma_f \in \Gamma_{\mathsf{cvg}}$: a terminal configuration of $\mathcal{A}$ is reached when the answer is provided. Basically, the scheme of the following is:

Proposition 4.1

To prove that in γ_s, $\mathcal{A}$ has terminated, we must prove that if $\mathcal{A}$ had not, then:

1. *a rule of $\mathcal{A}$ would be executed on one node u, and*
2. *this action would have been transmitted to node v before f, postponing γ_s.*

For the sake of the proof, we aim at finding one early such activation of a rule of $\mathcal{A}$.

We prove the first part of Proposition 4.1 in Theorem 4.5, and the second part in the proof of Theorem 4.6 (which is a proof by contradiction, so the statement of Theorem 4.6 concludes that $\mathcal{A}$ has terminated in γ_s).

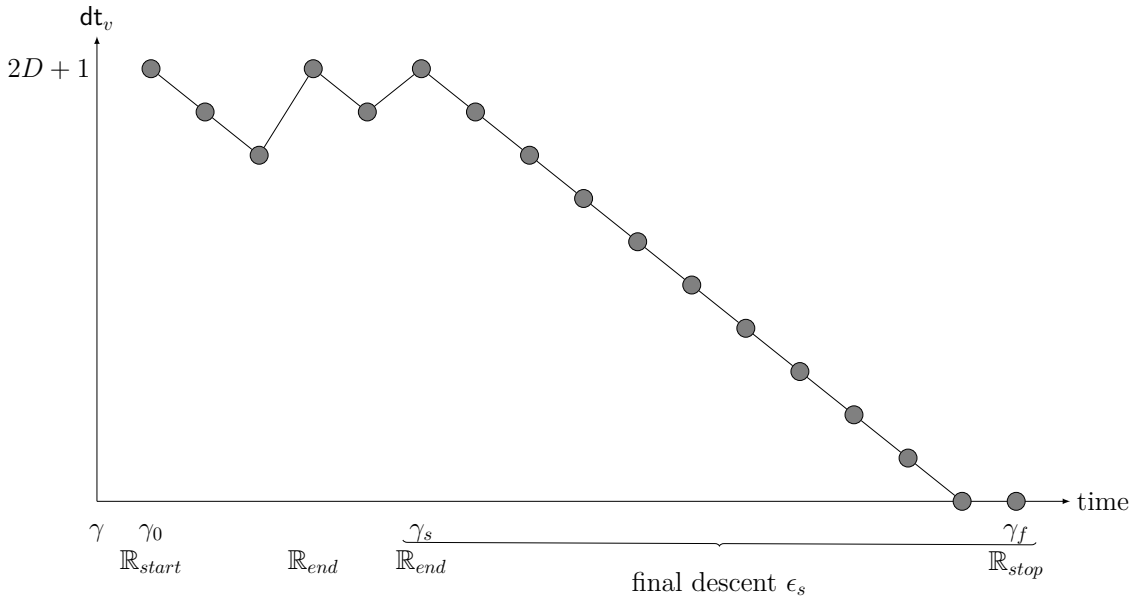

Figure 4.4: Example of Final Descent on node v, progression indicated by the decrease of dt_v

To prove that nodes are necessarily activated, and that nodes influence themselves through the network, we need the properties of Unison algorithms established in Section 4.3.4, which apply to $\mathcal{T}$ as noted in Remark 4.1.

Lemma 4.10 allows us to use Corollary 4.1.

Lemma 4.10

Let v be a node and let $\epsilon = (\gamma \underset{\mathcal{T}}{\rightarrow} \gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\mathcal{T}}{\rightarrow} \gamma_0$.

Then v does not execute any rule of $\mathbb{R}_{cvg}$ in ϵ_s, and v executes an N-sequence of length at least $2D+1$ in ϵ_s.

Proof: By contradiction, suppose v executes a rule of $\mathbb{R}_{cvg}$ in $\mathsf{cs}_i = \gamma_{i-1} \underset{\mathcal{T}}{\rightarrow} \gamma_i$ a computing step of ϵ_s.

Then, $\mathsf{da}_v^{\gamma_i} = 2D+2$. By construction, v executes $\mathbb{R}_{stop}$ in $\gamma_{f-1} \underset{\mathcal{T}}{\rightarrow} \gamma_f$. According to the algorithm and guard definitions, v also executes $\mathbb{R}_{wait}$ in $\gamma_{f-1} \underset{\mathcal{T}}{\rightarrow} \gamma_f$. Let us consider $\mathsf{cs}_j = \gamma_{j-1} \underset{\mathcal{T}}{\rightarrow} \gamma_j$ the first computing step of $\gamma_i \underset{\mathcal{T}}{\rightarrow} \cdots \underset{\mathcal{T}}{\rightarrow} \gamma_{f-1} \underset{\mathcal{T}}{\rightarrow} \gamma_f$ in which v executes $\mathbb{R}_{wait}$.

We have $\mathsf{da}_v^{\gamma_i} = 2D+2$, and only $\mathbb{R}_{wait}$ can decrease da_v, so $\mathsf{da}_v^{\gamma_{j-1}} = 2D+2$. Since v executes $\mathbb{R}_{wait}$ in cs_j, we also have $\mathsf{UnisonOnly}^{\gamma_{j-1}}(v)$. By construction of Algorithm 1, v executes $\mathbb{R}_{end}$ in cs_j, which is contradictory with the definition of γ_s. Thus, v does not execute any rule of $\mathbb{R}_{cvg}$ in ϵ_s.

Let us now prove that v executes an N-sequence of length at least $2D+1$ in ϵ_s. The only rule of $\mathcal{R}^S_{\text{proper}}$ executed by v along ϵ_s is $\mathbb{R}_{cpt}$, because $\mathsf{req}_v = \mathsf{wk}$ and because, by definition, $\mathbb{R}_{stop}$ and $\mathbb{R}_{end}$ are not executed.

During the ϵ-subexecution ϵ_s, v executes $\mathbb{R}_{cpt}$ exactly $2D+1$ times, since $\mathsf{dt}_v^{\gamma_s} = 2D+1$ and $\mathsf{dt}_v^{\gamma_{f-1}} = 0$. By definition of the algorithm and guards, in a step where v executes $\mathbb{R}_{cpt}$, v also executes $\mathbb{R}_{wait}$. Hence v also executes $\mathbb{R}_{wait}$ at least $2D+1$ times during ϵ_s. Since

v does not execute any rule of $\mathbb{R}_{cvg}$ in ϵ_s, those $2D+1$ executions of $\mathbb{R}_{wait}$ constitute an N-sequence of length at least $2D+1$. ■

Let us introduce *anchors*, that are nodes which execute a convergence action (and thus reset da at its maximal value) at their first activation during some execution. Anchors are natural candidates to prove point 1 of Proposition 4.1.

Definition 4.11 (*Anchor of an execution*)

Let $u \in V$ and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \cdots)$ be an execution. We say that u is an anchor of ϵ if during ϵ, u executes a rule that is not $\mathbb{R}_{start}$, and if the first rule different from $\mathbb{R}_{start}$ executed by u is a rule of $\mathbb{R}_{cvg}$.

We now introduce Theorem 4.5 that establishes that if an execution does not respect Condition 3-ii of Specification 4.2 then there exists an anchor of that execution.

Theorem 4.5 is a direct consequence of Lemmas 4.12 and 4.13.

Theorem 4.5

Let v be a node and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\rightarrow} \gamma_0$. If $\gamma_s \notin \Gamma_{\mathsf{cvg}}$, then there exists a node u that is an anchor of ϵ_s.

We first establish by contradiction that, if ϵ is an execution that starts in a configuration where $\mathcal{A}$ or $\mathcal{U}$ has not stabilized, and such that all nodes are activated, then at least one node executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during ϵ. This is formally stated in Lemma 4.11.

Lemma 4.11

Let $\epsilon = \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots$ be a finite or infinite execution such that all the nodes of V execute at least once a rule that is not $\mathbb{R}_{start}$ in ϵ. If $\gamma_0 \notin \Gamma_{\mathsf{cvg}}$, then there exists $u \in V$ that executes a rule of $\mathbb{R}_{cvg}$ in ϵ.

Proof: Since $\gamma_0 \notin \Gamma_{\mathsf{cvg}}$, then $\exists u \in V : \mathsf{Act}^{\gamma_0}(R_{\mathcal{A}}, u) \vee \mathsf{Act}^{\gamma_0}(R_{\mathcal{U}_C}, u)$.

Case 1: $\mathsf{Act}^{\gamma_0}(R_{\mathcal{A}}, u)$.

Let $cs_u = \gamma_u \underset{\tau}{\rightarrow} \gamma'_u$ be the first computing step of ϵ in which u is activated. If in cs_u, u executes a rule of $\mathbb{R}_{cvg}$ then our proof is complete. Otherwise, it means that in $\gamma_u, \neg\mathsf{Act}(R_{\mathcal{A}}, u)$. This is possible only if one neighbor u' of u updates $\mathtt{state}_{u'|\mathcal{A}}$ before γ_u, and thus u' executes a rule of $\mathbb{R}_{cvg}$ in ϵ.

Case 2: $\mathsf{Act}^{\gamma_0}(R_{\mathcal{U}_C}, u)$.

This means that $\neg P^{\gamma_0}(u)$. Thus there exists $u' \in N_u$ such that $\mathsf{clock}_{u'}^{\gamma_0} \notin \{\mathsf{clock}_u^{\gamma_0} - 1, \mathsf{clock}_u^{\gamma_0}, \mathsf{clock}_u^{\gamma_0} + 1\}$. Let $cs_u = \gamma_u \underset{\tau}{\rightarrow} \gamma'_u$ be the first computing step of ϵ in which u executes a rule that is not $\mathbb{R}_{start}$.

If in cs_u, u executes a rule of $\mathbb{R}_{cvg}$ then our proof is complete. Otherwise, it means that in $P^{\gamma_u}(u)$. This is possible only if before γ_u, u' executes a rule that updates $\mathtt{state}_{u'|\mathcal{U}}$. Let $cs_{u'} = \gamma_{u'} \underset{\tau}{\rightarrow} \gamma'_{u'}$ be the first activation of u' that updates $\mathtt{state}_{u'|\mathcal{U}}$. By construction, $\neg P^{\gamma_{u'}}(u')$ since the clocks of u and u' are unsynchronized. Consequently, in $cs_{u'}$, u' executes a rule of $\mathbb{R}_{cvg}$. ■

We now establish that if an execution does not respect Condition 3-ii of Specification 4.2 then there exists a node that executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during a final descent ϵ_s. Once again, we reason by contradiction, and observe that we can use Lemma 4.10 and Corollary 4.1. This is stated in Lemma 4.12.

Lemma 4.12

Let v be a node and let $\epsilon = (\gamma \underset{\mathcal{T}}{\rightarrow} \gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\mathcal{T}}{\rightarrow} \gamma_0$. If $\gamma_s \notin \Gamma_{\mathsf{cvg}}$, then there exists a node u that executes a rule of $\mathbb{R}_{cvg}$ in ϵ_s.

Proof: Let us reason by contradiction and suppose that no node execute any rule of $\mathbb{R}_{cvg}$ in ϵ_s. Let us prove that under such circumstances, all nodes execute a rule that is not $\mathbb{R}_{start}$ in ϵ_s. Let us consider any $w \in V$, and let us consider $(v = v_0)v_1 \cdots v_{k-1}(v_k = w)$ a path of length $k \leq D$.

According to Lemma 4.10, v executes an N-sequence of length at least $2D+1$ in ϵ_s, which means that v is a causal pyramid of length $2D+1$. Since we supposed that no node executed rules of $\mathbb{R}_{cvg}$ in ϵ_s, we can apply Corollary 4.1: $v_0v_1 \cdots v_{k-1}v_k$ is a causal pyramid of length $2D+1$. In particular, v_k executes $\mathbb{R}_{wait}$ in ϵ_s.

We can therefore apply Lemma 4.11, which ensures that there exists a node u that executes a rule of $\mathbb{R}_{cvg}$ in ϵ_s. ■

Finally, we prove that if there exists a node that executes a convergence rule for $\mathcal{A}$ of $\mathcal{U}$ during a final descent ϵ_s, there exists an anchor of that execution. Indeed, consider cs_a the first computing step of ϵ_s where a node executes a convergence rule, and consider u a node that executes a convergence rule during cs_a. Then, necessarily, the first activation of u during ϵ_s happens at cs_a. This is formally stated in Lemma 4.13.

Lemma 4.13

Let $\epsilon = \gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots$ be a finite or infinite execution such that there exists one node w that executes a rule of $\mathbb{R}_{cvg}$ in ϵ. Then there exists $u \in V$ that is an anchor of ϵ.

Proof: Case 1: in ϵ, w executes a rule of $\mathbb{R}_{cvg}$ that updates $\texttt{state}_{w|\mathcal{A}}$.

Let us consider $cs_a = \gamma_a \underset{\mathcal{T}}{\rightarrow} \gamma'_a$ the first computing step of ϵ in which one node u executes a rule that updates $\texttt{state}_{u|\mathcal{A}}$, and let us consider one such u. By definition, $\forall v \in V, \texttt{state}_{v|\mathcal{A}}$ remains constant during $\epsilon_a = \gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots \underset{\mathcal{T}}{\rightarrow} \gamma_a$. As a consequence, $\mathsf{Act}(R_{\mathcal{A}}, u)$ remains constant during ϵ_a too, and thus is evaluated to `true` all along ϵ_a since in cs_a, u updates $\texttt{state}_{u|\mathcal{A}}$. Thus, any activation of u during ϵ_a implies that u updates $\texttt{state}_{u|\mathcal{A}}$. This means that cs_a is the first activation of u in ϵ.

Case 2: in ϵ, w executes a rule of $\mathbb{R}_{cvg}$ that simulates a rule of $R_{\mathcal{U}_C}$.

Let $cs = \gamma_a \underset{\mathcal{T}}{\rightarrow} \gamma'_a$ be the first computing step of ϵ in which one node u executes a rule of $\mathbb{R}_{cvg}$ that simulates a rule of $R_{\mathcal{U}_C}$, and let us consider one such u. In $\gamma_a, \neg P(u)$. Recall that normal and transparent rules cannot invalidate $P(v)$, on any node $v \in V$. Thus, since before γ_a, no node executes rules of $R_{\mathcal{U}_C}$, and in $\gamma_a, \neg P(u)$, we have $\neg P(u)$ all along $\epsilon_a = \gamma_0 \underset{\mathcal{T}}{\rightarrow} \gamma_1 \underset{\mathcal{T}}{\rightarrow} \cdots \underset{\mathcal{T}}{\rightarrow} \gamma_a$. Therefore, the first activation of u different from $\mathbb{R}_{start}$ during ϵ_a is a rule of $\mathbb{R}_{cvg}$. ■

The combination of that last result and of the previous one terminates the proof of Theorem 4.5.

Theorem 4.6 is a consequence of Theorem 4.5. We show that if there exists an anchor of ϵ, then a contradiction is raised, thus the premisses of Theorem 4.5 do not hold. In the proof, we consider a causal pyramid of maximal length with origin v that ends near a node u, such that u executes a convergence rule before the action of the last node of the pyramid. We then prove that variable `da` spreads down the pyramid from u to v, which leads to the conclusion that v executes $\mathbb{R}_{end}$ during ϵ_s, raising a contradiction.

Theorem 4.6

Let v be a node, and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \gamma_1 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that v executes $\mathbb{R}_{start}$ in $\gamma \underset{\tau}{\rightarrow} \gamma_0$. We have: $\gamma_s \in \Gamma_{\mathsf{cvg}}$.

Proof: Let us reason by contradiction and suppose that $\gamma_s \notin \Gamma_{\mathsf{cvg}}$. According to Theorem 4.5, there exists a node u such that in ϵ_s, u executes a rule of $\mathbb{R}_{cvg}$ before any execution of $\mathbb{R}_{wait}$. Let us consider a path p of length $k \leq D$ between v and u: $p = (v = v_0 v_1 \cdots v_k = u)$. According to Lemma 4.10, v_0 is a causal pyramid of length $2D+1$.

Case 1: $v_0 v_1 \cdots v_k$ is not a causal pyramid of length $2D+1$. Let q be such that $v_0 v_1 \cdots v_q$ is a causal pyramid of length $2D+1$ and $v_0 v_1 \cdots v_q v_{q+1}$ is not a causal pyramid of length $2D+1$. According to Theorem 4.1, v_{q+1} executes $\mathbb{R}_{cvg}^{\mathcal{U}_C}$ during $(t_q^q, t_{2D-(q-1)}^q)$. Furthermore, by definition there exists $(v_q, t_{2D-q}^q) \preceq^N \cdots \preceq^N (v_0, t_{2D}^0)$.

Case 2: $v_0 v_1 \cdots v_k$ is a causal pyramid of length $2D+1$. Let $q = k-1$. By definition there exists $(v_k, t_{(2D)-k}^k) \preceq^N \cdots \preceq^N (v_0, t_{2D}^0)$.

In both cases, we have $(v_q, t_{2D-q}^q) \preceq^N \cdots \preceq^N (v_0, t_{2D}^0)$ and v_{q+1} executes a rule of $\mathbb{R}_{cvg}$ before (v_q, t_{2D-q}^q). We will now prove that there is a contradiction. The fact that one node near the pyramid executes a convergence rule before a dependance relation path $\preceq^N$ necessarily implies that v will receive a non-zero value for da before the end of its N-sequence, which will force it to execute $\mathbb{R}_{end}$. This is contradictory with the definition of final descent.

Let us consider $\gamma_{t'_{q+1}} \underset{\tau}{\rightarrow} \gamma'_{t'_{q+1}}$ the first computing step in which v_{q+1} execute a rule of $\mathbb{R}_{cvg}$ in ϵ_s. Remark that in $\gamma'_{t'_{q+1}}, \mathsf{da}_{v_{q+1}} = 2D+2$.

Let us now define, $t'_i, \forall i \leq q$, such that $\gamma_{t'_i} \underset{\tau}{\rightarrow} \gamma'_{t'_i}$ is the first computing step of $\gamma'_{t'_{i+1}} \underset{\tau}{\rightarrow} \cdots$ in which v_i executes $\mathbb{R}_{wait}$. Such integers exist since $(v_k, t_{(2D)-k}^k) \preceq^N \cdots \preceq^N (v_0, t_{2D}^0)$ exists, and we have $\forall i, t'_i \leq t_{2D-i}^i$.

Let us remark that da_{v_i} can decrease only when v_i executes $\mathbb{R}_{wait}$, and that according to Lemma 4.1, a node cannot execute more than twice $\mathbb{R}_{wait}$ or $\mathbb{R}_{cvg}^N$ before its neighbors execute $\mathbb{R}_{wait}$ or $\mathbb{R}_{cvg}^N$. Thus, in $\gamma_{t'_q}, \mathsf{da}_{v_{q+1}} \geq 2D$, so in $\gamma'_{t'_q}, \mathsf{da}_{v_q} \geq 2D-1$. By induction, we obtain that $\forall i$, in $\gamma'_{t'_i}, \mathsf{da}_{v_i} \geq 2D-1-2(q-i)$. Since $q \leq k-1 \leq D-1$, in $\gamma'_{t'_0}, \mathsf{da}_v \geq 2D-1-2q \geq 1$.

Let us now consider the first rule executed by v at time $t > t'_0$. As $\mathsf{da}_v^{\gamma_{f-1}} = 0$, $t < f$. This rule does not belong to $\mathbb{R}_{cvg}$ according to Lemma 4.10. As $\mathsf{req}_v^{\gamma_{t'_k}} = \mathsf{wk}$, this rule cannot be $\mathbb{R}_{start}$. Therefore, v executes $\mathbb{R}_{wait}$ at time t, and since $\mathsf{da}_v^{\gamma_t} \neq 0$ and $\mathsf{req}_v^{\gamma_t} = \mathsf{wk}$, v also executes $\mathbb{R}_{end}$ at time t. There is a contradiction with the definition of γ_s. ■

We can now use Theorem 4.6 to prove that maximal executions of $\mathcal{T}$ satisfy the last condition of Specification 4.2: Condition 3-ii

Theorem 4.7

Let $\epsilon = \gamma_0 \underset{\tau}{\rightarrow} \cdots$ be a maximal execution such that $\exists v \in V, t \geq 0 : \mathsf{req}_v^{\gamma_t} = \mathsf{on}$, and let $t' > t$ such that $\mathsf{req}_v^{\gamma_{t'}} = \mathsf{off}$.

Then $\mathcal{A}$ has terminated in $\gamma_{t'}$.

Proof: Let us first make the same reasoning as in the proof of Theorem 4.4: according to Lemma 4.6, and since $\mathcal{U}$ is a unison algorithm, v is activated at least once after γ_t, and when first activated it executes $\mathbb{R}_{start}$. Let us call $\gamma_{act} \underset{\tau}{\rightarrow} \gamma_{act+1}$ that computing step.

Let us now consider the subexecution $\epsilon' = \gamma_{act} \underset{\tau}{\rightarrow} \gamma_{act+1} \underset{\tau}{\rightarrow} \cdots$.

By definition, t' occurs after γ_{act} in ϵ, since $\mathsf{req}_v = \mathsf{on}$ as long as v does not execute $\mathbb{R}_{start}$. More precisely, the first t' such that $\mathsf{req}_v^{\gamma_{t'}} = \mathsf{off}$ is γ_f.

The execution ϵ' satisfies the premises of Theorem 4.6 and thus we have $\gamma_s \in \Gamma_{\mathsf{cvg}}$. In particular, $\mathcal{A}$ has terminated in γ_s. Since $\gamma_{t'}$ occurs after γ_s, and since termination of $\mathcal{A}$ is a closed property, this guarantees that $\mathcal{A}$ has terminated in $\gamma_{t'}$. ■

4.5.4 Time complexity

Definition 4.12 (*Full Round*)

Recall that the sluggishness of $\mathcal{U}$ is the maximal number of transparent rules a node can execute between two normal rules, in a stabilized execution. A full round of an execution is defined as $1 + \mathcal{S}(\mathcal{U})$ rounds.

The notion of full round is the suitable notion to evaluate the time of response of our snap-stabilizing algorithm, since Algorithm $\mathcal{U}$ is for us a black box. Recall that, for unison algorithms presented in [CFG92, BPV04, DJ19, EK21], the notion of full round is identical to the more classical notion of round.

Theorem 4.8

Let v be a node, and let $\epsilon = (\gamma \underset{\tau}{\rightarrow} \gamma_0 \underset{\tau}{\rightarrow} \cdots)$ be a maximal execution such that $\gamma \in \Gamma_{\mathsf{cvg}}$ and in $\gamma \underset{\tau}{\rightarrow} \gamma_0$, v executes $\mathbb{R}_{start}$. Then f occurs in $O(D)$ full rounds after γ.

Proof: Theorem 4.2 guarantees that all the rounds are finite.

During one full round, all nodes u such that $\forall w \in N_u, \mathsf{clock}_w \in \{\mathsf{clock}_u, \mathsf{clock}_u + 1\}$ are activated and execute rule $\mathbb{R}_{wait}$. These activations imply that $\min_{w \in V} \mathsf{clock}_w$ increases by at least one each full round. Since, at any moment, the maximal difference between the clocks of two nodes is D, we obtain that $\forall k \in \mathbb{N}$, during $D + k$ full rounds all the nodes are activated at least k times. After $D + 1$ full rounds, all the nodes are activated, and the following property holds for any node u in any configuration: $\mathsf{da}_u = \max_{w \in V} \mathsf{da}_w \wedge \mathsf{da}_u > 0 \Rightarrow \mathsf{UnisonOnly}(w)$. In other words, the nodes with maximal value of da are enabled for the rule $\mathbb{R}_{wait}$. This implies that the maximal value of da decreases by at least one in each full round. This ensures that after at most $3D + 3$ full rounds, the system is in Γ_{da}. Moreover after $D + 2D + 2$ more full rounds, all the nodes are activated at least $2D + 2$ times, so v executes $\mathbb{R}_{cpt}$ $2D + 1$ times, after which it executes rule $\mathbb{R}_{stop}$.

We established that f occurs in at most $6D + 5 = O(D)$ full rounds after the execution of $\mathbb{R}_{start}$ by v. ■

4.6 Conclusion

In this chapter, we introduced a generic, deterministic, snap-stabilizing, silent algorithm that solves Termination Detection in asynchronous networks. Our solution works assuming an unfair scheduler. It has the nice feature of working in anonymous networks, but requires that each node knows (an upper bound on) the network diameter D. The space complexity of our solution is $O(\log D)$ bits per node, and provides an answer in $O(\max(k, k', D))$ rounds, where k and k' are the stabilization time complexities of the observed and the unison algorithms, respectively. We have endeavored to provide a generic algorithm that works with any self-stabilizing unison algorithm in the literature, *e.g.*, [CFG92, BPV04, DJ19, EK21].

Chapter 5

Optimal Self-stabilizing Token Circulation in DODAGs

It is a love based on giving and receiving
As well as having and sharing
And the love that they give and have is shared and received
And through this having and giving and sharing and receiving
We too can share and love and have and receive

Reverend Tribbiani

Contents

5.1 Introduction

In this chapter, we are interested in the problem of *Token Circulation* in a distributed system. The objective is to maintain one single token circulating in the network, from one node to another, the token fairly visiting every node infinitely often. It is one very common way to achieve global mutual exclusion, a fundamental problem in distributed systems. This problem inherently requires that the token holder can designate a unique neighboring node to pass the token to. This single task is actually, challenging as soon as we consider wireless networks, in which each transmitted message is identically received by all of the transmitter's neighbors.

The contribution of this chapter is fourfold. First, we present the first token circulation algorithm for rooted wireless networks that uses only $O(\log\log n)$ bits per node for communications. This is an exponential improvement compared to the classical addressing. Second, our algorithm assumes a Destination-Oriented Directed Acyclic Graph (DODAG) spanning the network. DODAGs are very desirable in wireless networks because, conversely to rooted spanning trees, they allow multiple communication paths and do not require that nodes distinguish one unique parent. Third, our algorithm has the very desirable property of being self-stabilizing, *i.e.*, starting from a configuration where zero or more than one token circulate in the network, the system is guaranteed to eventually behave correctly, *i.e.* a unique token eventually fairly visits every node infinitely often. Finally, we show that our algorithm is optimal in terms of space complexity. Meaning that, for every n-node wireless network, no token circulation algorithm can use less than $\Omega(\log\log n)$ bits of memory per node, even given a rooted spanning tree and even without considering self-stabilization.

5.1.1 Motivation

Token Circulation, also referred to as *token passing* is a fundamental problem that consists in guaranteeing that a single token circulates from one node to another, the token fairly visiting every node infinitely often. It captures the objective of perpetually and fairly allocating a resource to the nodes of a distributed system, while insuring mutual exclusion. We are interested in *self-stabilizing* [Dij74] token circulation in distributed systems. A typical example of such systems is the celebrated Dijkstra's stabilizing token ring algorithm, which guarantees that a unique token is circulating among the n nodes of a ring-shaped network.

A self-stabilizing algorithm must return the system to a correct behavior whatever its initial configuration. In the context of token circulation, examples of initial arbitrary system configurations are scenarios in which no token exists in the network, or several tokens exist in the network. To satisfy the requirement of self-stabilization, a distributed token circulation algorithm must return in finite time the system in a configuration that contains exactly one token. It must actually do more than that, as the presence of a unique token is not sufficient. Indeed, the algorithm must also guarantee that, eventually, this unique token is circulating fairly among the nodes. Following the seminal work of Dijkstra [Dij74], token circulation has been widely investigated deterministically in the context of self-stabilization, on rings [BP89, GH96], on acyclic networks [BGW89, Gho93, PV07], and on arbitrary shaped networks [HC93, DIM93, JB95, DJPV00, CDPV06, PV07, CDV09], to quote only a few.

All the above algorithms were written in the *state model* [Dij74], where the communications between the nodes are modeled by the ability for a node to atomically read the content of its neighbors registers, and update its own registers. Hence, they all assume wired distributed systems, *i.e.* the port-known state model $\mathcal{S}_{\mathsf{PK}}$ in which nodes can address a message to exactly one of their neighbors. This hypothesis is particularly helpful for the problem of Token Circulation, where sending the token to exactly one neighbor is crucial to maintain the unicity of the token.

In recent years wireless network technology has gained tremendous importance. It not only more and more replaces so far 'wired' network installations, but also gives rise to new applications and with them, new tech and soft stacks enabling specific communications. Wireless networks often involve devices with small memory and/or battery. Therefore, designing memory efficient algorithms for token circulation in wireless networks is essential.

5.1.2 Related Work

The port-known state model $\mathcal{S}_{\mathsf{PK}}$ has the desirable advantage to achieve algorithm design that does not necessarily need global information and to facilitate maintaining a global covering structure, as tree networks (by maintaining a pointer parent and, if necessary, some pointers descendants). Since Dijkstra's seminal work on ring-shape networks, this model has been widely used for various problems. Refer to [Dol00, ADDP19] for more algorithms.

Note that some of the token circulation algorithms written in $\mathcal{S}_{\mathsf{PK}}$ are interested in optimizing the space complexity. They achieve $O(\log \delta_u)$ bits per node (recall that δ_u is the degree of u) [Dij74, BP89, GH96, BGW89, Gho93] on rings or chains, [PV07] on trees, and [DJPV00] on arbitrary networks. Except [BP89] that assumes a uniform prime size ring, all the above algorithms require a root node, *i.e.*, a node with a particular local algorithm with respect to the other nodes.

Unlike $\mathcal{S}_{\mathsf{PK}}$, the port-unknown state model $\mathcal{S}_{\mathsf{PU}}$ introduces no local assumption for a node to select one of its neighbors, except its identifier. This means that given a pair of neighbors u and v, v has no way to know the port number of u that links u to v. This model fits more to wireless networks than the first variant. By contrast with the first variant where a node has the possibility to select a specific neighbor with the use of a pointer, in this model, with local information only, a node has a priori no way to know whether it is the specific recipient of a message sent by one of its neighbors. The lack of discernment between neighbors makes the design of algorithms much more difficult, which often require the knowledge of extra information used by each local algorithm, as the use of node's identifiers or the construction of an underlying vertex coloring. Token circulation algorithms for $\mathcal{S}_{\mathsf{PU}}$ are proposed in [BDF20, BBD18, BPBRT09]. All these articles work on tree topologies and require $O(\log n)$ bits.

Still in $\mathcal{S}_{\mathsf{PU}}$, it has been shown that many standard "one-shot" tasks like leader election and spanning tree construction can be constructed in a self-stabilizing manner with memory $O(\log\log n + \log\Delta)$ bits in any n-node network with maximum degree Δ [BT18, BT20]. This indicates that some tasks can be achieved in $\mathcal{S}_{\mathsf{PU}}$ with only sublogarithmic memory. As the results established in Chapter 3 suggest, it is much harder to reach $o(\log\log n)$ memory.

5.1.3 Contributions

In this chapter, we address the problem of token circulation in $\mathcal{S}_{\mathsf{PU}}$ on a particular class of graphs, *Destination-Oriented Directed Acyclic Graphs*, DODAG for short [Mar65]. Recall that DODAGs are bi-directional graphs in which each edge is given an orientation, so that the resulting graph contains no loop, and has exactly one root: a node with only incoming edges from its neighbors. Our interest for DODAGs is multifold. First, it provides the nodes with a weak notion of sense-of-direction [FMS03]. Second, it relaxes the notion of rooted spanning tree by allowing the nodes to have more than a single parent, which increases the difficulty of token circulation. Third, DODAGs avoid loops, and provide the network with a single sink, which fit with the standard semi-uniform model [Dol00] for self-stabilization — every node with the same degree executes the same program, except one. Last but not least, DODAGs are the basic structures used in practice by RPL (Routing Protocol for Low power and lossy networks [WTB+12]), which is the standard routing protocol for IPv6-based multi-hop wireless sensor networks [TR21].

We propose a self-stabilizing algorithm for the token circulation problem on DODAGs. Our algorithm works under the strongest adversary, the *unfair scheduler*, which can activate any subset of the enabled nodes at each step, without any fairness requirements. Our algorithm requires only $\Theta(\log\log n)$ bits per node, which is similar to the complexity of [BT18, BT20], without the dependence on the degree of the graph Δ. We prove that this space complexity is optimal for the token circulation in $\mathcal{S}_{\mathsf{PU}}$, even under a least challenging scheduler, and even on simpler classes of graphs, such as trees.

To obtain such complexity, we use a similar technique than [BT18, BT20] for the communication of identifiers, which is basically bit-by-bit communication. Although it is possible to communicate an identifier by small pieces, it remains impossible for nodes to store the different identifiers of their neighbors, and to directly address one of them. Therefore, we designed an additive mechanism that relies on that bit-by-bit communication to allow the token holder to give the token to one of its children, and to somehow remember it did once, so that when its child sends it back, it can give the token to one other child, and therefore achieve the fairness property.

Chapter Outline The rest of the chapter is organized as follows. We first introduce specific notations, definitions, and formal structures on which our work relies. Then in Section 5.3 we present our token circulation algorithm. Section 5.4 contains the proofs of the validity and optimality of our algorithm. We prove in particular that our algorithm is a self-stabilizing algorithm for the fair token circulation problem. We make some concluding remarks in Section 5.5.

5.2 Model and Definitions

5.2.1 General Model

Our token circulation algorithm is designed for DODAGs (see Section 5.2.2), which are graphs that are inherently semi-uniform. We consider the port-unknown state model $\mathcal{S}_{\mathsf{PU}}$, and suppose that nodes are given unique identifiers taken in $[1, n^c]$. Our algorithm does not require that all identifiers have the same length, but we make the reasonable hypothesis that all identifiers have at least one bit, even a 0 bit. Finally, our algorithm works under the most challenging adversary, the unfair distributed scheduler, which activates any subset of the enabled nodes at each step, without any fairness requirement.

5.2.2 DODAGs

In this thesis, we only consider graphs in which the communication links are not oriented, in the sense that the communication is bi-directional. Some networks are nevertheless structured, sometimes hierarchically, DNS servers, routers to quote a few. This hierarchy is materialized by the fact that nodes can distinguish messages sent by neighbors that dominate them, from messages sent by neighbors that they dominate.

In tree networks, nodes different from the root have exactly one node which dominate them. This hypothesis is very strong, and in particular it does not allow any redundancy, which could be useful if faults occur. We consider structures that do not require the unicity of the parent.

Two characteristics are desirable for such organized network, being loop-free, and being rooted. An oriented graph is loop-free if there does not exist any path along the directed edges from one node to itself, and it is rooted if there exists one node (necessarily unique by loop-free property) that can reach any other node by a path along the directed edges.

Similarly as Dijkstra's stabilizing token ring algorithm [Dij74] assumes a consistent notion of left and right in the ring network, we assume that each edge $\{u, v\} \in E$ is provided with a direction, from u to v, or from v to u, so that the resulting directed graph form a *Destination-Oriented Directed Acyclic Graph* [Mar65] (DODAG), *i.e.*, a Directed Acyclic Graph (DAG) with a unique root (see Fig. 5.1).

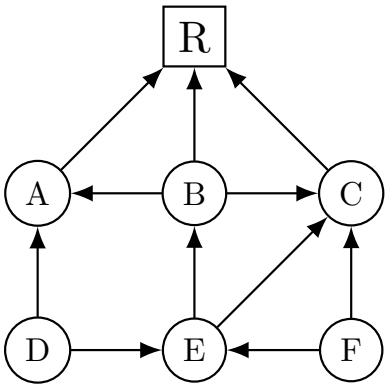

Figure 5.1: A destination-oriented directed acyclic graph ($\mathtt{D}^o$).

Let us be more formal:

Definition 5.1 (*Destination-Oriented Directed Acyclic Graph, DODAG*)

A DODAG is a tuple $D = (G, \rightarrow)$ where $G = (V, E)$ is a graph and $\rightarrow$ is a relation between the nodes of G which:

- *covers the edges of G: $(u \rightarrow v \vee v \rightarrow u) \iff \{u, v\} \in E$,*
- *is acyclic: there does not exist any path from one node v to itself w.r.t. $\rightarrow$, and*
- *is rooted: there exists one node $r \in V$ such that $\forall v \in V, v \rightarrow^* r$.*

The nodes of D are V, and the edges of D are $\rightarrow$.
By the acyclic property, such a node r is unique, and is called the root *of D. Also by the acyclic property, for any edge $\{u, v\} \in E$, we have either $u \rightarrow v$ or $v \rightarrow u$ but not both.*

In practice, the relation $\rightarrow$ is implemented at the layer of the local port numbers of the nodes. A DODAG is a graph in which the port numbers can be split into two disjoint sets $\texttt{port}^+$ and $\texttt{port}^-$, and the relation $\rightarrow$ is defined by $u \rightarrow v \iff \texttt{port}_u(v) \in \texttt{port}^+$.

In the latter, we often need to consider, on one node v, the set of its neighbors that precede it, and the set of neighbors that it precedes. We call those sets the parents and the children of the node, by analogy with the lexical field relative to trees.

Note that contrary to what is possible in trees, nodes may have several parents in DODAGs.

Definition 5.2 (*Parental Relationships in a DODAG*)

Let $D = (G, \rightarrow)$ be a DODAG.
We call parents *of v in D the set $\mathcal{P}(v) = \{u \in N_v : v \rightarrow u\}$.*
We call children *of v in D the set $\mathcal{C}(v) = \{u \in N_v : u \rightarrow v\}$.*

Remark 5.1

The information of the set of the parents of each node is equivalent to the information of the relation $\rightarrow$. In practice, we define a DODAG by the parental relationship it induces more than by expliciting the relation $\rightarrow$.

It will be useful in the latter to reason on sub-structures of the DODAG in which our algorithm is executed. In most cases, the sub-structures considered also have the desirable property of being DODAGs. A sub-DODAG of D is any DODAG whose nodes and edges are a subset of those of D.

Definition 5.3 (*Sub-DODAG of G, Anchor*)

A tuple $D' = (G', \rightarrow')$ is a sub-DODAG of $D = (G, \rightarrow)$ if

- *D' is a substructure of D: $(V' \subseteq V) \wedge (\rightarrow' \subseteq \rightarrow) \wedge (\rightarrow \subseteq V' \times V')$.*
- *D' is a DODAG*

Remark that the root of D' may not be the same than the root of D. To avoid being misleading, we call the root of a sub-DODAG an anchor.

Remark 5.2

Following what was stated in Remark 5.1, we will not use relation $\rightarrow'$ to define sub-DODAGs in practice, and rather use the underlying set of parents and children, $\mathcal{P}_{D'}$ and $\mathcal{C}_{D'}$.

One typical example of sub-DODAG is the restriction of the DODAG to the nodes which can reach one particular node (other than the root) in $(G, \rightarrow)$.

Definition 5.4 (G_a *DODAG under* a)

Let $(G, \rightarrow)$ be a DODAG, and let $a \in V$.

We denote by V_a the nodes that can reach a by $\rightarrow$: $V_a = \{v \in V : v \rightarrow^* a\}$. We denote by E_a and $\rightarrow_a$ the restrictions of E and $\rightarrow$ to nodes of V_a: $E_a = E \cap \mathcal{P}_2(V_a)$ and $\rightarrow_a = \rightarrow \cap (V_a \times V_a)$.

We denote by $(G_a, \rightarrow_a)$, or simply G_a, and call the DODAG under a, the sub-DODAG $((V_a, E_a), \rightarrow_a)$.

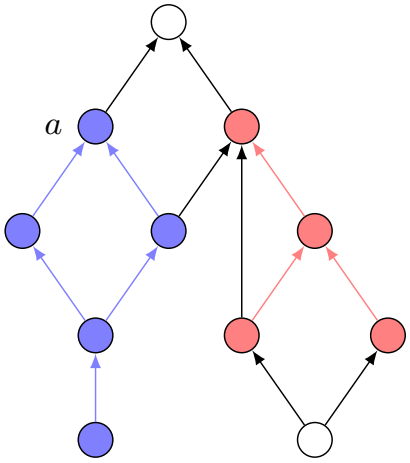

Figure 5.2: Example of a DODAG under the anchor a in blue, and of a sub-DODAG of G in red

Although DODAGs are the structure in which our algorithm and our proofs are established, it is often simpler to reason on linear structures than on DODAGs. In the latter, we will consider branches of DODAGs, which are descending paths from one node to one of its descendants.

Definition 5.5 (*Branch of a DODAG*)

We call branch of a DODAG $D = (G, \rightarrow)$ a path of nodes in D (which is a path of nodes in G w.r.t. $\rightarrow$). For the sake of readability, we denote branches by starting at the highest node. For example, if $w \rightarrow v \rightarrow u$ is a path in D, then we say that uvw is a branch of D.

We denote branches by the letter $\mathcal{B}$.

A branch $\mathcal{B} = v_0 v_1 \dots v_k$ of a DODAG D is maximal if v_0 is the root (or the anchor) of D, and if v_k does not have any children in D.

Given a sub-DODAG of D, it will be interesting to consider the longest branches it shares with D. These longest branches are called projections of the sub-DODAG on branches.

Definition 5.6 (*Projection of a sub-DODAG on a Branch*)

Let us consider a DODAG D, and a sub-DODAG of D with anchor a, D_a. Let $\mathcal{B} = v_1 v_2 \dots v_k$ be a maximal branch of G_a, the DODAG under a. In particular, $v_1 = a$.

We denote $D_a(\mathcal{B})$, and call the projection of D_a on $\mathcal{B}$ the longest branch of D_a that coincides with $\mathcal{B}$:

$D_a(\mathcal{B}) = v_1 v_2 \cdots v_j$ *where* $\forall i < j, v_{i+1} \in \mathcal{C}_{D_a}(v_i)$ *and* $v_{j+1} \notin \mathcal{C}_{D_a}(v_j)$ *(recall that* $v \in \mathcal{C}_{D_a}(u) \iff v \to_{Da'} u$*).*

Remark 5.3 (*DODAG* $\equiv \mathtt{D}^o$)

In the latter, we consider different types of DODAGs, depending on their purpose. For the sake of readability, we will use $\mathtt{D}^o$ *as an alias for the classical notion of DODAG, and* $\mathtt{CD}^o$, $\mathtt{WD}^o$, $\mathtt{FD}^o$ *as aliases for particular types of DODAG.*

5.2.3 Bit-by-Bit Communication of Identifier

In $\mathcal{S}_{\mathsf{PU}}$, nodes cannot communicate a message to one single neighbor without using the fact that identifiers are, at least locally, unique (this is formally proven in Section 5.4.4). In other words, nodes must communicate their identifier to their neighbors to achieve the circulation of one single token. Hence, the size of an identifier is $\Theta(\log n)$ bits, while we aim at designing a sub-logarithmic algorithm.

To reach a $\Theta(\log \log n)$ memory, [BT18] make the nodes communicate their identifier by smaller pieces. In this seminal paper, the authors used a function called $\mathtt{Bit}_v(i)$, which returns the position of the i^{th} most significant bit equal to 1 in $\mathtt{id}_v$.

We use a similar technique, but for the sake of simplicity, we slightly modified this function for this work, and send the identifier bit-by-bit, which does not change the asymptotic complexity. Essentially, nodes do not communicate their entire identifier at once, but when asked for, they send to their neighbors the value of their i-th bit (which is 0 or 1). Although the value of the bit is 1-bit long, we must take into account the fact that the position of the bit which is asked, i, is now part of the message.

This position variable i takes value between 1 and the maximal length of an identifier, which is in $\Theta(\log n)$. Therefore, it is encoded on $\Theta(\log \log n)$ bits. To simplify, we suppose that all nodes are given a local function $\mathtt{Bit}$ that associates to each position, the value of the bit at this position in their identifier. If the position asked is bigger than the length of their identifier, then the function simply returns $\perp$. Suppose for example that node v has identifier $\mathtt{id}_v = 1011$. Then we define the function $\mathtt{Bit}_v$ by:

$$\mathtt{Bit}_v(i) := \begin{cases} 1 & \text{if} \quad \text{i=1} \\ 0 & \text{if} \quad \text{i=2} \\ 1 & \text{if} \quad \text{i=3} \\ 1 & \text{if} \quad \text{i=4} \\ \perp & \text{if} \quad i > 4 \end{cases}$$

Nodes keep storing their own identifier in their immutable memory, but what is communicated to the neighbors through the immutable memory is the tuple (i, bit), which requires $\Theta(\log \log n)$ bits. Remark that since the identifiers are globally unique, the different functions $\mathtt{Bit}$ are also globally unique, although they can coincide for some values of i.

5.2.4 Well-Founded Sets

Recall that an ordered set $(M, \preceq)$ is well-founded if there does not exist an infinite sequence of elements of M, $v_0 v_1 \ldots$ such that $\forall i \in \mathbb{N}, v_{i+1} \prec v_i$. Such relations are very desirable to prove termination of algorithms, or their convergence. In the latter, we consider an arbitrary well-founded set $(M, \preceq)$.

Given $(M, \preceq)$, we can define an order on tuples of elements of M, which is itself well-founded when restricted to sequences of length at most k, for any k.

Definition 5.7 (*Lexicographic Order*)

For any $k \geq 1$ we define the lexicographic order on M^k induced by $\preceq$, and denote $\preceq^k_{lex}$, by:

$$(u_1, \ldots, u_k) \prec^k_{lex} (v_1, \ldots, v_k) \iff \exists i \in [1, k] : \begin{cases} (u_1, \ldots, u_{i-1}) = (v_1, \ldots, v_{i-1}) \\ u_i \prec v_i \end{cases}$$

In our proof, we are sometimes led to compare sequences of elements of M which have variable length. We extend the definition of lexicographic order to sequences of arbitrary length, which corresponds to the alphabetical order. The resulting order is itself well-founded.

Definition 5.8 (*Alphabetical Order*)

The lexicographic order on (possibly empty) sequences of elements of M induced by $\preceq$, is denoted $\preceq_\alpha$ and defined by:

$$(u_1, \ldots, u_k) \prec_\alpha (v_1, \ldots, v_l) \iff \begin{cases} \exists i \leq \min k, l : (u_1, \ldots u_i) \prec^i_{lex} (v_1, \ldots v_i) \\ \vee \\ k < l \wedge (u_1, \ldots, u_k) = (v_1, \ldots, v_k) \end{cases}$$

5.2.5 Token Circulation

In this chapter, we consider the problem of fair token circulation, which requires that one unique token perpetually circulates through the network.

The part of the specification of the token circulation problem which corresponds to the existence and unicity of the token is not complicated to state, and is basically what we made in Chapter 3 when we gave a specification for the leader election problem. It boils down to defining a local predicate on nodes, which represents the fact that the token is held by that node, and guaranteeing that the predicate is evaluated to true on exactly one node in each configuration. This first predicate will be denoted by T, for token.

It is a bit harder to formally specify the fairness circulation property. For a token circulation to be fair, one could expect that all nodes have the token at the same rate. But the more a node has children, the more it will receive it back from its children before sending it to its other children. We must be more specific on what fairness is. We define a more restrictive local predicate, which represents the fact that the node has the token *and* is allowed to use it to access the resource, the service,... This second predicate will be denoted by R, for resource access.

Now that we have this second predicate, we only have to guarantee that, between two configurations in which one particular node has access to the resource, then all the other nodes also have it, and exactly once. In practice, the node that we consider to delimit fairness is the root. A sub-execution in which the root receives the token for access, then drops it, and then receives it back for access, is called a *round*, or a *circulation round*.

Definition 5.9 (*Round according to R*)

Let R be a local predicate, and let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an execution.
A sub-execution $\epsilon' = \gamma_i \to \cdots \to \gamma_j$ of ϵ is a round according to R if:

- $\neg R^{\gamma_{i-1}}(r)$
- $R^{\gamma_{j+1}}(r)$
- $\exists i \leq k < j$ *such that*
  - $\forall t \in [i,k], R^{\gamma_t}(r)$
  - $\forall t \in [k+1,j], \neg R^{\gamma_t}(r)$

Note that by construction, circulation rounds perfectly follow each other in any execution.

We say that a circulation round is fair if for any node v, there exists exactly one continuous sequence of configurations in which v has access to the resource.

Definition 5.10 (*Fair Round according to R*)

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an execution, and let $\epsilon' = \gamma_i \to \cdots \to \gamma_j$ be a round according to R.

ϵ' is fair if for any $v \in V$, there exist $i \leq t_1 < t_2 \leq j$ such that

- $\forall t \in [t_1, t_2], R^{\gamma_t}(v)$
- $\forall t \in [i,j] \setminus [t_1, t_2], \neg R^{\gamma_t}(v)$

We can now formally define the specification of Fair Token Circulation:

Specification 5.1 (*Token Circulation*)

$\mathcal{TC}$ (Token Circulation) is defined by the existence of two local predicates T and R such that $\neg T \Rightarrow \neg R$.

A configuration γ is correct if it contains one unique token

$$U_{\mathcal{TC}}(\gamma) \equiv \exists! v \in V : T^{\gamma}(v)$$

An execution $\epsilon = \gamma_0 \to \cdots$ is correct if

1. *All its configurations are correct: $\forall i \geq 0, U_{\mathcal{TC}}(\gamma_i)$.*
2. *ϵ is infinite and can be divided into an infinite number of rounds according to R.*
3. *All of these rounds are fair*

Definition 5.11 (*Circulation Rounds*)

In the context of Token Circulation, we call Circulation Round *a round according to the predicate R.*

5.3 Algorithm

The general idea of our token circulation algorithm is to implement a perpetual Depth-First traversal from the root. The root sends the token to its child with maximum identifier, which does the same, until a leaf is reached. Then, the leaf sends the token back to its parent, which visits its other children, until all of them are visited, then it sends it back to its own parent, and so on.

Nodes have to remember which of their children were already visited, in order to send the token to unvisited children, or to send it back to a parent. Classically, we implement that with a variable *color* which can take two values. Nodes only send the token to the nodes that do not have the same color as themselves, and as soon as one node receives the token from a parent, it changes its color, and therefore becomes *visited*.

5.3.1 Issues Relative to the Communication Model and Partial Solutions

The main difficulty to design a token circulation algorithm in $\mathcal{S}_{\mathsf{PU}}$ with $\Theta(\log\log n)$ bits per node is that nodes cannot store any pointer to one neighbor.

This is yet crucial to avoid duplicating the token when sending it to one child. As explained above, the presence of unique identifiers, and the bit-by-bit communication scheme will allow us to implement such a behavior. Figure 5.3 presents how a token may circulate from top to bottom in what we call a *triangle configuration*.

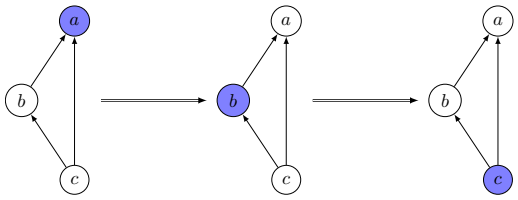

Figure 5.3: Scheme of the circulation of a token from top to bottom in a triangle, with $\mathtt{id}_b > \mathtt{id}_c$. The node holding the token is colored in blue.

But a second issue comes with the return of the token. Indeed, nodes cannot store a pointer to the parent from which they received the token. For fairness concerns, it is necessary that the parent that receives the token back from one node is the one that sent it to that node. Indeed, in the scheme presented in Figure 5.3, it might be that node b has other unvisited children after c, let us say d. If a receives the token directly from c, then it won't send it to b before the next circulation round, which is likely to be a simple repetition of what we just described. Therefore, d would never receive the token. We show on Figure 5.4 how this returning should be.

As we said previously, a node cannot remember from which of its parents it received the token. Therefore, from c's point of view, a and b are indistinguishable. This implies that the responsibility to not duplicate the token belongs to a and b. The idea is that each node that has sent the token to its children, and which has not received it back yet, keeps that information in its variables, meaning that it is involved in a non-terminated circulation. Considering this new information, a and b now differ in the sense that, from b's perspective, when c offers the token up, it does not have any other children involved in a token circulation, while from a's perspective, it also has b as a child which waits for the token. Therefore, only b will take the token when c offers it.

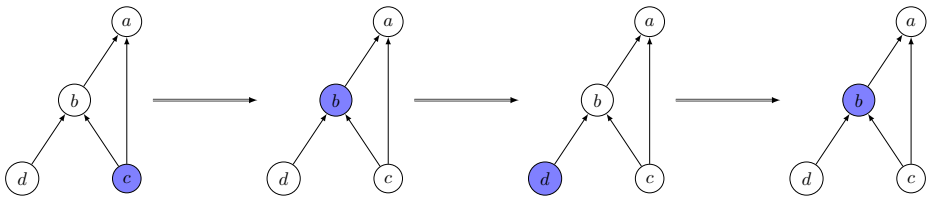

Figure 5.4: Scheme of the return of a token to the right parent in a triangle for fairness concerns.

5.3.2 General Ideas of our Algorithm

Implicit Repair as much as possible One very fundamental idea of the design of our algorithm is to repair the nodes as rarely as possible. One first reason is that the algorithm is already highly complex, due to the numerous tasks that nodes must fulfil to achieve fair token circulation. Any additional rule designed to repair an inconsistent situation would complicate the understanding of both the algorithm and its proof. One second reason is that for most cases, it in unnecessary to repair nodes, since if the algorithm is resilient enough, then the token will circulate in the network. For most cases, the simple presence of the token and the rules that allow fair executions will allow the network to eventually stabilize. There are few exceptions *i.e.* repairing rules that are necessary for convergence.

Fairness Property: Coloring the Nodes As we said above, we use a one-bit variable, referred to as *color*, to achieve the fairness requirement. The idea is that one node that has the token considers that its children with the same color already received it. Conversely, a node that has the same color as a parent offering the token will ignore it. Each time a node receives the token, it switches its color and becomes visited. Once the root receives the token from one child, and notices that all its children have its color, then it simply switches it color, and a new circulation round starts. This guarantees that one node can only receive the token once at each round.

In consistency with the previous paragraph, the color of the node is only checked from the child to the parent, and not the opposite. This means that if a parent v has a child u from the opposite color which offers the token to its ancestry, then v will accept the token, although it does not match a regular execution.

Even if we start in a configuration where the color of the nodes is inconsistent, after some circulation rounds, all the graph will eventually be reached by the token. Indeed, at each round, the color of the token switches and thus nodes that were ignored for being supposedly visited are now seen as unvisited, and therefore receive the token at once. This is depicted in Figure 5.5.

Choice of the Child: Identifier-Based Local Election To determine which of its children will receive the token, the parent progressively eliminates them, until only one remains. When it has exactly one child running for the token, it can finally declare that it gives the token. All the children that were eliminated ignore that message, and the one winner can take the token from its parent, and therefore no duplication of the token can occur.

The elimination process relies on the identifiers of the children, and on one additional variable which allows unvisited nodes to declare themselves as negotiating, or as losers of the negotiation. Step by step, the parent asks for the value of the i-th bit of the identifier of its children, and when all have answered, it reveals the biggest value it sees. All the nodes

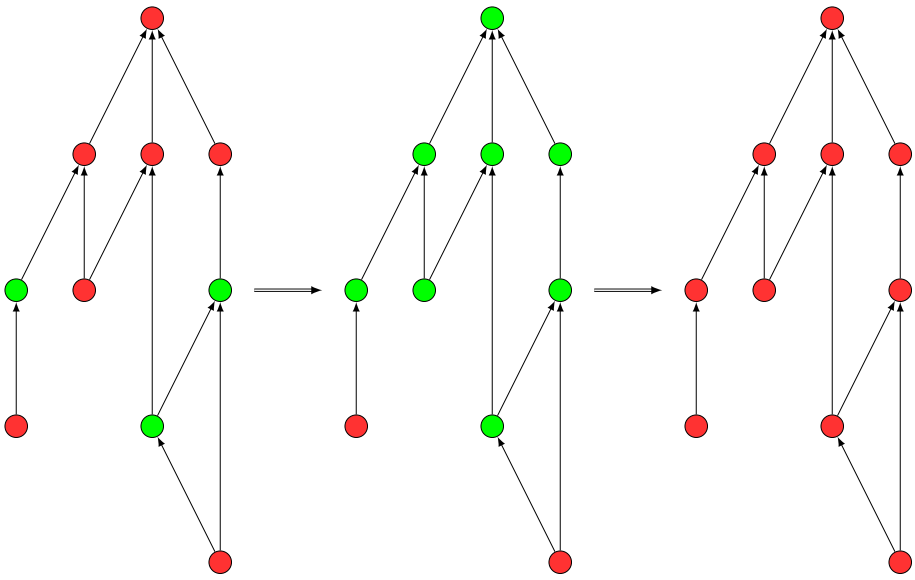

Figure 5.5: Increase of the area in which the token circulates starting from an arbitrary configuration. Shots are taken just before the root switches its color.

that have not announced this value consider themselves as eliminated until their parent allows them to negotiate again. This moment, when the parent allows its children who lost the election to negotiate again happens immediately after the winner of the election takes the token.

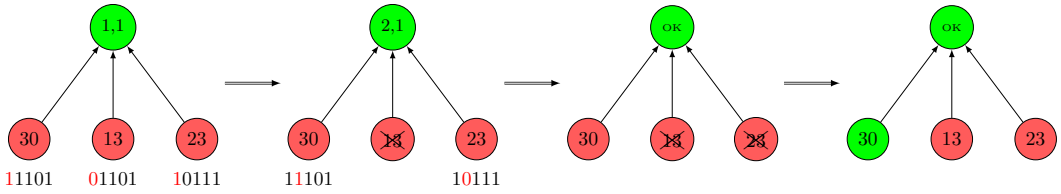

Figure 5.6: Process of the designation of one child to give the token to. The parent reveals the biggest bit announced, then the children with small identifiers quit the negotiation, and in the end the winner takes the token and the losers are ready to negotiate again.

Preventing Livelocks and Deadlocks due to Several Circulating Token: Coloring is not Enough If we consider executions in which one single token is circulating in the network, then the previous is basically enough to ensure fairness and unicity of the token. One variable is used to remember whether the node is still negotiating for the token, and the color of the node states whether the node has already been visited. Hence, we must design an algorithm which achieves fair token circulation from an arbitrary initial configuration.

Things become more complicated when several tokens are circulating. Even with only two tokens, we can design pathological configurations, and especially if the two tokens are of different colors. What we must absolutely prevent is situations in which one node alternatively takes a red token from one parent, sends it back later, then switches color and takes the green token from its other parent, sends it back, and then takes the red one

again, and so on. With two such children, we can design scenarios where the tokens are in a livelock, and the circulation is blocked.

Note that nodes can neither simply ignore the token, for it would create a deadlock: both tokens would be blocked.

To avoid such situations, we force a node that has received the token from one of its parents to not being available for a new token until this parent has sent the token back to its own parent. This requires being more subtle in how we define the election-related variables on the children.

Non-Atomicity of Token Passing Figure 5.6 presents a simplified version of what is actually made. Indeed, at each step of the token passing, from one node to the other, numerous tasks must be fulfilled in order to guarantee the consistency of the network, such as acknowledging the receipt of a token, or resetting the variables of other children, for example. This non-atomicity reflects on the variable used to encode the token.

In a model where nodes can implement pointers to their neighbors, the token can be implemented with only two values (present or absent). In our model, two states is not enough, we give more details on our variable token in Section 5.3.3.1.

5.3.3 First Tools for the Algorithm

5.3.3.1 Variables

Color of the Token To achieve fair perpetual circulation, and in particular to avoid nodes to take the token twice from two different parents, we use a variable *color* which takes two values.

$$\mathtt{c}_v \in \{\mathtt{red}, \mathtt{green}\}$$

Only one operation is used in this variable, which is switching its value from `red` to `green`, or from `green` to `red`. This operation is denoted by $\mathtt{c}_v := \neg \mathtt{c}_v$.

State of the Token To achieve atomicity of the operations required to maintain the network in a consistent state, we need to define a variable which takes eight different values.

$$\mathtt{tok}_v \in \{\bot, \bigstar, \bullet, \Downarrow, \circlearrowleft, \downarrow, \circ, \uparrow\}$$

Let us give some explanations on those different values.

The state $\underline{\mathtt{tok}_v = \bot}$ corresponds to nodes that are **away** from the current token circulation. For example, when the root has the token, all the other nodes are such that $\mathtt{tok}_v = \bot$. Reciprocally $\bot$ is a value that cannot be taken by the root, which would mean that no token is circulating in the network.

When a node that is not involved in the current token circulation receives the token from its parent, it simultaneously switches it color, and switches its token value to $\underline{\mathtt{tok}_v = \bigstar}$. This value corresponds to the moment where one node **receives the token from its parent**, which is supposed to happen exactly once in each circulation round. Therefore, this value will be particularly interesting for the design of the predicate R which will correspond to the implementation of Specification 5.1.

When the parent of v has finally left the token (recall that such operations are not atomic), v can **start negotiating** with its children to decide to which one it will send the token. To do that, it sets $\underline{\mathtt{tok}_v = \bullet}$, and keep that state until one winner is elected.

Once a winner u is elected *i.e.* when all nodes of a different color than v have declared themselves losers, u excepted, v updates $\underline{\texttt{tok}_v = \Downarrow}$ to **offer the token to one of its children**. After that, the child u that won the negotiation can update its own variable $\texttt{tok}_u$ to $\bigstar$.

After its child u took the token, v updates its variable to $\underline{\texttt{tok}_v = \circlearrowleft}$, to let its children which lost the negotiation reset their variable in order to be **ready for the next negotiation**, when v will receive back the token from u.

When all the children of v have finished resetting their variable, v sets $\underline{\texttt{tok}_v = \downarrow}$, which indicates that it is a node involved in a token circulation, but that **the token it had is below it**, to one of its descendants.

After u has finished the token circulation below it, it sets $\texttt{tok}_u = \uparrow$, to send the token back to v. If nothing wrong happened, v has only one child with a value $\texttt{tok} \neq \perp$ at that moment, and therefore can take the token. To do so, it updates $\underline{\texttt{tok}_v = \circ}$, and then **waits for u to effectively drop the token**, by setting $\texttt{tok}_u = \perp$.

After that, v restarts a negotiation, setting $\texttt{tok}_v = \bullet$. After some time, v has no more children of a different color. When this happens, v **offers the token to its parent** w by setting $\underline{\texttt{tok}_v = \uparrow}$. Before effectively dropping the token, v waits that w acknowledges the token by updating its variable to $\texttt{tok}_w = \circ$, but it also waits for all of its children to update their variable linked to the negotiation.

Indeed, in the current situation, all the children of v have successively won the token. As explained above, those nodes are prevented from taking one token of another color, to avoid livelocks. But now, the circulation of the token of v is terminated, so this becomes irrelevant. Worse, if nodes do not reset their variable to a value that allows them to negotiate, then they will all be ignored when a token from another color will come the next round. When all the children of v have reset their variable, v finally drops the token, and sets $\texttt{tok}_v = \perp$.

Remark that for a node v with $\texttt{tok}_v \notin \{\perp, \downarrow\}$, there is a strong pressure on the possible values of $\texttt{tok}_u$, where u is a child of v, and most combinations are actually not supposed to occur in an execution.

Figure 5.7 represents the order between the different states taken by the variable $\texttt{tok}$.

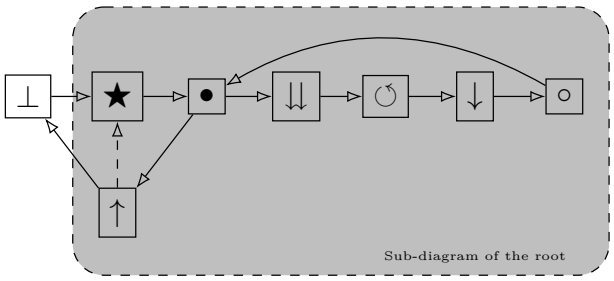

Figure 5.7: Transition diagram for variable $\texttt{tok}_v$.

Negotiation Variables: Bit-by-Bit Communication To negotiate for the token, nodes send bit-by-bit the value of their identifier to their parent. Two variables are required for that. The first one, $\texttt{ph}$, is used to communicate the position of the bit that is currently asked. It takes values between 1 and N, where N is the largest length of an identifier in the network, and thus $N \in O(\log n)$. The second variable is $\texttt{b}$, and is used to communicate the value of the bit that each child has, and for the parent to communicate the highest value it has seen. The variable $\texttt{b}_v$ is used to communicate the values given

by function $\texttt{Bit}_v$. It takes values in $\{0, 1, \perp\}$, where $\perp$ is the value that corresponds, for children, to the absence of a bit at that position (the identifier is too short), and, for the parent, to the request of the value of the bit. For simplicity, we denote the tuple $(\texttt{ph}_v, \texttt{b}_v)$ by the notation $\texttt{id}_v$.

The general scheme is the following: first the parent sets $\texttt{id}_v = (1, \perp)$. Then all the children answer with the value of their first bit, by setting $\texttt{id}_u = (1, \texttt{Bit}_u(1))$. After all the children have answered, v sets $\texttt{id}_v = (1, \texttt{b})$ where $\texttt{b}$ is 1 if v saw a 1 in its children, and 0 otherwise. Then, all the children that do not have answered $(1, \texttt{b})$ lose the negotiation. After all losers have left the negotiation, v sets $\texttt{id}_v = (2, \perp)$, and so on until only one node remains.

Negotiation Variables: State in the Negotiation Process In addition to the identifier variables $\texttt{ph}$ and $\texttt{b}$, we need a variable to remember where each node is in the negotiation process. This variable is denoted by $\texttt{play}$ and takes four values:

$$\texttt{play}_v \in \{\texttt{P}, \texttt{L}, \texttt{W}, \texttt{F}\}$$

Nodes v such that $\texttt{play}_v = \texttt{P}$ are the nodes that are available for a negotiation process with any of their parent that has a different color. They are the only nodes which can **participate** in a negotiation, the others have either already won, or lost. Note that $\texttt{P}$ also stands for players.

When a node v **loses** the negotiation, it sets $\texttt{play}_v = \texttt{L}$, and thus becomes a loser, and is not involved in the current negotiation process anymore.

When a node v **wins** the negotiation and receives the token, it updates its color, its variable $\texttt{tok}_v$, and it also updates $\texttt{play}_v$ to $\texttt{W}$, to signify that it won a negotiation. After that, and until its parent sends the token back to its own parent, v won't negotiate with any of its parents. This guarantees that whatever the initial configuration of the system, v will not create oscillations between two tokens, and therefore no livelock.

We need the fourth value $\texttt{F}$ to overcome a tricky situation. Recall that when a node u sends back the token to its parent w, it first waits for its children who won the negotiation (*i.e.* such that $\texttt{play}_v = \texttt{W}$) to reset their variable $\texttt{play}_v$ to $\texttt{P}$, so that they will be able to negotiate during the next circulation round. But in triangle configurations, it might be that one such node v with $\texttt{play}_v = \texttt{W}$ has a common ancestor, with its parent u, as depicted in Figure 5.8.

In such a situation, v should not update $\texttt{play}_v = \texttt{P}$. Indeed if it does, then from w's perspective, it is exactly as if we never introduced the value $\texttt{W}$: it has one child who won the token and is at $\texttt{P}$. In particular, we can now create a livelock at the level of w.

But v can neither keep its value at $\texttt{W}$, for it creates a deadlock with its parent u. Therefore, v **fakes** resetting its variable $\texttt{play}_v$, by setting it to $\texttt{F}$, which is understood by its parent with $\texttt{tok}_u = \uparrow$ as a $\texttt{P}$ value, but both v and its grandparent w do recall that v has already had the token. This requires that as soon as possible, node v update its variable $\texttt{play}_v = \texttt{W}$, otherwise it could miss the opportunity to eventually reset its variable at $\texttt{P}$, notably if w sets $\texttt{tok}_w = \uparrow$.

Non-Redundancy of $\texttt{c}_v$ and $\texttt{play}_v = \texttt{W}$ Although the alternating between two colors and the state $\texttt{play}_v = \texttt{W}$ share the common goal, which is to prevent a node to have several times the token, they are relevant in totally different situations.

The alternation between two colors is built to prevent a node from receiving twice the same token, from parents that might be totally unrelated in the $\texttt{D}^o$.

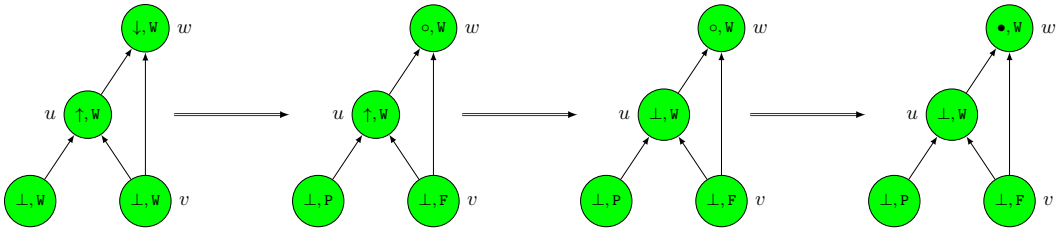

Figure 5.8: Cleaning children before sending up the token in a triangle: $\texttt{play}_v = \texttt{F}$.

On the other hand, the value W prevents a node to oscillate between two colors, from two different parents, which would block the circulation of the token.

5.3.3.2 Common Sets

To define the rules of our algorithm, some sets are especially useful. An important distinction that nodes almost systematically do is the difference between their parents with the same color, and their parents with the other color.

$$\mathcal{P}_{\texttt{eq}}(v) = \{u \in \mathcal{P}(v) \mid \texttt{c}_u = \texttt{c}_v\} \tag{5.1}$$

$$\mathcal{P}_{\texttt{neq}}(v) = \{u \in \mathcal{P}(v) \mid \texttt{c}_u \neq \texttt{c}_v\} \tag{5.2}$$

Similarly, nodes distinguish their children with the same color from their children with the other color.

$$\mathcal{C}_{\texttt{eq}}(v) = \{u \in \mathcal{C}(v) \mid \texttt{c}_u = \texttt{c}_v\} \tag{5.3}$$

$$\mathcal{C}_{\texttt{neq}}(v) = \{u \in \mathcal{C}(v) \mid \texttt{c}_u \neq \texttt{c}_v\} \tag{5.4}$$

One other central set for the negotiation is the set of the nodes with whom the parent negotiates. The parent negotiates with its children that do not have the same color, and such that $\texttt{play}_u = \texttt{P}$. A node that negotiates is not supposed to have any children u such that $\texttt{tok}_u \neq \bot$, and a specific rule is designed for such erroneous situations. Yet, to simplify some reasoning, we also add as a constraint the fact that v only negotiates with its children such that $\texttt{tok}_u = \bot$. In the latter, we call these nodes the players of v.

$$\texttt{Players}(v) = \{u \in \mathcal{C}_{\texttt{neq}}(v) \mid \texttt{tok}_v = \bot \wedge \texttt{play}_u = \texttt{P}\} \tag{5.5}$$

5.3.4 Rules of the Algorithm

In this section, we present the predicates, actions, and rules of our algorithm. For the sake of readability, we have grouped the rules according to the task they relate to. In Section 5.3.4.1, we present the rule dedicated to deal with inconsistencies of the variable tok. In Section 5.3.4.3, we present the negotiation process, based on the identifiers of the children of the node holding the token. In Section 5.3.4.2 we present the rules that make nodes quit that negotiation process. In Section 5.3.4.4 we present the rules which reset variables of nodes to a proper value after the end of the negotiation process. In Section 5.3.4.5 we present the rules that allow one node to receive the token back from one of its children. In Section 5.3.4.6 we present the rules that are executed when one circulation below a node is terminated, whether the node is the root or another node. Finally, in Section 5.3.4.7, all the rules are gathered to allow an easy access during the reading of the proofs.

5.3.4.1 Error

As hinted above, some combinations of the variable $\texttt{tok}$ should not occur on a node and its children. In general, a node v that has the token, or which is very near a token, *i.e.* with $\texttt{tok}_v \in \{\bigstar, \bullet, \downdownarrows, \circlearrowleft, \circ, \uparrow\}$, should only have children u with $\texttt{tok}_u = \bot$, with a few exceptions.

Indeed, if $\texttt{tok}_v \in \{\downdownarrows, \circlearrowleft\}$, then v is allowed to have a child such that $\texttt{tok}_u = \bigstar$, the child to which it just gave the token.

If $\texttt{tok}_v = \circ$, then v is allowed to have a child such that $\texttt{tok}_u = \uparrow$, the child from which it is receiving the token.

One other exception is when $\texttt{tok}_v = \uparrow$. In this case, v is sending the token back to its parent, which means that it is on its way to set $\texttt{tok}_v = \bot$. We do not consider any error for such nodes, since it would not be very effective, and could create an error at the level of its own parent.

To repair such errors, the principle is that the parent trusts its child, in the sense that if both v and u believe they have a token, then v forgets its token, and repairs itself by setting $\texttt{tok}_v = \downarrow$, acknowledging the fact that there is a token below it.

The illegal pairs are detected by the predicate $\texttt{Er}(v)$ and the correctness of the state of the token of node v is handled by the rule $\mathbb{E}_{\texttt{TrustChild}}$.

$$\begin{aligned} \texttt{Er}(v) \quad \equiv \quad & \big(\texttt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \neq \bot\big) \\ \vee \quad & \big(\texttt{tok}_v \in \{\downdownarrows, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \bigstar\}\big) \\ \vee \quad & \big(\texttt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \uparrow\}\big) \end{aligned} \tag{5.6}$$

$\mathbb{E}_{\texttt{TrustChild}}$: for all $v \in V$

$$\texttt{Er}(v) \longrightarrow \texttt{tok}_v := \downarrow$$

All the other rules of the algorithm suppose that node v is not in such an error. For homogeneity and readability, we constrained all rules with $\neg\texttt{Er}(v)$, even rules which suppose that $\texttt{tok}_v \in \{\bot, \downarrow, \uparrow\}$.

5.3.4.2 End of the Negotiation Phase

Before showing how the negotiation phase is implemented, let us first explain how nodes may quit this phase. In a negotiation, two types of nodes are involved, the parent v which has the token, such that $\texttt{tok}_v = \bullet$, and the children u that are elements of $\texttt{Players}(v)$.

The parent can quit the negotiation phase by two means.

Rule $\mathbb{R}_{\texttt{OfferUp}}$ The first one is when the parent observes that all of its children in $\texttt{Players}(v)$ have been visited, which means that it ended the circulation below itself. When this happens the parent sends the token back to its own parent.

Before doing so, the parent must first be assured that it is not in a situation as depicted in the last configuration of Figure 5.8. If v has a child u such that $\texttt{play}_u = \texttt{F}$, then it is a node which actually won the token, but faked being reset to $\texttt{P}$ to another parent p such that $\texttt{tok}_p = \uparrow$. If v is the last parent of u, and switches $\texttt{tok}_v = \uparrow$, then u will never be reset to $\texttt{P}$, which breaks the fairness of the token circulation. Therefore, before executing this action, v waits that all of its children have quit the value $\texttt{F}$.

The predicate WaitSib corresponds to this requirement.

$$\texttt{WaitSib}(v) \equiv \exists c \in \mathcal{C}_{\texttt{eq}}(v) : \texttt{play}_c = \texttt{F} \tag{5.7}$$

$\mathbb{R}_{\texttt{OfferUp}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \neg\texttt{WaitSib}(v) \wedge (\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \in \{\texttt{W}, \texttt{F}\}) \longrightarrow \texttt{tok}_v :=\uparrow$$

Note that even if the root has no parent, it may execute rule $\mathbb{R}_{\texttt{OfferUp}}$ when it does not have any more children to visit. Since some operations subsequent to having finished circulation below oneself are identical for the root and the other nodes, we factorize the rules as far as possible.

Rule $\mathbb{R}_{\texttt{Give}}$ The second possibility for a parent to quit the negotiation is to decide to offer the token to one of its children. This happens when node v has exactly one child in $\texttt{Players}(v)$.

But there is one other situation to which we must give attention. Suppose that due to a wrong initialization of the network, no node belongs to $\texttt{Players}(v)$, but the circulation is not over yet, for one or several children of v are losers, *i.e.* $\texttt{play}_u = \texttt{L}$. In this situation, node v needs to reset the variable play of these children to P, and then will perform a negotiation between them. One simple way to do that is to follow the transition diagram of tok depicted in Figure 5.7. No child of v will take the token offered by $\texttt{tok}_v =\downdownarrows$, but immediately after, when $\texttt{tok}_v$ is set to $\circlearrowleft$, the losers will update $\texttt{play}_u = \texttt{P}$, which will allow a further negotiation process.

Actually, v basically executes a full round of its variable tok just to refresh its wrongly initialized children. Note that this may only be caused by a corrupted initial configuration, and that after convergence, this part of the rule becomes useless.

The predicate Give describes the situations in which node v ends the negotiation by sending the token to one of its children.

$$\texttt{Give}(v) \equiv |\texttt{Players}(v)| = 1 \vee (|\texttt{Players}(v)| = 0 \wedge \exists u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u = \texttt{L}) \tag{5.8}$$

$\mathbb{R}_{\texttt{Give}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \texttt{Give}(v) \longrightarrow \texttt{tok}_v :=\downdownarrows$$

Let us now consider the children $u \in \texttt{Players}(v)$. A child quits the negotiation if it quits $\texttt{Players}(v)$. It may do that by three means.

Rule $\mathbb{R}_{\texttt{FakeWin}}$ The first situation in which a child quits the negotiation is when it has an inconsistent parenthood. More precisely, if v has several parents that are dealing with a token. Such parents are those whose value $\texttt{tok}_p \notin \{\bot, \downarrow, \uparrow\}$.

We do not consider the nodes who are not involved in a token circulation ($\texttt{tok}_p = \bot$) neither do we consider the nodes who announce a token in their descendants ($\texttt{tok}_p =\downarrow$). We also do not consider $\uparrow$ for the same reason that it was ignored in predicate Er: such parent is about to drop the token to its own parent, it is therefore unnecessary to consider it.

The parents which may force v to quit the negotiation are ParNeg:

$$\texttt{ParNeg}(v) = \{p \in \mathcal{P}(v) \mid \texttt{tok}_p \notin \{\bot, \uparrow, \downarrow\}\} \tag{5.9}$$

Let us describe the situations where v does not have to quit the negotiation due to an inconsistent parenthood. First, if v has exactly one parent in $\texttt{ParNeg}(v)$, then there is no inconsistency. If v has no parent in $\texttt{ParNeg}(v)$, there is no inconsistency either.

There is one other situation that may occur in legal execution. In a triangle configuration, v has two parents and one of them is the children of the other. Then the middle one receives the token and sets its variable $\texttt{tok}$ to $\bigstar$, there is a moment where v has its variable $\texttt{play}_v = \texttt{P}$ and also has two parents in $\texttt{ParNeg}(v)$, since the other parent is at $\circlearrowleft$. This is described in Figure 5.9

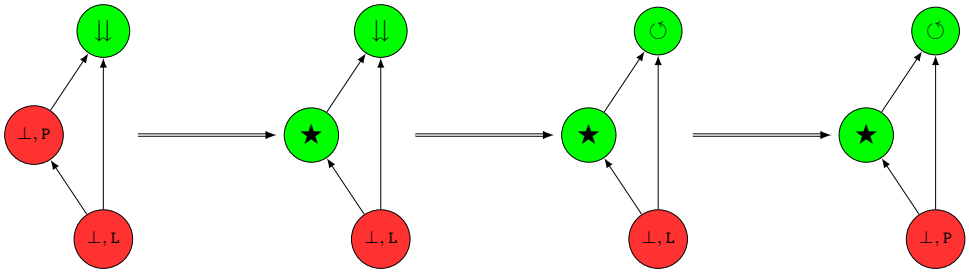

Figure 5.9: Node with several parents indicating a token

Formally, the situations that do not force v to interrupt the negotiation are:

$$\texttt{OkNeg}(v) \equiv \begin{cases} |\texttt{ParNeg}(v)| \leq 1 \\ \vee \\ (|\texttt{ParNeg}(v)| = 2 \wedge \{\texttt{tok}_p \mid p \in \texttt{ParNeg}(v)\} = \{\circlearrowleft, \bigstar\}) \end{cases} \tag{5.10}$$

When a node decides to stop negotiating with its parent for inconsistency reasons, it fakes winning the token, in the sense that it becomes visited by switching its color, and updates its variable $\texttt{play}$, but does not actually take the token. In most cases, switching to $\texttt{W}$ is sufficient. Hence, if before switching its color, v has a parent of the other color, such that $\texttt{tok}_p = \uparrow$, then at the next computing step, v will update $\texttt{play}_v$ to $\texttt{F}$, as shown in Figure 5.8. In order to spare one computing step, and to simplify some proofs, we set $\texttt{play}_v$ at the right value at once.

The action corresponding to the rule $\mathbb{R}_{\texttt{FakeWin}}$ is given by $\texttt{FakeWin}$:

$$\texttt{FakeWin}(v) \equiv \begin{cases} \texttt{c}_v & := \neg \texttt{c}_v; \\ \texttt{play}_v & := \begin{cases} \texttt{W} & \text{if } \forall p \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_p \neq \uparrow; \\ \texttt{F} & \text{if } \exists p \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_p = \uparrow; \end{cases} \end{cases} \tag{5.11}$$

$\mathbb{R}_{\texttt{FakeWin}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \neg\texttt{OkNeg}(v) :\longrightarrow \texttt{FakeWin}(v)$$

Rule $\mathbb{R}_{\texttt{Win}}$ The second situation in which a child quits the negotiation is when it wins the negotiation. A node v wins the negotiation if its parents are not in an inconsistent configuration (which is $\texttt{OkNeg}(v)$), and if it is still at $\texttt{play}_v = \texttt{P}$ when its parent offers it the token (*i.e.* when $\texttt{tok}_p = \downdownarrows$).

The predicate $\texttt{WinNeg}$ is dedicated to detect such situations.

$$\texttt{WinNeg}(v) \equiv \texttt{OkNeg}(v) \wedge \exists p \in \mathcal{P}_{\texttt{neq}}(v) : \texttt{tok}_p = \downdownarrows \tag{5.12}$$

When a node v wins the negotiation, it takes the token, and switches its color. Furthermore, it updates its variable $\texttt{play}_v$ to W. Contrary to how we did for FakeWin, we do not have to worry about the precise value of $\texttt{play}_v$ right now. Indeed, since v has a value different than $\perp$ for $\texttt{tok}_v$, v does not have to worry about the moment where it will be reactivated by its parent, which happens at the end of the circulation of its parent.

$$\texttt{Win}(v) \equiv \begin{cases} \texttt{tok}_v & := \bigstar; \\ \texttt{c}_v & := \neg \texttt{c}_v; \\ \texttt{play}_v & := \texttt{W}; \end{cases} \tag{5.13}$$

$\mathbb{R}_{\texttt{Win}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{WinNeg}(v) \longrightarrow \texttt{Win}(v)$$

Rule $\mathbb{R}_{\texttt{Lose}}$ The third and last situation in which a child quits the negotiation is when it loses it. The exact condition of how a child loses the negotiation will be properly defined in the following section. For now, let us just consider that there exists a predicate $\texttt{LosePar}(v, p)$, which depends on the values of $\texttt{id}_v$ and $\texttt{id}_p$, where $p \in \mathcal{P}_{\texttt{neq}}(v)$, which describes that v should lose the negotiation according to p.

A node loses the negotiation if its parents are not in an inconsistent configuration (which is $\texttt{OkNeg}(v)$, and if it does not win the negotiation (which is $\neg\texttt{WinNeg}(v)$).

All these conditions are grouped in the predicate $\texttt{LoseNeg}(v)$:

$$\texttt{LoseNeg}(v) \equiv \texttt{OkNeg}(v) \wedge \neg\texttt{WinNeg}(v) \wedge \exists p \in \mathcal{P}_{\texttt{neq}}(v) : (\texttt{tok}_p = \bullet \wedge \texttt{LosePar}(v, p)) \tag{5.14}$$

When v loses the negotiation, it simply sets $\texttt{play}_v = \texttt{L}$.

$\mathbb{R}_{\texttt{Lose}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{LoseNeg}(v) \longrightarrow \texttt{play}_v := \texttt{L}$$

5.3.4.3 Negotiation Identifier-Based

In the negotiation phase, the only children which are relevant to consider, from the parent's perspective, are the children in $\texttt{Players}(v)$. In the description of the rules, we write players instead of children in $\texttt{Players}(v)$, to facilitate the reading.

Let us first describe the general mechanism of the election, then explain some subtleties, before presenting the rules of the bit-by-bit negotiation.

One first important idea is that the parent does not execute any action unless all of its players have executed their rule. In particular, the parent waits for all of its players to be at the same value of ph as it is.

$$\texttt{Synch}(v) \equiv \forall u \in \texttt{Players}(v), \texttt{ph}_u = \texttt{ph}_v \tag{5.15}$$

One other general principle, which is crucial due to the asynchrony of the system, and due to the self-stabilizing requirement, is that nodes do not correct their answer. More precisely, if one node v (parent or child in the negotiation) is in a state that may have an influence on the execution of a rule by one other node u, then v does not update its state. Even if this state is inconsistent with its identifier, for a child, or with the values of $\texttt{id}_u$ on

its players, for a parent, v waits that its neighbors have finished interpreting this state, and accept the consequences of this error. Such inconsistencies are typically due to an incorrect initialization of the system. This general principle only applies to the negotiation rules, and does not prevent nodes to execute rules such as $\mathbb{E}_{\texttt{TrustChild}}$, $\mathbb{R}_{\texttt{FakeWin}}$, or $\mathbb{R}_{\texttt{Give}}$ for example.

Let us recall the general scheme of how the negotiation process happens

0. Parent v asks for the value of the first bit of its players $u \in \texttt{Players}(v)$, by setting $\texttt{id}_v = (1, \bot)$.

1. Players u of v answers with their own value $(1, \texttt{Bit}_u(1))$ (rule $\mathbb{R}_{\texttt{NewBit}}$).

2. When all its players have answered, *i.e.* when all have the right value $\texttt{ph}_u = 1$, v announces the biggest value it sees, by setting $\texttt{id}_v = (1, \texttt{b})$ (rule $\mathbb{R}_{\texttt{maxPos}}$).

3. Players u of v which have a lower value than $\texttt{b}_v$ lose the negotiation (rule $\mathbb{R}_{\texttt{Lose}}$).

4. When all of its players have $\texttt{ph}_u = 1$ and $\texttt{b}_u = \texttt{b}_v$, v asks for the value of the second bit of its remaining players, by setting $\texttt{id}_v = (2, \bot)$ (rule $\mathbb{R}_{\texttt{NewPh}}$), and so on.

A particular situation is when all the players u of v answer $(\texttt{ph}, \bot)$, *i.e.* when all the players declare that they do not have a long-enough identifier to provide a $\texttt{ph}$-th bit. This situation is not possible in correct executions, since all identifiers are distinct, we cannot reach a point where several identifiers are similar and terminated. Hence, this may happen in the early steps of an execution which starts in an inconsistent configuration. Either at least one child has an identifier actually greater than $\texttt{ph}$, and could go further, but due to an incorrect initialization, it has a wrong value for $\texttt{b}$, either due to an incorrect initialization of the parent, the negotiation process did not start at $\texttt{ph} = 1$ but further, and we could have missed the opportunity to distinct the identifiers of the nodes in $\texttt{Players}(v)$.

When this happens, the parent cannot simply increase $\texttt{ph}$, since its players formulate that they can't even reach the current value of $\texttt{ph}$. Therefore, if all of its players answer $(\texttt{ph}, \bot)$, then the correct move for the parent is to restart at $\texttt{ph}_v = 1$. Only one exception to that rule, is when it happens when $\texttt{ph}_v = 1$ already. In this situation, v would not change a single bit of information by setting $\texttt{id}_v = (1, \bot)$. We suppose that all identifiers have at least one bit, even if it is a single bit 0. Therefore, $(1, \bot)$ can never be a valid answer for a child. In consequences, without any contradiction with the previous principle stating that no possibly valid answer can be revised, players such that $\texttt{id}_v = (1, \bot)$ can actually update their state to send their actual value for $\texttt{Bit}_u(1)$. This is not true for other values of $\texttt{ph}$: if $\texttt{id}_v = (\texttt{ph}, \bot)$ with $\texttt{ph} > 1$ and $\texttt{Bit}_v(\texttt{ph}) \neq \bot$, then to avoid confusing its parent, it does not update its state.

Let us now treat the different rules corresponding to the scheme presented above.

Rule $\mathbb{R}_{\texttt{NewBit}}$ Some first conditions for a node v to answer the negotiation is that its parenthood is not inconsistent, and it is not winning the negotiation, nor losing it. This corresponds to the partial predicate $\texttt{OkNeg}(v) \wedge \neg\texttt{WinNeg}(v) \wedge \neg\texttt{LoseNeg}(v)$.

The other condition is that v is in a situation where it is its turn to answer. One first situation in which it should answer is when it does not have the same value of $\texttt{ph}$ as its negotiating parent p. The second situation is the particular case where $\texttt{ph}_v = \texttt{ph}_p = 1$, and v has answered $\bot$, which is never a valid answer, and the parent p is still waiting for its players to answer. Indeed, in an poorly initialized configuration, the parent p could have already decided which value was the highest, and thus v must not answer.

This is synthesized in predicate AnswerPar:

$$\texttt{AnswerPar}(v,p) \equiv \texttt{ph}_v \neq \texttt{ph}_p \vee \begin{cases} \texttt{ph}_p = 1 \\ \texttt{b}_p = \bot \\ \texttt{b}_v = \bot \end{cases} \tag{5.16}$$

We also define the predicate which excludes concurrency with $\mathbb{R}_{\texttt{FakeWin}}$, $\mathbb{R}_{\texttt{Win}}$, and $\mathbb{R}_{\texttt{Lose}}$:

$$\texttt{AnswerNeg}(v) \equiv \begin{cases} \texttt{OkNeg}(v) \wedge \neg\texttt{WinNeg}(v) \wedge \neg\texttt{LoseNeg}(v) \wedge \\ \exists p \in \mathcal{P}_{\texttt{neq}}(v) : (\texttt{tok}_p = \bullet \wedge \texttt{AnswerPar}(v,p)) \end{cases} \tag{5.17}$$

The action which corresponds to answering the negotiation is updating variable $\texttt{ph}_v$ to the value of our parent, and $\texttt{b}_v$ according to function $\texttt{Bit}_v$.

Remark that such a parent p is necessarily unique, due to $\texttt{OkNeg}(v)$. To simplify the notations, we consider that such parent p is given as an external information to the action which corresponds to AnswerNeg, Announce:

$$\texttt{Announce}(v) \equiv \texttt{id}_v := (\texttt{ph}_p, \texttt{Bit}_v(\texttt{ph}_p)) \tag{5.18}$$

$\mathbb{R}_{\texttt{NewBit}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bot \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{AnswerNeg}(v) \longrightarrow \texttt{Announce}(v)$$

Rule $\mathbb{R}_{\texttt{maxPos}}$ One first condition for the parent to announce the maximal value it sees on its players is that the negotiation is not over yet, *i.e.* there are at least two players.

One second condition is that all of those players have answered, *i.e.* that they have the same value as itself for ph, which is captured by $\texttt{Synch}(v)$.

One third condition is that it does not have already announced a value, *i.e.* $\texttt{b}_v = \bot$.

There are two more subtleties. The first one is that if we have $\texttt{ph}_v = 1$ then v actually waits for all of its players to produce a non-empty answer, a value for $\texttt{b}_u$ which is not $\bot$, an invalid answer for $\texttt{ph} = 1$.

The second subtlety is that if, for a value of ph greater than 1, all the players of v announce $\bot$, then it is irrelevant to keep increasing phases, and actually irrelevant to answer $\bot$ as well. In this situation, the correct move is to restart the negotiation from $\texttt{ph}_v = 1$, which will be dealt with by rule $\mathbb{R}_{\texttt{NewPh}}$.

The last condition and the subtleties are described by predicate NextPlay.

$$\texttt{NextPlay}(v) \equiv \texttt{b}_v = \bot \wedge \begin{cases} \texttt{ph}_v = 1 \wedge \forall u \in \texttt{Players}(v), \texttt{b}_u \neq \bot \\ \vee \\ \texttt{ph}_v \neq 1 \wedge \exists u \in \texttt{Players}(v) : \texttt{b}_u \neq \bot \end{cases} \tag{5.19}$$

Once all these conditions are fulfilled, v can announce the biggest value of b it sees in its playing childhood (classically we have $\bot < 0 < 1$). This action is denoted MaxPos.

$$\texttt{MaxPos}(v) \equiv \texttt{b}_v := \max_{u \in \texttt{Players}(v)} \texttt{b}_u \tag{5.20}$$

$\mathbb{R}_{\texttt{maxPos}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge |\texttt{Players}(v)| \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{NextPlay}(v) \longrightarrow \texttt{MaxPos}(v)$$

Predicate for Rule $\mathbb{R}_{\texttt{Lose}}$ In the previous section we did not explicitly give the predicate `LosePar`. One node v loses the negotiation when they have the same value of `ph` as their parent, and that the answer provided by their parent is different from the one they have.

$$\texttt{LosePar}(v,p) \equiv \texttt{ph}_p = \texttt{ph}_v \wedge \texttt{b}_p \neq \bot \wedge \texttt{b}_p \neq \texttt{b}_v \tag{5.21}$$

Rule $\mathbb{R}_{\texttt{NewPh}}$ The parent increases its phase only when the negotiation round is terminated, which means that nonetheless all its players have the same value as it has, for `ph`, but also that they all have the same value as is has for `b`.

Either this value is $\bot$, and therefore all players are out of bits, and the new phase should be 1, either it is not $\bot$, and therefore all the nodes asked to lose have actually lost and are now out of $\texttt{Players}(v)$, and the new phase should be one more than the current one.

There is one particular case where this increase should not happen: when we have $\texttt{ph} = 1$ and $\texttt{b} = \bot$ (on both the parent and its players), since it does not correspond to a valid answer. Predicate `PhComplete` summarizes that.

$$\texttt{PhComplete}(v) \equiv (\texttt{ph}_v \neq 1 \vee \texttt{b}_v \neq \bot) \wedge \forall u \in \texttt{Players}(v), \texttt{b}_u = \texttt{b}_v \tag{5.22}$$

$$\texttt{PhasePlus}(v) \equiv \begin{cases} \texttt{b}_v := \bot \\ \texttt{ph}_v := \begin{cases} 1 & \text{if } \forall u \in \texttt{Players}(v), \texttt{b}_u = \bot \\ \texttt{ph}_v + 1 & \text{if } \exists u \in \texttt{Players}(v), \texttt{b}_u \neq \bot \end{cases} \end{cases} \tag{5.23}$$

$\mathbb{R}_{\texttt{NewPh}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge |\texttt{Players}(v)| \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{PhComplete}(v) \longrightarrow \texttt{PhasePlus}(v)$$

5.3.4.4 Operations Post-Negotiation

In this section we consider actions subsequent to the executions of $\mathbb{R}_{\texttt{Give}}$ by the parent, and $\mathbb{R}_{\texttt{Win}}$ by one child, which were presented in Section 5.3.4.2. The parent has its variable $\texttt{tok} = \downdownarrows$, the winning player has its variable $\texttt{tok} = \bigstar$, and the other children that have not received the token yet have their variable $\texttt{play} = \texttt{L}$.

Rule $\mathbb{R}_{\texttt{NewPlay}}$ The first action that the parent takes, after its child has actually received the token, is switching its variable `tok` to $\circlearrowleft$ to inform its other children that the negotiation phase is terminated, and that they are free to negotiate again. This is necessary to ensure a proper depth-first traversal, since some of them may also be children of the winning player.

This can be done as soon as v does not have any players. Since we classically suppose that v is not in error, it means that its children u of the opposite color are either at $\texttt{tok}_u = \bigstar$, or at $\texttt{play}_u \neq \texttt{P}$.

$\mathbb{R}_{\texttt{NewPlay}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_u = \downdownarrows \wedge (\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{P} \vee \texttt{tok}_u = \bigstar) \longrightarrow \texttt{tok}_v := \circlearrowleft$$

Rule $\mathbb{R}_{\texttt{ReplayD}}$ When v has one parent which notifies that it can reset its value of $\texttt{play}_v$ to P, v first gets sure that it is not near one other negotiation phase. Indeed, v should not join a running negotiation phase by updating $\texttt{play}_v$ from L to P. In correct executions, v cannot be close to several negotiation phases, by unicity of the token. Yet, due to an inconsistent initialization of the network, this must be considered.

Parents that correspond to ongoing negotiation phases are those with $\texttt{tok}_p = \bullet$ and $\texttt{tok}_p = \downdownarrows$.

$$\texttt{ReplayL}(v) \equiv \forall u \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_u \notin \{\bullet, \downdownarrows\} \wedge \exists u \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_u \in \{\circlearrowleft, \circ\} \tag{5.24}$$

Note that we also consider situations where v has a parent p with $\texttt{tok}_p = \circ$. This corresponds to a similar situation, which we will develop in Section 5.3.4.5

When this predicate is satisfied, v updates $\texttt{play}_v$ to P, and simultaneously resets its variable $\texttt{id}_v$ to a neutral value.

$$\texttt{Replay}(v) \equiv \left\{ \begin{array}{l} \texttt{play}_v := \texttt{P}; \\ \texttt{id}_v := (1, \bot) \end{array} \right. \tag{5.25}$$

$\mathbb{R}_{\texttt{ReplayD}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bot \wedge \texttt{play}_v = \texttt{L} \wedge \texttt{ReplayL}(v) \longrightarrow \texttt{Replay}(v)$$

Rule $\mathbb{R}_{\texttt{Drop}}$ When all of its children have reset their variable play from L to P, node v can finally update its variable tok to $\downarrow$, which terminates the passing of the token to the child who won the negotiation.

To avoid deadlocks due to an inconsistent initial configuration, we must consider situations in which some child of v has the token ($\texttt{tok}_u = \bigstar$) but also has its variable $\texttt{play}_u$ at L. Rather than creating a new rule to make u update its variable play, we chose to let v drop the token in such situations.

$\mathbb{R}_{\texttt{Drop}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \circlearrowleft \wedge (\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{L} \vee \texttt{tok}_u = \bigstar) \longrightarrow \texttt{tok}_v := \downarrow$$

Rule $\mathbb{R}_{\texttt{Nego}}$ Once the parent of the node v who won the negotiation has terminated the previous actions, and has set its variable tok to $\downarrow$, v can finally start negotiating with its own children, by setting $\texttt{tok}_v = \bullet$ and its variables ph and b to the initial value of the negotiating process.

$$\texttt{StartNego}(v) \equiv \left\{ \begin{array}{ll} \texttt{tok}_v & := \bullet; \\ \texttt{id}_v & := (1, \bot); \end{array} \right. \tag{5.26}$$

$\mathbb{R}_{\texttt{Nego}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bigstar \wedge (\forall p \in \mathcal{P}(v) : \texttt{tok}_u \notin \{\downdownarrows, \circlearrowleft\}) \longrightarrow \texttt{StartNego}(v)$$

Figure 5.11 presents the process of how a token is given to a child.

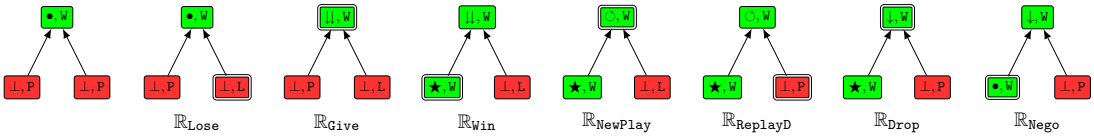

Figure 5.10: Execution of rules for returning the token back to one child. The activated node is the one with its line doubled

5.3.4.5 Reception of the Token from a Child

In this section we consider actions subsequent to the execution of $\mathbb{R}_{\texttt{OfferUp}}$ (presented in Section 5.3.4.2). We consider one node v such that $\texttt{tok}_v = \downarrow$, which has one child u that offered the token to its parent by setting $\texttt{tok}_u = \uparrow$.

Rule $\mathbb{R}_{\texttt{Receive}}$ When node v has its child u announcing that it returns the token back, v needs to be sure that it does not have any other children which is involved in a token circulation, in which case it cannot take the token before its other children have terminated their circulation. Recall that in Figure 5.4, the top node did not take the token.

If v takes the token while having another child involved in a token circulation, it becomes in error, and thus executes $\mathbb{E}_{\texttt{TrustChild}}$, and sets $\texttt{tok}_v = \downarrow$, which may create a livelock.

$\mathbb{R}_{\texttt{Receive}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \downarrow \wedge (\forall u \in \mathcal{C}(v), \texttt{tok}_u \in \{\bot, \uparrow\}) \longrightarrow \texttt{tok}_v := \circ$$

Rule $\mathbb{R}_{\texttt{ReNego}}$ If node v did not have any children and took the token by setting $\texttt{tok}_v = \circ$ with rule $\mathbb{R}_{\texttt{Receive}}$, then before starting negotiating, it waits that its child, from which it received the token, actually drops it.

More precisely, v start negotiating only if it all of its children are such that $\texttt{tok}_u = \bot$, otherwise it would create an error by setting $\texttt{tok}_v = \bullet$. The following predicate guarantees that one node has no children involved in a token circulation:

$$\texttt{Leaf}(v) \equiv \forall u \in \mathcal{C}(v), \texttt{tok}_u = \bot \tag{5.27}$$

Furthermore, before starting a negotiation round, v also waits that all its children which have not won the token yet have properly reset their variable `play` to `L`. This is guaranteed since Rule 5.3.4.4 applies to situations where the parent has its variable $\texttt{tok} = \circ$.

$\mathbb{R}_{\texttt{ReNego}}$: for all $v \in V$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \circ \wedge \texttt{Leaf}(v) \wedge \forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{L} \longrightarrow \texttt{StartNego}(v)$$

5.3.4.6 End of the Circulation

In this section we also present actions subsequent to the execution of $\mathbb{R}_{\texttt{OfferUp}}$ (presented in Section 5.3.4.2), but from the child perspective. We consider the node which executed $\mathbb{R}_{\texttt{OfferUp}}$, and its potential children.

Rule $\mathbb{R}_{\texttt{Return}}$ After it has offered the token to its parent, a node v which is not the root should eventually drop the token, and set $\texttt{tok}_v = \bot$. Before that, it must check that all its children have correctly updated their variable `W` to `P` to assure that when a token of the other color will come, all those nodes will be available for a negotiation phase. This only applies to children of v which have the same color as v, since other children may be involved in a concurrent token circulation, and we do not want to create any livelock. Also, it only applies to nodes which are not involved in any circulation of the token, *i.e.* such that $\texttt{tok}_u = \bot$ (recall that since $\texttt{tok}_v = \uparrow$, v cannot be in error).

If such children u of v, with $\texttt{play}_u = \texttt{W}$ still have parents involved in a negotiation, they will shift that value to `F`, faking a reset to v. Thus, v should tolerate children u with $\texttt{play}_u = \texttt{F}$.

On the other hand, if v has children u which have their variable $\texttt{play}_u = \texttt{L}$, those children might be involved in another negotiation phase, which they have lost. Those children should not reset their variable `play` to `P`, for it would harm the negotiation process.

Thus, v checks that all of its children with the same color have a value of `play` different from `W` before it sends the token back.

We add one more consistency check, which is that v does not have any parent in error, *i.e.* parents p such that $\texttt{tok}_p \in \{\bigstar, \bullet, \downdownarrows, \circlearrowleft\}$. If it has, it waits for those parents to execute $\mathbb{E}_{\texttt{TrustChild}}$ before returning the token to its parents. This predicate is not necessary for our algorithm to work, but it may make convergence faster.

These predicates are combined in predicate `MayDropUp`.

$$\texttt{MayDropUp}(v) \equiv \left\{ \begin{array}{ll} & \forall p \in \mathcal{P}(v), \texttt{tok}_p \in \{\bot, \downarrow, \uparrow, \circ\} \\ \wedge & \forall u \in \mathcal{C}_{\texttt{eq}}(v), (\texttt{tok}_u = \bot \Rightarrow \texttt{play}_u \in \{\texttt{P}, \texttt{L}, \texttt{F}\}) \end{array} \right. \tag{5.28}$$

When this condition is fulfilled, v can drop the token. This includes setting $\texttt{tok}_v = \bot$ and updating $\texttt{play}_v$ to a correct value depending on the parents of v. As depicted in Figure 5.8, there are situations where nodes without a token must set their variable at `F` instead of `W`, to avoid blocking one parent which is close to execute $\mathbb{R}_{\texttt{Return}}$ itself.

$$\texttt{Drop}(v) \equiv \left\{ \begin{array}{lll} \texttt{tok}_v & := \bot; & \\ \texttt{play}_v & := \left\{ \begin{array}{ll} \texttt{W} & \text{if } \forall p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p \neq \uparrow \\ \texttt{F} & \text{if } \exists p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p = \uparrow \end{array} \right. & \end{array} \right. \tag{5.29}$$

$\mathbb{R}_{\texttt{Return}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \uparrow \wedge \texttt{MayDropUp}(v) \longrightarrow \texttt{Drop}(v)$$

Rule $\mathbb{R}^r_{\texttt{NewDFS}}$ In a similar situation, where all of its children have repaired themselves, the root does not drop the token, for it would simply destroy it. When the root is in such a situation, it means that the current circulation round is terminated, and one other can start.

To do that, the root resets its token value to $\bigstar$, which corresponds to the first time the token is held by any node in the new circulation round, and switches its color. After that, the root will be able to execute rule $\mathbb{R}_{\texttt{Nego}}$ and to start a negotiation process with its children.

$$\texttt{NewToken}(r) \equiv \left\{ \begin{array}{ll} \texttt{c}_r & := \neg\texttt{c}_r \\ \texttt{tok}_r & := \bigstar \end{array} \right. \tag{5.30}$$

This action can be seen as one atomic execution of the two actions Drop and Win, which corresponds to what r would have done, non-atomically, if it were not the root.

$\mathbb{R}^r_{\texttt{NewDFS}}$: for $v = r$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \uparrow \wedge \texttt{MayDropUp}(v) \longrightarrow \texttt{NewToken}(v)$$

Let us now consider the actions taken by children of such a node, *i.e.* a node v which has one parent of the same color and its variable tok $=\uparrow$.

Rule $\mathbb{R}_{\texttt{ReplayUp}}$ If node v only has parents that are sending the token higher, *i.e.* no other parent which is still involved in a token circulation, then v can reset its variable $\texttt{play}_v = \texttt{P}$ to be ready for the next circulation round.

$$\texttt{ReplayW}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v) : \texttt{tok}_p = \uparrow \wedge \forall p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p \in \{\perp, \uparrow\} \tag{5.31}$$

Note that v only considers its parent with the same color, since the situation which we want to prevent is livelock caused by the oscillation between two tokens. Therefore, v can join a negotiation of the other color by this action.

$\mathbb{R}_{\texttt{ReplayUp}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{ReplayW}(v) \longrightarrow \texttt{Replay}(v)$$

Rule $\mathbb{R}_{\texttt{Fake}}$ If v has, in addition to some parents that are sending the token higher, other parents which have not terminated their token circulation, it must update its variable $\texttt{play}_v = \texttt{F}$.

$$\texttt{FakeReplay}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p = \uparrow \wedge \neg\texttt{ReplayW}(v) \tag{5.32}$$

$\mathbb{R}_{\texttt{Fake}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{FakeReplay}(v) \longrightarrow \texttt{play}_v := \texttt{F}$$

Rule $\mathbb{R}_{\texttt{ReWin}}$ Finally, we must design a rule to cancel the effect of updating one's variable at F, to reset it at W. Without this rule, the predicate WaitSib could be infinitely false on one parent of v, and thus the token could not be sent higher. This rule can be activated only when the node does not have any parent with the same color such that tok $=\uparrow$.

Up to this rule, all the guards of the rules require that the node is near a token. For consistency, we also require proximity of a token for this rule, but it is not actually necessary.

$$\texttt{StopFaking}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v) : \texttt{tok}_v \neq \perp \wedge \forall p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p \neq \uparrow \tag{5.33}$$

$\mathbb{R}_{\texttt{ReWin}}$: for all $v \in V \setminus \{r\}$

$$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{F} \wedge \texttt{StopFaking}(v) \longrightarrow \texttt{play}_v := \texttt{W}$$

Figure 5.10 present the process of how a token is returned to a parent.

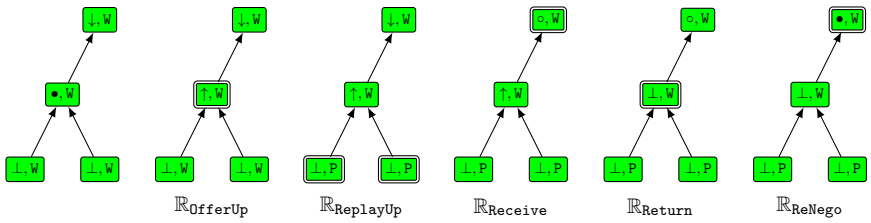

Figure 5.11: Execution of rules for returning the token back to one parent. The activated node is the one with its line doubled

5.3.4.7 Complete Algorithm

Algorithm 2 presents all the rules of our algorithm.

Algorithm 2: Token Circulation algorithm

$\forall v$	$\mathbb{E}_{\texttt{TrustChild}}$	:	$\texttt{Er}(v)$	$\longrightarrow \texttt{tok}_v :=\downarrow$
$\forall v$	$\mathbb{R}_{\texttt{OfferUp}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \neg\texttt{WaitSib}(v) \wedge (\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \in \{\texttt{W},\texttt{F}\})$	$\longrightarrow \texttt{tok}_v :=\uparrow$
$\forall v$	$\mathbb{R}_{\texttt{Give}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \texttt{Give}(v)$	$\longrightarrow \texttt{tok}_v :=\downdownarrows$
$v \neq r$	$\mathbb{R}_{\texttt{FakeWin}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \neg\texttt{OkNeg}(v)$	$\longrightarrow \texttt{FakeWin}(v)$
$v \neq r$	$\mathbb{R}_{\texttt{Win}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{WinNeg}(v)$	$\longrightarrow \texttt{Win}(v)$
$v \neq r$	$\mathbb{R}_{\texttt{Lose}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{LoseNeg}(v)$	$\longrightarrow \texttt{play}_v := \texttt{L}$
$v \neq r$	$\mathbb{R}_{\texttt{NewBit}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{P} \wedge \texttt{AnswerNeg}(v)$	$\longrightarrow \texttt{Announce}(v)$
$\forall v$	$\mathbb{R}_{\texttt{maxPos}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \lvert\texttt{Players}(v)\rvert \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{NextPlay}(v)$	$\longrightarrow \texttt{MaxPos}(v)$
$\forall v$	$\mathbb{R}_{\texttt{NewPh}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bullet \wedge \lvert\texttt{Players}(v)\rvert \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{PhComplete}(v)$	$\longrightarrow \texttt{PhasePlus}(v)$
$\forall v$	$\mathbb{R}_{\texttt{NewPlay}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_u =\downdownarrows \wedge(\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{P} \vee \texttt{tok}_u = \bigstar)$	$\longrightarrow \texttt{tok}_v :=\circlearrowleft$
$v \neq r$	$\mathbb{R}_{\texttt{ReplayD}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{L} \wedge \texttt{ReplayL}(v)$	$\longrightarrow \texttt{Replay}(v)$
$\forall v$	$\mathbb{R}_{\texttt{Drop}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v =\circlearrowleft \wedge(\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{L} \vee \texttt{tok}_u = \bigstar)$	$\longrightarrow \texttt{tok}_v :=\downarrow$
$\forall v$	$\mathbb{R}_{\texttt{Nego}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bigstar \wedge (\forall p \in \mathcal{P}(v) : \texttt{tok}_u \notin \{\downdownarrows,\circlearrowleft\})$	$\longrightarrow \texttt{StartNego}(v)$
$\forall v$	$\mathbb{R}_{\texttt{Receive}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v =\downarrow \wedge(\forall u \in \mathcal{C}(v), \texttt{tok}_u \in \{\perp,\uparrow\})$	$\longrightarrow \texttt{tok}_v := \circ$
$\forall v$	$\mathbb{R}_{\texttt{ReNego}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \circ \wedge \texttt{Leaf}(v) \wedge \forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{L}$	$\longrightarrow \texttt{StartNego}(v)$
$v \neq r$	$\mathbb{R}_{\texttt{Return}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v =\uparrow \wedge\texttt{MayDropUp}(v)$	$\longrightarrow \texttt{Drop}(v)$
r	$\mathbb{R}^r_{\texttt{NewDFS}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v =\uparrow \wedge\texttt{MayDropUp}(v)$	$\longrightarrow \texttt{NewToken}(v)$
$v \neq r$	$\mathbb{R}_{\texttt{ReplayUp}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{ReplayW}(v)$	$\longrightarrow \texttt{Replay}(v)$
$v \neq r$	$\mathbb{R}_{\texttt{Fake}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{FakeReplay}(v)$	$\longrightarrow \texttt{play}_v := \texttt{F}$
$v \neq r$	$\mathbb{R}_{\texttt{ReWin}}$	:	$\neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{F} \wedge \texttt{StopFaking}(v)$	$\longrightarrow \texttt{play}_v := \texttt{W}$

Figures 5.12 and 5.13 present the scheme of an execution of our algorihtm. These are two dual visions of the same phenomenon. The first one focuses on the values of the different variables of one node v, and the second on the executions of the rules on one node v.

5.4 Proof of the Correctness of our Algorithm

In this section, we prove that Algorithm 2 is a self-stabilizing algorithm for the token circulation problem, as it is defined in Specification 5.1. We also prove that this algorithm is optimal in terms of memory, *i.e.* that any algorithm which solves $\mathcal{TC}$ in $\mathcal{S}_{\mathsf{PU}}$ requires $\Omega(\log\log n)$ bits per nodes, even under less general hypothesis than ours.

This section is subdivided in four subsections. In Section 5.4.1 we establish that maximal

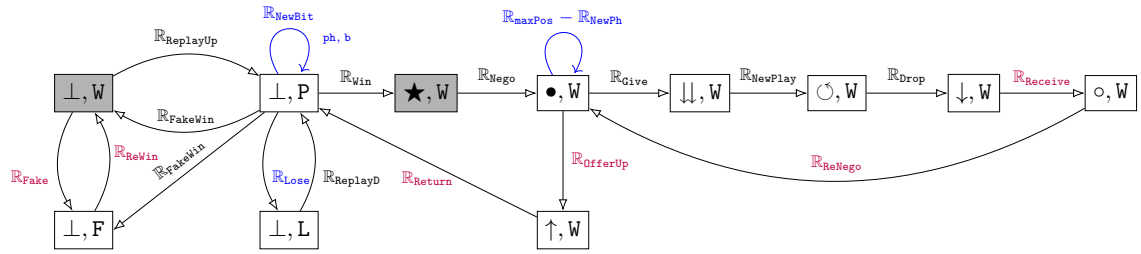

Figure 5.12: Chain of the states taken by a node during the execution of the agorithm. The update of $\mathtt{c}_v$ is represented by color gray. Rules in blue corresponds to the negotiation. Rules in red correspond to the sending of a token from a child to a parent. Rules in black correspond to the sending of a token from a parent to a child.

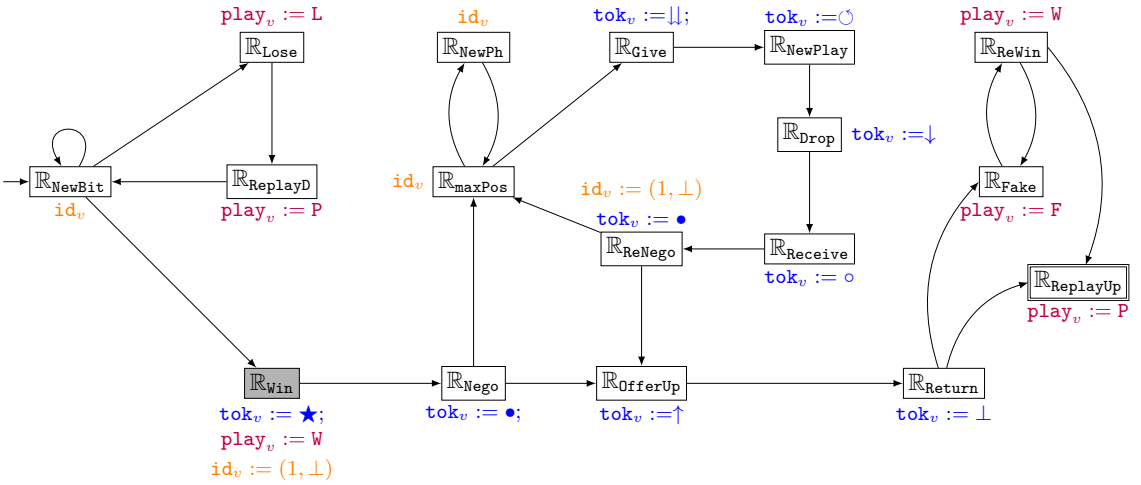

Figure 5.13: Chain of execution of the different rules on one node, and effects of these rules. The update of $\mathtt{c}_v$ is represented by color gray.

executions of our algorithm are infinite. Formally, we prove that in any configuration, there is at least one enabled node. This work is static in the sense that we do not need to consider an execution of the algorithm. It boils down to a syntactical analysis of the guards of the rules of our algorithm (Algorithm 2).

In Section 5.4.2 we establish that no token can circulate indefinitely in the $\mathtt{D}^o$, unless it regularly reaches the root and is reset by the execution of $\mathbb{R}^r_{\mathtt{NewDFS}}$. This guarantees that a token which is not anchored at the root of the $\mathtt{D}^o$ either vanishes, or is eventually not activated. Using the previous, we also establish that there are an infinite number of circulation rounds in maximal executions of our algorithm. This guarantees that our algorithm satisfies Condition 2 of Specification 5.1.

In Section 5.4.3 we use the previous results to establish that the token anchored at the root circulates fairly in deeper and deeper parts of the $\mathtt{D}^o$. Thus, even if there are other tokens in the network, they will eventually be reached and destroyed by the legitimate token anchored at the root. Therefore we prove in this section that, at some point, the execution of our algorithm satisfies Conditions 1 and 3 of Specification 5.1, which means that it is a self-stabilizing algorithm for $\mathcal{TD}$.

Finally, in Section 5.4.4 we use the results of Chapter 3 to prove that the space complexity of our algorithm is optimal.

In order to prove that our algorithm satisfies conditions of Specification 5.1, we must first define the predicates T and R involved in this specification. One node v is considered to hold the token if it has a non-empty value for its variable $\mathtt{tok}_v$, and if none of its children holds the token. One node v is considered to have access to the resource if it is in the state where it just received the token from its parent, which corresponds to $\mathtt{tok}_v = \bigstar$.

Definition 5.12 (*Predicates for our Token Circulation Algorithm*)

The resolution of $\mathcal{TD}$ by our algorithm will be proven under the following specification predicates:

$$T(v,\gamma) \equiv \mathtt{tok}_v^\gamma \neq \bot \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma = \bot$$
$$R(v,\gamma) \equiv T(v,\gamma) \wedge \mathtt{tok}_v^\gamma = \bigstar$$

Definition 5.13 (*Reset Points*)

Let $\epsilon = \gamma_0 \to \cdots$ be an execution. We say that γ_i is a reset point of ϵ if r executes $\mathbb{R}^r_{\mathtt{NewDFS}}$ during $\gamma_{i-1} \to \gamma_{i+1}$. We say that two reset points γ_i and γ_j are consecutive if $i < j$ and $\forall k \in [i+1, j+1]$, γ_k is not a reset point.

Theorem 5.1 (*Circulation Rounds*)

Let $\epsilon = \gamma_0 \to \cdots$ be an execution. Let us denote by $t_0 \leq t_1 \leq \cdots$ all the reset points of ϵ, such that $\forall i \geq 0, \gamma_{t_i}$ and $\gamma_{t_{i+1}}$ are consecutive.
$\forall i \geq 0, \gamma_{t_1} \to \cdots \to \gamma_{t_{i+1}-1}$ is a circulation round.

Proof: For any computing step $cs = \gamma \to \gamma'$, we have $\neg R^\gamma(r) \wedge R^{\gamma'}(r)$ if and only if r executes $\mathbb{R}^r_{\mathtt{NewDFS}}$, which is equivalent to γ' being a reset point. ■

5.4.1 Liveness

Recall that we denote by $\mathcal{A}^e(\gamma)$ the set of the enabled nodes for $\mathcal{A}$ in configuration γ. In this section, we prove that $\forall \gamma \in \Gamma, \mathcal{A}^e(\gamma) \neq \emptyset$.

The proof is subdivided in several lemmas. In each lemma we work under the hypothesis that some value for $\mathtt{tok}_v$, on one node v, is present in γ, and in each case we prove that

there exists at least one enabled node, in γ. Not all the lemmas are independent: it happens that to prove a lemma, we simply prove that there exists one node u with another value of $\texttt{tok}_u$, value which has already been dealt with in a previous lemma.

One lemma is even proven by induction, the lemma which considers as a hypothesis the presence of a node v such that $\texttt{tok}_v =\downarrow$. Such nodes can actually pretty far from the actual token, and thus to find an enabled node, we may have to descend a branch of the $\texttt{D}^o$, until we fall on a leaf, or on a node with a different value for $\texttt{tok}_v$.

More precisely, we prove that either we can find an enabled node, or there exists one other node u such that $\texttt{tok}_u =\downarrow$, but u is deeper than v in the DODAG. Since DODAGs are finite, and acyclic, there can only be a finite number of times where the second hypothesis is the right one, and therefore this proves that there exists at least one enabled node.

The proofs are basically reasoning by case disjunction, which forms decision trees of variable depth. We must guarantee that at the end of each branch *i.e.* at each leaf, we find one enabled node.

Since it is basically syntactical analysis, we only present the different decision trees, which contain all the elements of the proofs, and are easier to parse than the written version of the proofs.

In each proof, we also provide a reasoning toolbox, to remind the reader of the detail of the predicates considered in the reasoning.

Lemma 5.1

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\texttt{tok}_v = \uparrow$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.14 ■

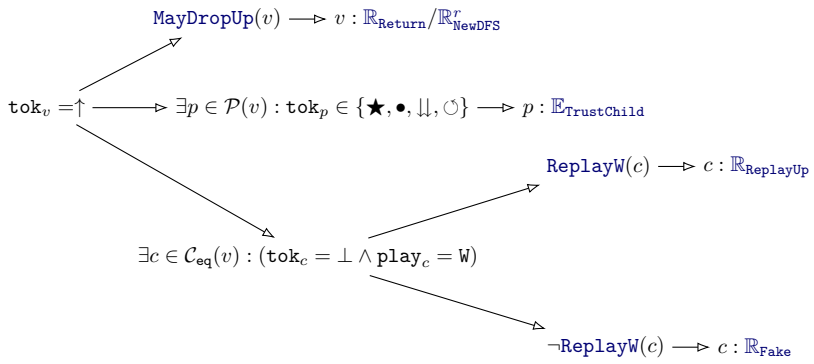

Figure 5.14: Proof of Lemma 5.1

Tools Lemma 5.1

$$
\begin{array}{llll}
\forall v & \mathbb{E}_{\texttt{TrustChild}} : & \texttt{Er}(v) & \longrightarrow \texttt{tok}_v := \downarrow \\
v \neq r & \mathbb{R}_{\texttt{Return}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \uparrow \wedge \texttt{MayDropUp}(v) & \longrightarrow \texttt{Drop}(v) \\
r & \mathbb{R}^r_{\texttt{NewDFS}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \uparrow \wedge \texttt{MayDropUp}(v) & \longrightarrow \texttt{NewToken}(v) \\
v \neq r & \mathbb{R}_{\texttt{ReplayUp}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bot \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{ReplayW}(v) & \longrightarrow \texttt{Replay}(v) \\
v \neq r & \mathbb{R}_{\texttt{Fake}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bot \wedge \texttt{play}_v = \texttt{W} \wedge \texttt{FakeReplay}(v) & \longrightarrow \texttt{play}_v := \texttt{F}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{Er}(v) & \equiv & \Big(\texttt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \neq \bot\Big) \\
& \vee & \Big(\texttt{tok}_v \in \{\Downarrow, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \bigstar\}\Big) \\
& \vee & \Big(\texttt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \uparrow\}\Big) \\
& \longrightarrow & \texttt{tok}_v := \downarrow
\end{array}
$$

$$
\texttt{MayDropUp}(v) \equiv \left\{ \begin{array}{ll} & \forall p \in \mathcal{P}(v), \texttt{tok}_p \in \{\bot, \downarrow, \uparrow, \circ\} \\ \wedge & \forall u \in \mathcal{C}_{\texttt{eq}}(v), (\texttt{tok}_u = \bot \Rightarrow \texttt{play}_u \in \{\texttt{P}, \texttt{L}, \texttt{F}\}) \end{array} \right.
$$

$$
\texttt{ReplayW}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v) : \texttt{tok}_p = \uparrow \wedge \forall p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p \in \{\bot, \uparrow\}
$$

$$
\texttt{FakeReplay}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v), \texttt{tok}_p = \uparrow \wedge \neg\texttt{ReplayW}(v)
$$

Lemma 5.2

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\mathtt{tok}_v = \downdownarrows$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.15 ■

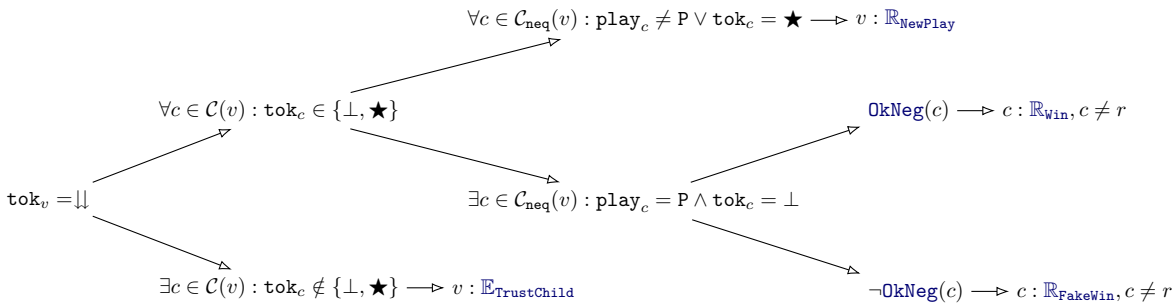

Figure 5.15: Proof of Lemma 5.2

Tools Lemma 5.2

$$
\begin{array}{llll}
\forall v & \mathbb{E}_{\mathtt{TrustChild}} : & \mathtt{Er}(v) & \longrightarrow \mathtt{tok}_v := \downarrow \\
\forall v & \mathbb{R}_{\mathtt{NewPlay}} : & \neg\mathtt{Er}(v) \wedge \mathtt{tok}_u = \downdownarrows \wedge (\forall u \in \mathcal{C}_{\mathtt{neq}}(v) : \mathtt{play}_u \neq \mathtt{P} \vee \mathtt{tok}_u = \bigstar) & \longrightarrow \mathtt{tok}_v := \circlearrowleft \\
\forall v \neq r & \mathbb{R}_{\mathtt{Win}} : & \neg\mathtt{Er}(v) \wedge \mathtt{tok}_v = \perp \wedge \mathtt{play}_v = \mathtt{P} \wedge \mathtt{WinNeg}(v) & \longrightarrow \mathtt{Win}(v) \\
\forall v \neq r & \mathbb{R}_{\mathtt{FakeWin}} : & \neg\mathtt{Er}(v) \wedge \mathtt{tok}_v = \perp \wedge \mathtt{play}_v = \mathtt{P} \wedge \neg\mathtt{OkNeg}(v) & \longrightarrow \mathtt{FakeWin}(v)
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{Er}(v) & \equiv & \big(\mathtt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \neq \perp\big) \\
& \vee & \big(\mathtt{tok}_v \in \{\downdownarrows, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\perp, \bigstar\}\big) \\
& \vee & \big(\mathtt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\perp, \uparrow\}\big) \\
& \longrightarrow & \mathtt{tok}_v := \downarrow
\end{array}
$$

$$\mathtt{WinNeg}(v) \equiv \mathtt{OkNeg}(v) \wedge \exists p \in \mathcal{P}_{\mathtt{neq}}(v) : \mathtt{tok}_p = \downdownarrows$$

$$\mathtt{OkNeg}(v) \equiv |\mathtt{ParNeg}(v)| \leq 1 \vee (|\mathtt{ParNeg}(v)| = 2 \wedge \{\mathtt{tok}_p \mid p \in \mathtt{ParNeg}(v)\} \in \{\{\downdownarrows, \bigstar\}, \{\circlearrowleft, \bigstar\}\})$$

$$\mathtt{ParNeg}(v) = \{p \in \mathcal{P}(v) \mid \mathtt{tok}_p \notin \{\perp, \uparrow, \downarrow\}\}$$

Lemma 5.3

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\mathtt{tok}_v = \bullet$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figures 5.16 and 5.17 ■

Tools Lemma 5.3

$$
\begin{array}{llll}
\forall v & \mathbb{E}_{\texttt{TrustChild}} : & \texttt{Er}(v) & \longrightarrow \mathtt{tok}_v := \downarrow \\
\forall v & \mathbb{R}_{\texttt{maxPos}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bullet \wedge |\texttt{Players}(v)| \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{NextPlay}(v) & \longrightarrow \texttt{MaxPos}(v) \\
\forall v & \mathbb{R}_{\texttt{NewPh}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bullet \wedge |\texttt{Players}(v)| \geq 2 \wedge \texttt{Synch}(v) \wedge \texttt{PhComplete}(v) & \longrightarrow \texttt{PhasePlus}(v) \\
\forall v & \mathbb{R}_{\texttt{Give}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bullet \wedge \texttt{Give}(v) & \longrightarrow \mathtt{tok}_v := \Downarrow \\
\forall v & \mathbb{R}_{\texttt{OfferUp}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bullet \wedge \neg\texttt{WaitSib}(v) \wedge (\forall u \in \mathcal{C}_{\texttt{neq}}(v) : \mathtt{play}_u \in \{\mathtt{W}, \mathtt{F}\}) & \longrightarrow \mathtt{tok}_v := \uparrow \\
v \neq r & \mathbb{R}_{\texttt{NewBit}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{P} \wedge \texttt{AnswerNeg}(v) & \longrightarrow \texttt{Announce}(v) \\
v \neq r & \mathbb{R}_{\texttt{Lose}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{P} \wedge \texttt{LoseNeg}(v) & \longrightarrow \mathtt{play}_v := \mathtt{L} \\
v \neq r & \mathbb{R}_{\texttt{Win}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{P} \wedge \texttt{WinNeg}(v) & \longrightarrow \texttt{Win}(v) \\
v \neq r & \mathbb{R}_{\texttt{FakeWin}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{P} \wedge \neg\texttt{OkNeg}(v) & \longrightarrow \texttt{FakeWin}(v) \\
v \neq r & \mathbb{R}_{\texttt{ReWin}} : & \neg\texttt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{F} \wedge \texttt{StopFaking}(v) & \longrightarrow \mathtt{play}_v := \mathtt{W}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Er}(v) \equiv & \big(\mathtt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \neq \bot\big) \\
\vee & \big(\mathtt{tok}_v \in \{\Downarrow, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\bot, \bigstar\}\big) \\
\vee & \big(\mathtt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\bot, \uparrow\}\big) \\
\longrightarrow & \mathtt{tok}_v := \downarrow
\end{array}
$$

$$\texttt{AnswerPar}(v,p) \equiv (\mathtt{ph}_v \neq \mathtt{ph}_p) \vee (\mathtt{ph}_p = 1 \wedge \mathtt{b}_p = \bot \wedge \mathtt{b}_v = \bot)$$

$$\texttt{LosePar}(v,p) \equiv \mathtt{ph}_p = \mathtt{ph}_v \wedge \mathtt{b}_p \neq \bot \wedge \mathtt{b}_p \neq \mathtt{b}_v$$

$$\texttt{ParNeg}(v) = \{p \in \mathcal{P}(v) \mid \mathtt{tok}_p \notin \{\bot, \uparrow, \downarrow\}\}$$

$$\texttt{OkNeg}(v) \equiv |\texttt{ParNeg}(v)| \leq 1 \vee (|\texttt{ParNeg}(v)| = 2 \wedge \{\mathtt{tok}_p \mid p \in \texttt{ParNeg}(v)\} \in \{\{\Downarrow, \bigstar\}, \{\circlearrowleft, \bigstar\}\})$$

$$\texttt{WinNeg}(v) \equiv \texttt{OkNeg}(v) \wedge \exists p \in \mathcal{P}_{\texttt{neq}}(v) : \mathtt{tok}_p = \Downarrow$$

$$\texttt{LoseNeg}(v) \equiv \texttt{OkNeg}(v) \wedge \neg\texttt{WinNeg}(v) \wedge \exists p \in \mathcal{P}_{\texttt{neq}}(v) : \mathtt{tok}_p = \bullet \wedge \texttt{LosePar}(v,p)$$

$$\texttt{AnswerNeg}(v) \equiv \texttt{OkNeg}(v) \wedge \neg\texttt{WinNeg}(v) \wedge \neg\texttt{LoseNeg}(v) \wedge \exists p \in \mathcal{P}_{\texttt{neq}}(v) : \mathtt{tok}_p = \bullet \wedge \texttt{AnswerPar}(v,p)$$

$$\texttt{Players}(v) = \{u \in \mathcal{C}_{\texttt{neq}}(v) \mid \mathtt{tok}_v = \bot \wedge \mathtt{play}_u = \mathtt{P}\}$$

$$\texttt{Synch}(v) \equiv \forall u \in \texttt{Players}(v), \mathtt{ph}_u = \mathtt{ph}_v$$

$$\texttt{NextPlay}(v) \equiv \mathtt{b}_v = \bot \wedge \begin{cases} \mathtt{ph}_v = 1 \wedge \forall u \in \texttt{Players}(v), \mathtt{b}_u \neq \bot \\ \vee \\ \mathtt{ph}_v \neq 1 \wedge \exists u \in \texttt{Players}(v) : \mathtt{b}_u \neq \bot \end{cases}$$

$$\texttt{PhComplete}(v) \equiv (\mathtt{ph}_v \neq 1 \vee \mathtt{b}_v \neq \bot) \wedge \forall u \in \texttt{Players}(v), \mathtt{b}_u = \mathtt{b}_v$$

$$\texttt{Give}(v) \equiv |\texttt{Players}(v)| = 1 \vee (|\texttt{Players}(v)| = 0 \wedge \exists u \in \mathcal{C}_{\texttt{neq}}(v) : \mathtt{play}_u = \mathtt{L})$$

$$\texttt{WaitSib}(v) \equiv \exists c \in \mathcal{C}_{\texttt{eq}}(v) : \mathtt{play}_c = \mathtt{F}$$

$$\texttt{StopFaking}(v) \equiv \exists p \in \mathcal{P}_{\texttt{eq}}(v) : \mathtt{tok}_v \neq \bot \wedge \forall p \in \mathcal{P}_{\texttt{eq}}(v), \mathtt{tok}_p \neq \uparrow$$

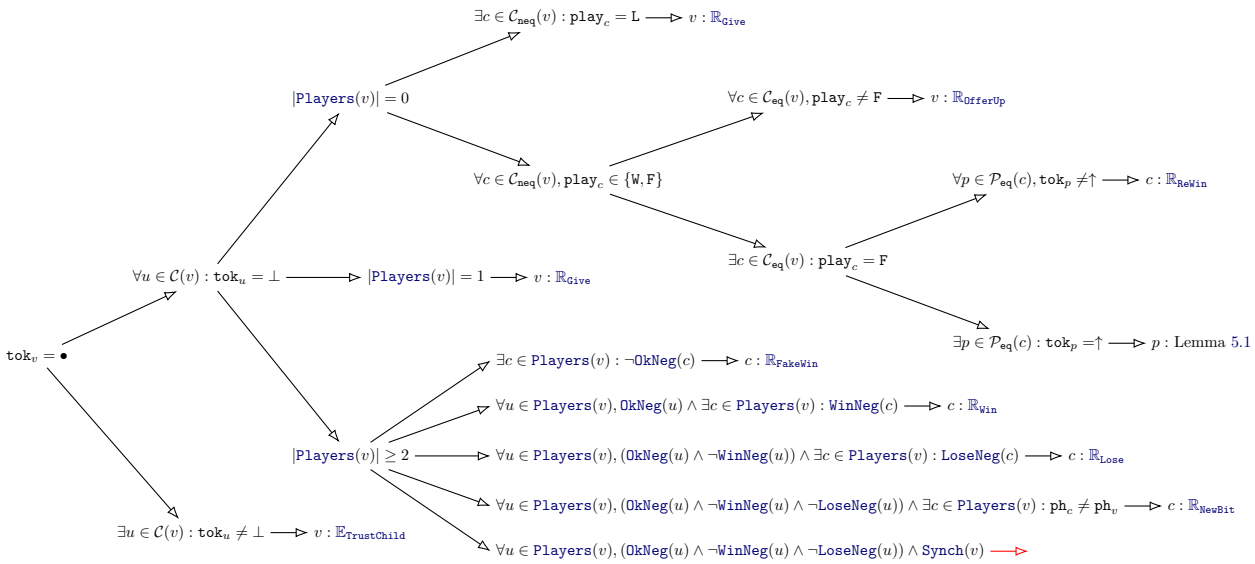

Figure 5.16: Proof of Lemma 5.3, part. 1

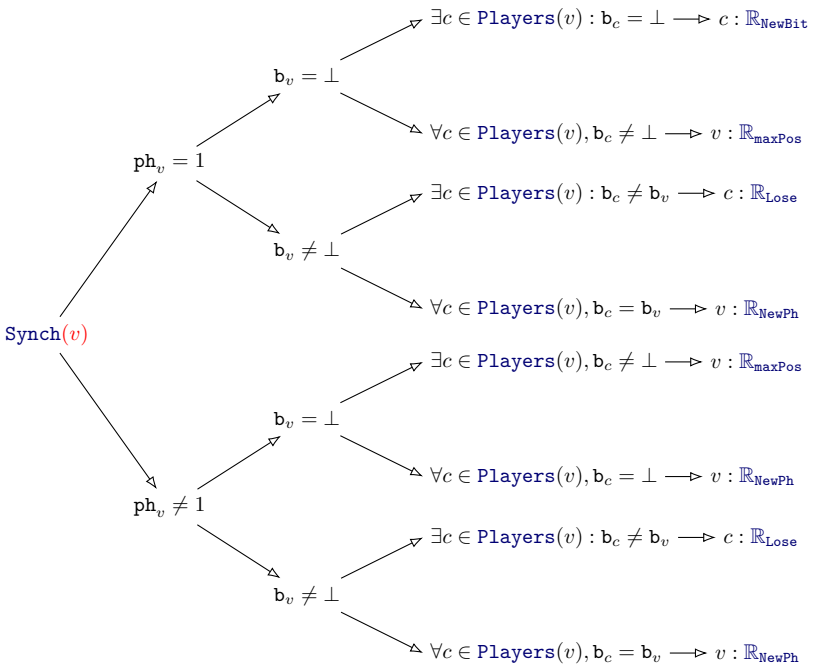

Figure 5.17: Proof of Lemma 5.3, part. 2

Lemma 5.4

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\mathtt{tok}_v = \circlearrowleft$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.18 ■

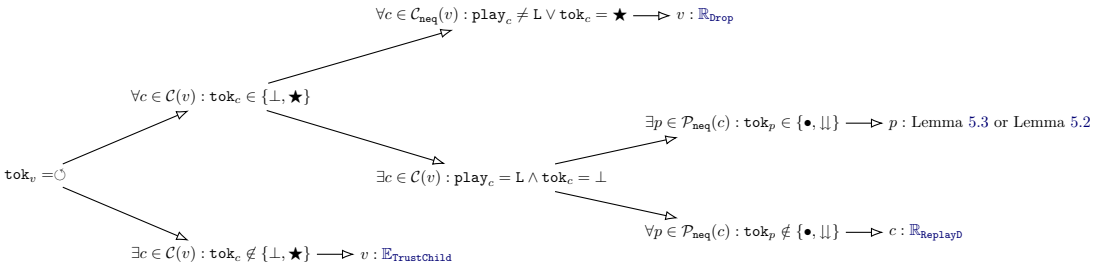

Figure 5.18: Proof of Lemma 5.4

Tools Lemma 5.4

$$
\begin{array}{llll}
\forall v & \mathbb{E}_{\mathtt{TrustChild}} : & \mathtt{Er}(v) & \longrightarrow \mathtt{tok}_v :=\downarrow \\
\forall v & \mathbb{R}_{\mathtt{Drop}} : & \neg\mathtt{Er}(v) \wedge \mathtt{tok}_v = \circlearrowleft \wedge (\forall u \in \mathcal{C}_{\mathtt{neq}}(v) : \mathtt{play}_u \neq \mathtt{L} \vee \mathtt{tok}_u = \bigstar) & \longrightarrow \mathtt{tok}_v :=\downarrow \\
\forall v \neq r & \mathbb{R}_{\mathtt{ReplayD}} : & \neg\mathtt{Er}(v) \wedge \mathtt{tok}_v = \bot \wedge \mathtt{play}_v = \mathtt{L} \wedge \mathtt{ReplayL}(v) & \longrightarrow \mathtt{Replay}(v)
\end{array}
$$

$$
\begin{array}{rcll}
\mathtt{Er}(v) & \equiv & \Big(\mathtt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \neq \bot\Big) \\
& \vee & \Big(\mathtt{tok}_v \in \{\downdownarrows, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\bot, \bigstar\}\Big) \\
& \vee & \Big(\mathtt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \mathtt{tok}_u \notin \{\bot, \uparrow\}\Big) \\
& \longrightarrow & \mathtt{tok}_v :=\downarrow
\end{array}
$$

$$\mathtt{ReplayL}(v) \equiv \forall u \in \mathcal{P}_{\mathtt{neq}}(v), \mathtt{tok}_u \notin \{\bullet, \downdownarrows\} \wedge \exists u \in \mathcal{P}_{\mathtt{neq}}(v), \mathtt{tok}_u \in \{\circlearrowleft, \circ\}$$

Lemma 5.5

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\texttt{tok}_v = \circ$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.19 ■

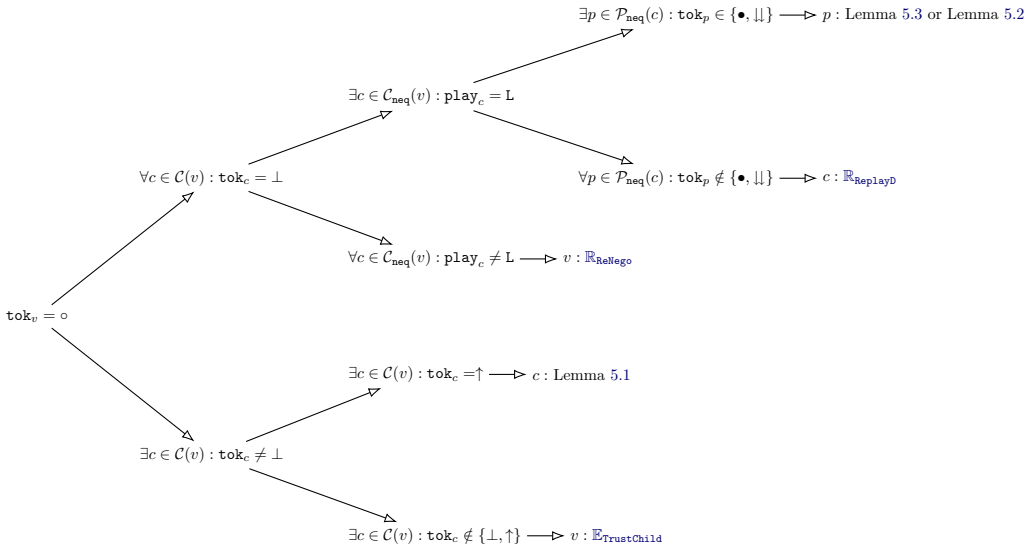

Figure 5.19: Proof of Lemma 5.5

Tools Lemma 5.5

$$
\begin{array}{llll}
\forall v & \mathbb{E}_{\texttt{TrustChild}} : & \texttt{Er}(v) & \longrightarrow \texttt{tok}_v :=\downarrow \\
\forall v & \mathbb{R}_{\texttt{ReNego}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \circ \wedge \texttt{Leaf}(v) \wedge \forall u \in \mathcal{C}_{\texttt{neq}}(v) : \texttt{play}_u \neq \texttt{L} & \longrightarrow \texttt{StartNego}(v) \\
v \neq r & \mathbb{R}_{\texttt{ReplayD}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \perp \wedge \texttt{play}_v = \texttt{L} \wedge \texttt{ReplayL}(v) & \longrightarrow \texttt{play}_v := \texttt{P}
\end{array}
$$

$$
\begin{array}{rcll}
\texttt{Er}(v) & \equiv & \big(\texttt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \neq \perp\big) \\
& \vee & \big(\texttt{tok}_v \in \{\downdownarrows, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\perp, \bigstar\}\big) \\
& \vee & \big(\texttt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\perp, \uparrow\}\big) \\
& \longrightarrow & \texttt{tok}_v :=\downarrow
\end{array}
$$

$$\texttt{ReplayL}(v) \equiv \forall u \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_u \notin \{\bullet, \downdownarrows\} \wedge \exists u \in \mathcal{P}_{\texttt{neq}}(v), \texttt{tok}_u \in \{\circlearrowleft, \circ\}$$

Lemma 5.6

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\texttt{tok}_v = \bigstar$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.20 ■

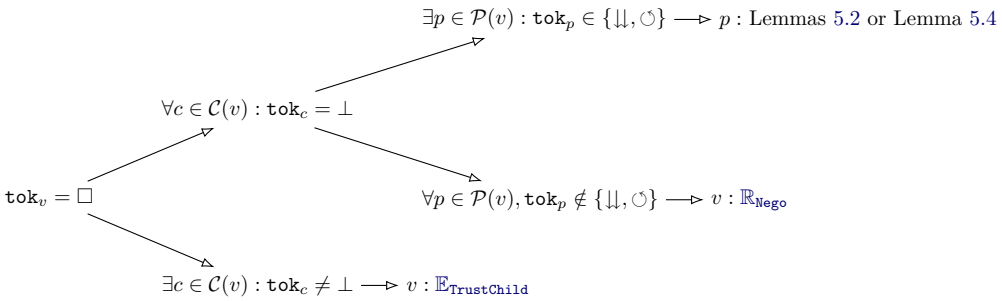

Figure 5.20: Proof of Lemma 5.6

Tools Lemma 5.6

$$\begin{array}{lll} \forall v & \mathbb{E}_{\texttt{TrustChild}} : & \texttt{Er}(v) \longrightarrow \texttt{tok}_v :=\downarrow \\ v \neq r & \mathbb{R}_{\texttt{Nego}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \bigstar \wedge (\forall p \in \mathcal{P}(v) : \texttt{tok}_u \notin \{\Downarrow, \circlearrowleft\}) \longrightarrow \texttt{StartNego}(v) \end{array}$$

$$\begin{array}{rcll} \texttt{Er}(v) & \equiv & \Big(\texttt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \neq \bot\Big) \\ & \vee & \Big(\texttt{tok}_v \in \{\Downarrow, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \bigstar\}\Big) \\ & \vee & \Big(\texttt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \uparrow\}\Big) \\ & \longrightarrow & \texttt{tok}_v :=\downarrow \end{array}$$

Lemma 5.7

Let $\gamma \in \Gamma$. If there exists $v \in V$ such that $\texttt{tok}_v = \downarrow$ in γ then $\mathcal{A}^e(\gamma) \neq \emptyset$.

Proof: The proof is given in Figure 5.21 ■

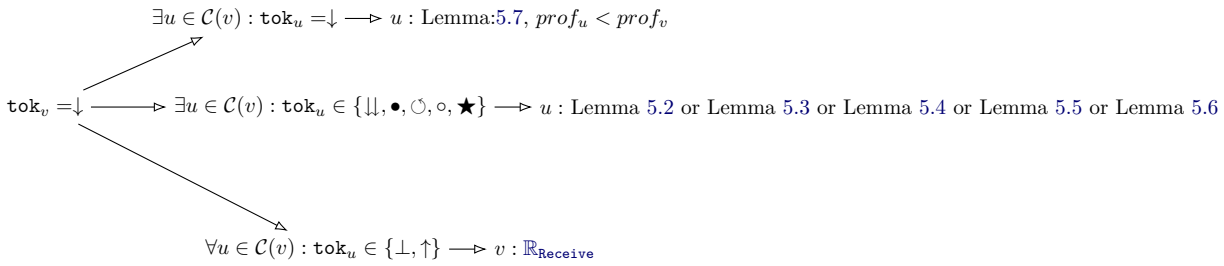

Figure 5.21: Proof of Lemma 5.7

Tools Lemma 5.7

$$\begin{array}{llll} \forall v & \mathbb{E}_{\texttt{TrustChild}} : & \texttt{Er}(v) & \longrightarrow \texttt{tok}_v := \downarrow \\ \forall v & \mathbb{R}_{\texttt{Receive}} : & \neg\texttt{Er}(v) \wedge \texttt{tok}_v = \downarrow \wedge (\forall u \in \mathcal{C}(v), \texttt{tok}_u \in \{\bot, \uparrow\}) & \longrightarrow \texttt{tok}_v := \circ \end{array}$$

$$\begin{array}{rcll} \texttt{Er}(v) & \equiv & \big(\texttt{tok}_v \in \{\bullet, \bigstar\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \neq \bot\big) \\ & \vee & \big(\texttt{tok}_v \in \{\Downarrow, \circlearrowleft\} \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \bigstar\}\big) \\ & \vee & \big(\texttt{tok}_v = \circ \wedge \exists u \in \mathcal{C}(v) : \texttt{tok}_u \notin \{\bot, \uparrow\}\big) \\ & \longrightarrow & \texttt{tok}_v := \downarrow \end{array}$$

Theorem 5.2

$\forall \gamma \in \Gamma, \mathcal{A}^e(\gamma) \neq \emptyset$

Proof: Let $\gamma \in \Gamma$ be a configuration. Since $\texttt{tok}_r \in \{\uparrow, \Downarrow, \bullet, \circlearrowleft, \circ, \bigstar, \downarrow\}$, one of the following lemmas applies: 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7. ■

5.4.2 Progress

In this section we formally establish, at once, two crucial properties for the validity of our algorithm. The first property is that if a token is not anchored at the root of the $\texttt{D}^o$ (we will give later a proper definition of the anchor of a token), then it can only circulate a finite number of times before being blocked, or deleted. This means that no scheduler, even unfair, can create an execution in which the token held by the root is never activated.

The second property is that circulation rounds (see Section 5.2.5) are always finite. We actually prove that the token held by the root can only circulate a finite number of times before it reaches the root, and the root executes $\mathbb{R}^r_{\texttt{NewDFS}}$.

The combination of these two properties, and of the liveness established in Theorem 5.2 leads to the guarantee that there is an infinity of finite circulation rounds in any maximal execution of our algorithm. We will prove in the next section that, eventually, the circulation of the token is fair.

To prove both properties, we define a potential function, which associates a positive value (a *weight*) to each token in any configuration. Then, we establish that any computing step decreases the weight of the token, unless the token is reset by an execution of $\mathbb{R}^r_{\texttt{NewDFS}}$. This establishes that a token cannot circulate indefinitely, since any step it takes makes a positive quantity decrease, unless this token is anchored at the root and there are an infinity of new circulation rounds.

In this section we provide a formal proof of these properties. Let us first have an overview of this proof. In a corrupted initial configuration, the network may contain several tokens, which may have common ancestors, and also have several ancestors. To deal with the diversity of possible configurations, we define a new object to reason on, circulation DODAGs, denoted by $\texttt{CD}^o$, for each token. Then, we define a weight function, which associates a weighted DODAG, denoted by $\texttt{WD}^o$, to each $\texttt{CD}^o$. We finally prove that each time a node of a $\texttt{CD}^o$ is activated, the corresponding $\texttt{WD}^o$ decreases. According to Theorem 5.2, there is at least one enabled node in each configuration, which guarantees that $\texttt{WD}^o$'s actually decrease in maximal executions.

In Section 5.4.2.1 we present the inherent difficulties to overcome for proving our results, and show how $\texttt{CD}^o$'s and $\texttt{WD}^o$'s are suitable solutions. In Section 5.4.2.2 we formally define $\texttt{CD}^o$'s and establish basic properties on them. In Section 5.4.2.3 we show how to define the weight function we use, and introduce the concept of $\texttt{WD}^o$'s. Then, in Sections 5.4.2.4, 5.4.2.5, 5.4.2.6, 5.4.2.7, and 5.4.2.8 we define the actual weight function, step by step, and prove properties on it at each step. Finally, we establish our main theorems in Section 5.4.2.9.

5.4.2.1 Difficulties to Overcome, Circulation DODAG

To associate a decreasing weight to a token, the main difficulty to overcome is that during one circulation, the token shifts a lot, upward and downward alternately. Thus, the information of whether the token is very close to the end of the circulation round, or still far, is not local. This information strongly depends, for example, on which branches have already been visited by the token, on the number of unvisited children the different nodes have, on whether one error will be raised, *etc.* As a result, we cannot focus only on the token if we want to succeed in estimating how many steps it still has to execute before it ends its circulation.

One first condition for our potential function to be consistent is that it takes into account the variables of the node holding the token, but also information relative to some ancestors of that node. In practice, only the ancestors which are somehow pointing to this token will be considered.

Yet, it might happen that one node in the ancestry of the token holder has several children involved in a token circulation. If this is due to a triangle topology, *i.e.* if one node has two parents, one being the parent of the other one, then considering the ascendants of the token is sufficient. But this could also be a situation in which one node has two independent token circulations below it. Our algorithm does not create such situations, but they might exist due to an incorrect initial configuration. In such a situation, our previous definition is not fully satisfactory since it does not highlight the fact that those two token circulations are, though independent, intertwined.

To embrace that complexity, we define an *ad hoc* object, called a Circulation DODAG, denoted $\texttt{CD}^o$, which has a $\texttt{D}^o$ structure. That new object follows the same idea as the upwards branch starting at the token, but we reverse that idea. We first look for nodes with no ancestry (called *anchor*), that is to say nodes v such that $\texttt{tok}_v \neq \bot \wedge \forall p \in \mathcal{P}(v), \texttt{tok}_p \in \{\bot, \uparrow\}$. Then, we go down from each anchor to all its descendants, following nodes u such that $\texttt{tok}_u \neq \bot$. Since nodes with $\texttt{tok}_u = \uparrow$ can only be updated by $\mathbb{R}_{\texttt{Return}}$, with effect

$\texttt{tok}_u := \bot$, we include them in our structures, but not their potential descendants: such nodes are necessarily leaves of the $\texttt{CD}^o$. Note that if the initial configuration is inconsistent, it may contain several $\texttt{CD}^o$'s. In Section 5.4.2.2 we formally define $\texttt{CD}^o$'s and we establish some basic properties on $\texttt{CD}^o$'s.

Our goal is now to associate a finite positive quantity, a weight, to any $\texttt{CD}^o$ in any configuration. Such an association will be called a weight function. A weight function is said admissible if any activation of the $\texttt{CD}^o$ results in a decrease of the weight of the $\texttt{CD}^o$. Since there does not exist an infinite decreasing sequence of positive integers, it comes that any $\texttt{CD}^o$ has a finite lifetime. Actually, since our algorithm of token circulation does not terminate, there necessarily exists one rule that does not respect that specification. The rule $\mathbb{R}^r_{\texttt{NewDFS}}$ is the only rule that increases the weight of a $\texttt{CD}^o$, as it refreshes the token by switching its color. Since only the root of the $\texttt{D}^o$ can execute $\mathbb{R}^r_{\texttt{NewDFS}}$, we guarantee that any other $\texttt{CD}^o$, anchored at another node than the root, has a finite lifetime.

Unfortunately, associating a simple integer to a $\texttt{CD}^o$ does not allow to take into account the diversity of situations that we must consider. The appropriate quantity to reason on has itself a $\texttt{D}^o$ structure, which has the same topology as the $\texttt{CD}^o$ to which it is associate by the weight function. Such $\texttt{D}^o$'s are called weighted DODAGs and are denoted $\texttt{WD}^o$.

More precisely, a $\texttt{WD}^o$ is a 5-tuple of $\texttt{D}^o$'s. Each component of the tuple deals with one aspect of the algorithm. The order in which the different components are set is crucial. Indeed, it happens that some computing step makes one of the components increase. We prove that this can be only if one component with higher priority decreases at the same time.

- The first component of the 5-tuple evaluates the number of unvisited children of each node of the $\texttt{CD}^o$.
- The second component is dedicated to the numbers of errors in the network.
- The third component is dedicated to the updates of variable `tok` on each node, as depicted in Figure 5.7.
- The fourth component is dedicated to the variable `play` on the children of each node of the $\texttt{CD}^o$.
- And the last component is dedicated to the negotiation process near nodes with $\texttt{tok} = \bullet$.

5.4.2.2 Circulation DODAGs

In this subsection, we define tools to capture the behavior of the token circulation, and especially Circulation DODAGs, $\texttt{CD}^o$'s. We prove in particular that the number of $\texttt{CD}^o$'s can never increase, which is a very desirable property to have to establish that the token is eventually unique.

Remember that we denote by $\texttt{var}_v^\gamma$ the state of variable `var` for the node v in configuration γ, and by $P^\gamma(v)$ the values of predicate P on node v in configuration γ.

We first define the anchors of our $\texttt{CD}^o$'s. The anchors are nodes v that are involved in a token circulation, *i.e.* $\texttt{tok}_v \neq \bot$, and which do not have any parent in this token circulation. Recall that nodes u such that $\texttt{tok}_u = \uparrow$ cannot be considered as parents, since they can only execute rule $\mathbb{R}_{\texttt{Return}}$, which makes them quit the circulation.

Definition 5.14 (*Anchor*)

Let $\gamma \in \Gamma$ be a configuration. Node $a \in V$ is an anchor in γ if

- $\texttt{tok}_a^\gamma \neq \bot$, *and*
- $\forall u \in \mathcal{P}(a), \texttt{tok}_u^\gamma \in \{\bot, \uparrow\}$.

We denote by $\mathbb{A}(\gamma)$ *the set of all anchors at* γ*, and by* $\mathbb{A}^*(\gamma)$ *the set of all anchors except the root.*

Remark 5.4

The root of G *is an anchor in any configuration* γ*.*

Now that we have defined anchors, we can define the nodes of the $\texttt{CD}^o$ anchored at one particular anchor. The nodes of the $\texttt{CD}^o$ anchored at a are all the nodes which can reach a by paths corresponding to a actual token circulation. In other words, we only consider paths in which all nodes have a value $\texttt{tok} \neq \bot$, and such that all the nodes apart from the source have a value $\texttt{tok} \neq \uparrow$. Indeed, nodes u such that $\texttt{tok}_u = \uparrow$ are necessarily leaves of $\texttt{CD}^o$'s.

Definition 5.15 (*Nodes of a Circulation DODAG*)

Let us consider a configuration $\gamma \in \Gamma$ *and an anchor* $a \in \mathbb{A}(\gamma)$*. We define by induction the nodes of the Circulation DODAG (*$\texttt{CD}^o$*) anchored at* a *in* γ*, and denote by* V_a^γ *as the smallest set which contains the node* a *and such that:*

$$v \in V_a^\gamma \iff \begin{cases} \texttt{tok}_v^\gamma \neq \bot \\ \exists u \in \mathcal{P}(v) : \big(u \in V_a^\gamma \wedge \texttt{tok}_u^\gamma \neq \uparrow\big). \end{cases}$$

We now give a definition of $\texttt{CD}^o$ and we present an example with several $\texttt{CD}^o$'s in one $\texttt{D}^o$ in Figure 5.22.

Definition 5.16 (*Circulation DODAG*)

Let us consider a configuration $\gamma \in \Gamma$ *and an anchor* $a \in \mathbb{A}(\gamma)$*. We define the Circulation DODAG (*$\texttt{CD}^o$*) anchored at* a *in* γ*, and denote* D_a^γ*, as the* $\texttt{D}^o$ *with nodes* V_a^γ*, and whose parental relationship is defined by:*

$$\forall u, v \in V_a^\gamma, (u \in \mathcal{C}_{D_a^\gamma}(v) \iff u \in \mathcal{C}_G(v) \wedge \texttt{tok}_v^\gamma \neq \uparrow)$$

Remark 5.5

We use the notation $v \in D_a^\gamma$ *as an alias for* $v \in V_a^\gamma$*.*

Remark 5.6

A $\texttt{CD}^o$ *with anchor* a *is a sub-*$\texttt{D}^o$ *of* G_a*, the* $\texttt{D}^o$ *under* a *(see Definition 5.4).*

Lemma 5.8 establishes that any node v such that $\texttt{tok}_v \neq \bot$ belongs to at least one $\texttt{CD}^o$. It will be useful in the latter to prove that any action taken by some node has an effect on at least one $\texttt{CD}^o$, and therefore decreases the weight of at least one $\texttt{CD}^o$.

Lemma 5.8

$\forall v \in V$*, if* $\texttt{tok}_v^\gamma \neq \bot$*, then there exists an anchor* $a \in \mathbb{A}(\gamma)$ *such that* $v \in D_a^\gamma$

Proof: Let us consider a maximal sequence $v_0 v_1 \cdots v_k$ such that $v = v_0$ and $\forall i \in [1, k], \texttt{tok}_{v_i}^\gamma \notin \{\bot \uparrow\} \wedge v_i \in \mathcal{P}(v_{i-1})$. Since the sequence is maximal, $v_k \in \mathbb{A}(\gamma)$, and by definition we also have $\forall i, v_i \in D_{v_k}^\gamma$. Consequently, $v \in D_{v_k}^\gamma$. ■

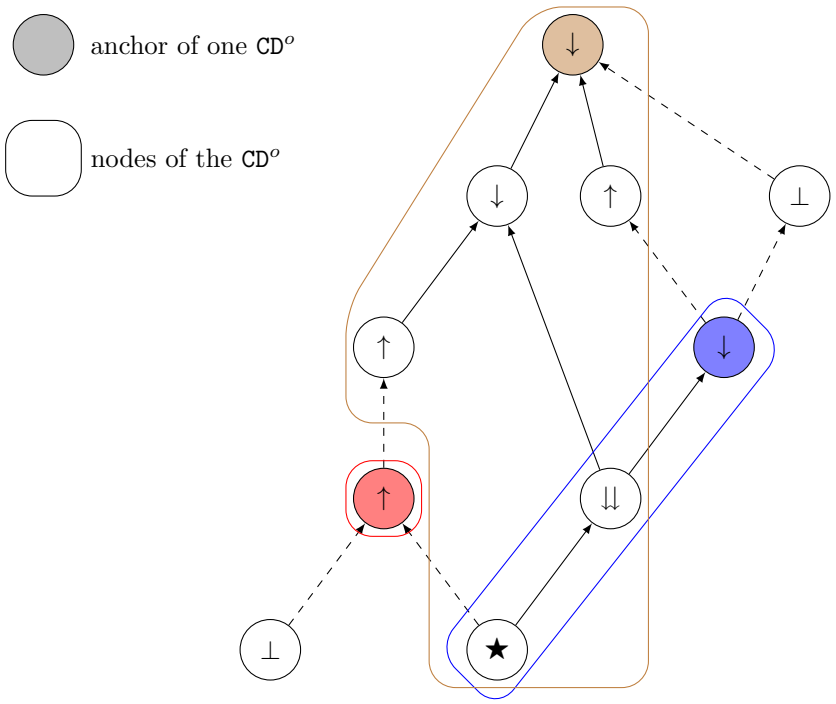

Figure 5.22: Example of three $\texttt{CD}^o$'s in the same graph G

Lemma 5.9 states that the set of anchors $\mathbb{A}$ of G is non-increasing: if one node v is not an anchor in a configuration γ, then it will never be an anchor in a further configuration. This lemma justifies that focusing on the decrease of the weight of $\texttt{CD}^o$ is sufficient, since no $\texttt{CD}^o$ appears.

Lemma 5.9

Let $cs = \gamma \to \gamma'$ be a computing step. We have $\mathbb{A}(\gamma') \subseteq \mathbb{A}(\gamma)$

Proof: Let us prove the equivalent statement: $V \setminus \mathbb{A}(\gamma) \subseteq V \setminus \mathbb{A}(\gamma')$. Let the node v be $v \notin \mathbb{A}(\gamma)$, and prove that $v \notin \mathbb{A}(\gamma')$.

- Suppose first that $\texttt{tok}_v^\gamma = \bot$.
  If $\texttt{tok}_v^{\gamma'} = \bot$, then $v \notin \mathbb{A}(\gamma')$. Otherwise, we have $\texttt{tok}_v^\gamma = \bot$ and $\texttt{tok}_v^{\gamma'} \neq \bot$, which is possible only if v executes $\mathbb{R}_{\texttt{Win}}$ during cs. But this is possible only if v has one parent p such that $\texttt{tok}_p^\gamma = \Downarrow$. But in such a case, we necessarily have $\texttt{tok}_p^{\gamma'} \in \{\Downarrow, \circlearrowleft, \downarrow\}$, and thus $v \notin \mathbb{A}(\gamma')$.
- Suppose now that $\texttt{tok}_v^\gamma \neq \bot$.
  Since $v \notin \mathbb{A}(\gamma)$, v has one parent p such that $\texttt{tok}_p^\gamma \notin \{\bot, \uparrow\}$. Let us prove that $\texttt{tok}_p^{\gamma'} \notin \{\bot, \uparrow\}$. Rule $\mathbb{R}_{\texttt{OfferUp}}$ is the only rule whose effect might update $\texttt{tok}$ from a state that is neither $\bot$ nor $\uparrow$, to a state that is $\bot$ or $\uparrow$. But p can execute $\mathbb{R}_{\texttt{OfferUp}}$ during cs only if $\texttt{tok}_p^\gamma = \bullet$. Under such circumstances, since $\texttt{tok}_v^\gamma \neq \bot$, $\texttt{Er}^\gamma(p) = \texttt{true}$, so p cannot execute $\mathbb{R}_{\texttt{OfferUp}}$ during cs. Thus, $\texttt{tok}_p^{\gamma'} \notin \{\bot, \uparrow\}$ and by definition, $v \notin \mathbb{A}(\gamma')$. ■

Remark 5.7

In the latter, when considering the effect of a computing step $\gamma \to \gamma'$ on a $\texttt{CD}^o$, we always

work under the hypothesis that $a \in \mathbb{A}(\gamma')$, which also implies that $a \in \mathbb{A}(\gamma)$ according to Lemma 5.9.

Corollary 5.1

The number of $\texttt{CD}^o$ does not increase.

A $\texttt{CD}^o$ evolves through an execution of the algorithm, and in particular it can federate new nodes during some computing steps, if some node close to the $\texttt{CD}^o$ executes $\mathbb{R}_{\texttt{Win}}$. To be able to consider such situations, we introduce the notion of *border* of a $\texttt{CD}^o$, which are close nodes that might join the $\texttt{CD}^o$.

Definition 5.17 (*Border of a* $\texttt{CD}^o$)

Let D_a^γ be a $\texttt{CD}^o$. We define the border of D_a^γ and denote $\mathbb{B}(D_a^\gamma)$ the set of nodes $v \in V$ such that $\texttt{tok}_v^\gamma = \perp$, and $\exists p \in \mathcal{P}(v) : p \in D_a^\gamma \wedge \texttt{tok}_p^\gamma \neq \uparrow$.

The border of a $\texttt{CD}^o$ describes the nodes that can join a $\texttt{CD}^o$. On the other hand, some nodes can leave a $\texttt{CD}^o$. Lemma 5.10 establishes that the only nodes susceptible to leave a $\texttt{CD}^o$ during one computing step are the leaves of the $\texttt{CD}^o$: the nodes v such that $\texttt{tok}_v = \uparrow$.

Lemma 5.10

Let $cs = \gamma \to \gamma'$ be a computing step, and let $a \in \mathbb{A}(\gamma')$. Let $v_1 v_2 \cdots v_k$ be a branch of D_a^γ such that $v_1 = a$. Then $v_1 v_2 \cdots v_{k-1}$ is a branch of $D_a^{\gamma'}$, and if v_k does not execute $\mathbb{R}_{\texttt{Return}}$ during cs, then $v_1 v_2 \cdots v_k$ is a branch of $D_a^{\gamma'}$.

Proof: If $\mathcal{B} = v_1 v_2 \cdots v_k$ is a branch of $D_a^{\gamma'}$ the result is immediate. Let us rather suppose that $\mathcal{B}$ is not a branch of $D_a^{\gamma'}$, and let us consider the highest value i such that $\mathcal{B}_i = v_1 v_2 \cdots v_i$ is a branch of $D_a^{\gamma'}$. Remark that $v_1 = a$ is necessarily a branch of $D_a^{\gamma'}$, so $i \in [1, k-1]$.

By definition, $v_{i+1} \in \mathcal{C}_{D_a^\gamma}(v_i)$, so $\texttt{tok}_{v_i}^\gamma \notin \{\perp, \uparrow\}$ and $\texttt{tok}_{v_{i+1}}^\gamma \neq \perp$. Let us first prove that $\texttt{tok}_{v_i}^{\gamma'} \notin \{\perp, \uparrow\}$. The only possibility for the opposite would be that v_i executes $\mathbb{R}_{\texttt{OfferUp}}$ during cs. But this implies that $\texttt{tok}_{v_i}^\gamma = \bullet$ and thus $\texttt{Er}^\gamma(v_i)$ due to v_{i+1}. Thus, if activated, v_i does not execute $\mathbb{R}_{\texttt{OfferUp}}$ during cs but $\mathbb{E}_{\texttt{TrustChild}}$.

Consequently, since we have $v_{i+1} \notin \mathcal{C}_{D_a^{\gamma'}}(v_i)$, we deduce that $\texttt{tok}_{v_{i+1}}^{\gamma'} = \perp$, and thus v_{i+1} executes $\mathbb{R}_{\texttt{Return}}$ during cs. But then, $\texttt{tok}_{v_{i+1}}^\gamma = \uparrow$ and thus v_{i+1} has no children in D_a^γ.

We deduce that $v_{i+1} = v_k$, which means that $v_i = v_{k-1}$ and thus $v_1 v_2 \cdots v_{k-1}$ is a branch of $D_a^{\gamma'}$, and if v_k does not execute $\mathbb{R}_{\texttt{Return}}$ then $v_1 v_2 \cdots v_k$ is a branch of $D_a^{\gamma'}$. ■

Lemma 5.11 establishes constraints on which actions might be taken by a node.

Lemma 5.11

Let $cs = \gamma \to \gamma'$ be a computing step, and let v be a node. If $v = r$ then v can update $\texttt{c}_v$ only if $\texttt{tok}_v^\gamma = \uparrow$, and if $v \neq r$ then v can update $\texttt{c}_v$ only if $\texttt{tok}_v^\gamma = \perp$.

Proof: If $v = r$ then only $\mathbb{R}^r_{\texttt{NewDFS}}$ can update $\texttt{c}_v$, and it is enabled only if $\texttt{tok}_v^\gamma = \uparrow$. If $v \neq r$ then only $\mathbb{R}_{\texttt{Win}}$ and $\mathbb{R}_{\texttt{FakeWin}}$ can update $\texttt{c}_v$, and both are enabled only if $\texttt{tok}_v^\gamma = \perp$. ■

5.4.2.3 Introduction to the potential function W

To prove that one $\texttt{CD}^o$ can only be activated a finite number of times, we associate a weight to each $\texttt{CD}^o$ in each configuration γ. We design this weight function (or potential function) such that if one node is activated in the $\texttt{CD}^o$ or near it during the computing step $\gamma \to \gamma'$,

then the weight of the $\texttt{CD}^o$ is less in γ' than in γ. The goal is to provide the domain of weights with a well-founded order, so that such decreasing sequences are necessarily finite. Recall that decreasing occurs only during periods during which no execution of $\mathbb{R}^r_{\texttt{NewDFS}}$ resets the weight of a $\texttt{CD}^o$ to a high value.

Unfortunately, the structure of $\texttt{CD}^o$ is too complex for it can be valuated by an object as simple as an integer. The most natural way to associate a weight to a $\texttt{CD}^o$ is to build a $\texttt{D}^o$, similar to the $\texttt{CD}^o$ considered, but whose nodes are labeled with weights. We call such structures Weighted DODAGs, denoted $\texttt{WD}^o$'s for short.

In this section, we consider $(M, \leq)$ an arbitrary well-founded set. Although our weight function is defined on $\texttt{CD}^o$'s, it is simpler to work through this section by considering arbitrary $\texttt{D}^o$'s. Since $\texttt{CD}^o$'s have a $\texttt{D}^o$ structure, all this work naturally applies to $\texttt{CD}^o$'s.

Definition 5.18 formally introduces $\texttt{WD}^o$'s: the object on which we will define a well-founded order and that will later be considered as the weight of $\texttt{D}^o$'s.

Definition 5.18 (*Weighted DODAG*)

A weighted $\texttt{D}^o$ *(*$\texttt{WD}^o$*) is a* $\texttt{D}^o$ *with labels in* M*:* $(D_a, \texttt{W})$*, where* D_a *is a* $\texttt{D}^o$ *and* $\texttt{W} \in M^{V_a}$ *is a map that associates an element of* M *to each node of* D_a*.*

The next step is to define an order of $\texttt{WD}^o$'s. This is feasible only if the two $\texttt{WD}^o$'s we compare are based on $\texttt{D}^o$'s having the same anchor a. This won't be restrictive since in practice, we compare the weights of the same $\texttt{CD}^o$ at different, consecutive, configurations of an execution.

To compare $\texttt{WD}^o$'s, the first step is to compare branches of $\texttt{WD}^o$'s. Since two $\texttt{WD}^o$'s that share the same anchor do not necessarily share the same branches in their topology, the appropriate notion to this comparison is the projection of $\texttt{WD}^o$'s on the branches of the $\texttt{D}^o$ under their common anchor. This is similar to what we introduced in Definition 5.6.

Definition 5.19 (*Projection of a* $\texttt{WD}^o$ *on a Branch*)

Let $(D_a, \texttt{W})$ *be a* $\texttt{WD}^o$*, and let* $\mathcal{B} = v_1 v_2 \cdots v_k$ *be a maximal branch of* G_a*. We call projection of* $(D_a, \texttt{W})$ *on* $\mathcal{B}$*, and denote* $(D_a, \texttt{W})(\mathcal{B})$ *the sequence* $\texttt{W}(v_1)\texttt{W}(v_2)\cdots\texttt{W}(v_j)$*, where* $D_a(\mathcal{B}) = v_1 v_2 \cdots v_j$ *is the projection of* D_a *on* $\mathcal{B}$*.*

Since the branches of two $\texttt{WD}^o$'s may have variable length, we compare them using the alphabetical order (see Definition 5.8). This order is well-founded since the branches of the $\texttt{WD}^o$'s are all bounded by n, the number of nodes in the graph.

To compare $\texttt{WD}^o$'s, we compare the weights of all the branches which start at a. One $\texttt{WD}^o$ D_a^1 is less than or equal to one other $\texttt{WD}^o$ D_a^2 if, for any branch $\mathcal{B}$ of G_a the $\texttt{D}^o$ under a, the projection of D_a^1 on $\mathcal{B}$ is less than or equal to the projection of D_a^2 on $\mathcal{B}$.

Definition 5.20 presents a well-founded order on $\texttt{WD}^o$'s.

Definition 5.20 (*Order on* $\texttt{WD}^o$*'s*)

We define a partial order on $\texttt{WD}^o$*'s which share the same anchor by:*

$$(D'_a, \texttt{W}') \preceq (D_a, \texttt{W}) \iff \begin{array}{l} \text{for all maximal branch } \mathcal{B} \text{ of } G_a, \\ (D'_a, \texttt{W}')(\mathcal{B}) \preceq_\alpha (D_a, \texttt{W})(\mathcal{B}) \end{array}$$

By construction, the resulting order is not a total order. Indeed, there exist $\texttt{D}^o$'s and branches such that $(D'_a, \texttt{W}')(\mathcal{B}_1) \prec_\alpha (D_a, \texttt{W})(\mathcal{B}_1)$, on the one hand, and $(D_a, \texttt{W})(\mathcal{B}_2) \prec_\alpha (D'_a, \texttt{W}')(\mathcal{B}_2)$ on the other hand. Yet, we guarantee in the following sections that any computing step makes the weight of the $\texttt{D}^o$ decrease (which implies that the corresponding $\texttt{WD}^o$'s are comparable).

Lemma 5.12

Definition 5.20 defines a well-founded order.

Proof: By construction, $\preceq$ is reflexive, anti-symmetric, and transitive. Let us prove that it is also well-founded. Consider an infinite sequence of $\mathtt{WD}^o$'s $(D_a^n, \mathtt{W}^n)_{n\in\mathbb{N}}$ such that $\forall n, (D_a^{n+1}, \mathtt{W}^{n+1}) \preceq (D_a^n, \mathtt{W}^n)$, and let us prove that there exists $N \in \mathbb{N}$ such that $\forall n \geq N, (D_a^n, \mathtt{W}^n) = (D_a^N, \mathtt{W}^N)$.

Let $\mathcal{B}$ be a maximal branch of G_a, and let k be the length of $\mathcal{B}$. By definition, $\forall n, (D_a^{n+1}, \mathtt{W}^{n+1})(\mathcal{B}) \preceq_\alpha (D_a^n, \mathtt{W}^n)(\mathcal{B})$. But $\preceq_\alpha$ is a well-founded order on sequences of length at most k, and $\forall n, (D_a^n, \mathtt{W}^n)(\mathcal{B})$ is a sequence of length at most k. Consequently, there exists $N_\mathcal{B} \in \mathbb{N}$ such that $\forall n \geq N_\mathcal{B}, (D_a^{n+1}, \mathtt{W}^{n+1})(\mathcal{B}) = (D_a^n, \mathtt{W}^n)(\mathcal{B})$.

Since there exists a finite number of maximal branches $\mathcal{B}$ of G_a, let us consider $N = \max_\mathcal{B}(N_\mathcal{B})$. By definition, $\forall n \geq N$, for all maximal branch $\mathcal{B}$ of G_a, $(D_a^n, \mathtt{W}^n)(\mathcal{B}) = (D_a^N, \mathtt{W}^N)(\mathcal{B})$, and thus, $\forall n \geq N, (D_a^N, \mathtt{W}^N) \preceq (D_a^n, \mathtt{W}^n)$. **Since** $\preceq$ is an order, we have $\forall n \geq N, (D_a^n, \mathtt{W}^n) = (D_a^N, \mathtt{W}^N)$. ■

We now show how to associate a $\mathtt{WD}^o$ to any $\mathtt{CD}^o$ as soon as we are given a weight function $\mathtt{W}$ on nodes of G. This association is itself a weight function on $\mathtt{CD}^o$, induced by the weight function on nodes. For simplicity, we also call $\mathtt{W}$ the induced weight function.

Contrary to what was presented in Definition 5.18, the weight function on nodes may depend on the global configuration γ of the system, and not only on the state of nodes v. More formally, we now consider a weight function $\mathtt{W}(v, \gamma)$ which associates an element of M to each node in a given configuration. This does not pose any fundamental problem since, given γ, we can still associate a $\mathtt{WD}^o$ to any $\mathtt{CD}^o$ D_a^γ. We can as well associate, given γ, a weight to any projection of D_a^γ on any branch of G_a. This is formally stated in Definition 5.21.

Definition 5.21 (*Weight of a circulation DAG*)

Let $\mathtt{W} : V \times \Gamma \to M$ be a function that associates a weight to any node of a configuration.

For any fixed configuration $\gamma \in \Gamma$, we define $\mathtt{W}(\,\cdot\,, \gamma) \in M^{V_{D_a}}$ the function that associates to any node $v \in V_{D_a}$ the weight of v in γ: $\mathtt{W}(\,\cdot\,, \gamma)(v) = \mathtt{W}(v, \gamma)$.

Then, for all $\gamma \in \Gamma$, for all $\mathtt{CD}^o$ D_a^γ at γ, we can consider $\mathtt{W}(D_a^\gamma, \gamma)$ the weight of D_a^γ in γ, which is the $\mathtt{WD}^o$ $(D_a^\gamma, \mathtt{W}(\,\cdot\,, \gamma))$.

We also define the weight of a branch $\mathcal{B} = v_1 v_2 \cdots v_k$ of D_a^γ in γ, and denote $\mathtt{W}(D_a^\gamma(\mathcal{B}), \gamma) = \mathtt{W}(D_a^\gamma, \gamma)(\mathcal{B}) = (D_a^\gamma, \mathtt{W}(\,\cdot\,, \gamma))(\mathcal{B}) = \mathtt{W}(v_1, \gamma)\mathtt{W}(v_2, \gamma) \cdots \mathtt{W}(v_k, \gamma)$.

To deal with all the different aspects of the algorithm, we actually define several weight functions in Sections 5.4.2.4 to 5.4.2.8, dealing with the different aspects of our algorithm. Each of these functions defines one different $\mathtt{WD}^o$ to the same $\mathtt{CD}^o$. In order to deduce global properties on the evolution of the entire algorithm, we must combine all those functions and to compare the different induces $\mathtt{WD}^o$'s all at once. To do so, we simply use the lexicographic order on $\mathtt{WD}^o$'s that is induced by the order $\preceq$ introduced in Definition 5.20.

Definition 5.22 (*Lexicographic order on $\mathtt{WD}^o$'s*)

We denote by $\preceq^k$ the lexicographic order on k-tuples of $\mathtt{WD}^o$'s which share the same anchor, induced by the order $\preceq$.

$\forall k \in \mathbb{N}$, $\preceq^k$ is a well-founded order, as a lexicographic order induced by a well-founded order.

Our goal is now to prove two properties. The first one is that the $\mathtt{WD}^o$'s associated to one $\mathtt{CD}^o$ never increase unless the anchor of the $\mathtt{CD}^o$ is the root, and the root executes $\mathbb{R}^r_{\mathtt{NewDFS}}$. The second property is that each time one node of the $\mathtt{CD}^o$, or near it, is activated (by a rule

which is not $\mathbb{R}^r_{\texttt{NewDFS}}$), the weight of the $\texttt{CD}^o$ decreases. A weight function which respects the first property is said *pre-admissible*, and it is said *admissible* if it respects both.

Since the weight of a $\texttt{CD}^o$ is expressed as a tuple of $\texttt{WD}^o$'s, the formal definitions of admissibility and pre-admissibility lie on the lexicographic order introduced in Definition 5.22. Each potential function has an even stronger weight as it appears on the first component of the tuple.

The exact definition of admissibility actually depends on how we define what a node *near a* $\texttt{CD}^o$ is. This definition depends on constraints which will emerge from the definitions of the weight functions, and therefore will be provided later, in Definition 5.25 of Section 5.4.2.9.

We give here the definition of pre-admissibility.

Definition 5.23 (*Pre-admissible weight-functions*)

Let $\texttt{W}_1, \ldots, \texttt{W}_k : V \times \Gamma \to M$ *be* k *weight functions on nodes. We say that* $(\texttt{W}_1, \ldots, \texttt{W}_k)$ *is pre-admissible if for any computing step* $cs = \gamma \to \gamma'$ *such that* r *does not execute* $\mathbb{R}^r_{\texttt{NewDFS}}$ *during* cs*, for all* $\texttt{CD}^o$ $D_a^{\gamma'}$*, we have*

$$\big(\texttt{W}_1(D_a^{\gamma'}, \gamma'), \cdots, \texttt{W}_k(D_a^{\gamma'}, \gamma')\big) \preceq^k \big(\texttt{W}_1(D_a^{\gamma}, \gamma), \cdots, \texttt{W}_k(D_a^{\gamma}, \gamma)\big).$$

Remark 5.8

In order to have relatively short and readable equations in the following, the previous inequality will often be written:

$$\big(\texttt{W}_1, \ldots, \texttt{W}_k\big)(D_a^{\gamma'}, \gamma') \preceq^k \big(\texttt{W}_1, \ldots, \texttt{W}_k\big)(D_a^{\gamma}, \gamma)$$

In Sections 5.4.2.4 to 5.4.2.8 we define five weight functions on nodes, that belong to $V \times \Gamma \to \mathbb{N}$: which are $\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}}$ and $\texttt{W}_{\texttt{Nego}}$. Each one of those 5 functions deals with one specific part of the algorithm, by decreasing order of importance, and is treated separately in one of the next five sections. Thus, we define, for each configuration, 5 $\texttt{WD}^o$'s that, associated with the lexicographic order $\preceq^5$, will allow us to formally prove that the token circulation terminates. Figure 5.23 summarizes the effects of the rules, and for each of them indicates which component of $\texttt{W}$ it decreases.

We prove that $\texttt{W} = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}}, \texttt{W}_{\texttt{Nego}})$ is pre-admissible, by proving step by step that all the prefixes of that 5-tuple are pre-admissible. We also establish some properties that will allow us to prove, in Section 5.4.2.9, that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}}, \texttt{W}_{\texttt{Nego}})$ is admissible.

5.4.2.4 First component of W: $\texttt{W}_{\texttt{Ch}}$ future children of v

Explanations One property which guarantees that circulation rounds are finite is that one node can receive the token from its parent only once, as long as the parent does not return the token to its own parent.

Indeed, once a node u receives the token from one of its parents p, it takes the same color as that parent, and its variable $\texttt{play}$ becomes $\texttt{W}$. Variable $\texttt{play}_u$ then cannot be set to $\texttt{P}$ before $\texttt{tok}_p$ is set to $\uparrow$, which implies that u cannot take the token once again from p before p itself returns the token to its own parent.

For that reason, the first component of the weight function $\texttt{W}$ should, at least, count on each node $u \in V$ the number of children of u that are susceptible to execute $\mathbb{R}_{\texttt{Win}}$ before u executes $\mathbb{R}_{\texttt{OfferUp}}$. The variables that we must consider to establish if one node is susceptible to execute $\mathbb{R}_{\texttt{Win}}$ are $\texttt{play}$ and $\texttt{c}$. In the following we detail how we formally define this count.

node	Rule	id	tok	c	play	W ↘
$v \neq r$	$\mathbb{R}_{\texttt{Win}}$		$\bot \to \bigstar$	switch	$\texttt{P} \to \texttt{W}$	$\texttt{W}_{\texttt{Ch}}$
$v \neq r$	$\mathbb{R}_{\texttt{FakeWin}}$			switch	$\texttt{P} \to \{\texttt{W}, \texttt{F}\}$	$\texttt{W}_{\texttt{Ch}}$
$\forall v \in V$	$\mathbb{E}_{\texttt{TrustChild}}$		$\{\bullet, \downdownarrows, \circlearrowleft, \circ, \bigstar\} \to \downarrow$			$\texttt{W}_{\texttt{Er}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{Give}}$		$\bullet \to \downdownarrows$			$\texttt{W}_{\texttt{Circ}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{NewPlay}}$		$\downdownarrows \to \circlearrowleft$			$\texttt{W}_{\texttt{Circ}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{Drop}}$		$\circlearrowleft \to \downarrow$			$\texttt{W}_{\texttt{Circ}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{Nego}}$	$(1, \bot)$	$\bigstar \to \bullet$			$\texttt{W}_{\texttt{Circ}}$
$v \neq r$	$\mathbb{R}_{\texttt{OfferUp}}$		$\bullet \to \uparrow$			$\texttt{W}_{\texttt{Circ}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{Receive}}$		$\downarrow \to \circ$			$\texttt{W}_{\texttt{Circ}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{ReNego}}$	$(1, \bot)$	$\circ \to \bullet$			$\texttt{W}_{\texttt{Circ}}$
$v \neq r$	$\mathbb{R}_{\texttt{Return}}$		$\uparrow \to \bot$		$\to \{\texttt{W}, \texttt{F}\}$	$\texttt{W}_{\texttt{Circ}}$
$v \neq r$	$\mathbb{R}_{\texttt{Lose}}$				$\texttt{P} \to \texttt{L}$	$\texttt{W}_{\texttt{Play}}$
$v \neq r$	$\mathbb{R}_{\texttt{ReplayD}}$				$\texttt{L} \to \texttt{P}$	$\texttt{W}_{\texttt{Play}}$
$v \neq r$	$\mathbb{R}_{\texttt{ReplayUp}}$				$\texttt{W} \to \texttt{P}$	$\texttt{W}_{\texttt{Play}}$
$v \neq r$	$\mathbb{R}_{\texttt{Fake}}$				$\texttt{W} \to \texttt{F}$	$\texttt{W}_{\texttt{Play}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{maxPos}}$	$\max_{u \in \mathcal{C}(v)} \texttt{id}_u$				$\texttt{W}_{\texttt{Nego}}$
$\forall v \in V$	$\mathbb{R}_{\texttt{NewPh}}$	$(\texttt{ph} + 1, \bot)$				$\texttt{W}_{\texttt{Nego}}$
$v \neq r$	$\mathbb{R}_{\texttt{NewBit}}$	$(\texttt{ph}, \texttt{Bit}(\texttt{ph}))$				$\texttt{W}_{\texttt{Nego}}$
$v \neq r$	$\mathbb{R}_{\texttt{ReWin}}$				$\texttt{F} \to \texttt{W}$	
r	$\mathbb{R}^{r}_{\texttt{NewDFS}}$		$\uparrow \to \bigstar$	switch		

Figure 5.23: Dedicated weight function for each rule. For readability, one color is associated to each weight function.

Let us consider one node v and one of its children u. Remark first that if $\texttt{tok}_v \in \{\bot, \uparrow\}$, then the weight of v can be defined as 0. Indeed, such a node cannot be involved in a pass of the token to its children before it itself receives the token from one of its parents. Thus, when v receives the token from its parent, its own weight increases from 0 to the actual count of how many children susceptible to execute $\mathbb{R}_{\texttt{Win}}$ it has. But at the same moment, the weight of v's parent decreases by one due to v cannot execute $\mathbb{R}_{\texttt{Win}}$ anymore. Consequently, due to how we defined $\preceq_\alpha$ on weighted chains, the weight of each branch of a $\texttt{CD}^o$ that contains v and its parent decrease when v executes $\mathbb{R}_{\texttt{Win}}$.

We now consider one node v such that $\texttt{tok}_v \notin \{\bot, \uparrow\}$ and one child u of v. In this section, we only focus on the transmission of the token between nodes, and not on the transitions of the variable tok on one node. Thus, we treat indifferently the different cases when $\texttt{tok}_v \in \{\bullet, \bigstar, \downdownarrows, \circlearrowleft, \circ, \downarrow\}$. Let us establish under which circumstances u might receive the token from v.

- Node u can receive the token, at first, if $\texttt{c}_u \neq \texttt{c}_v$ and $\texttt{play}_u \in \{\texttt{P}, \texttt{L}\}$, by simply executing $\mathbb{R}_{\texttt{Win}}$ (or after one execution of $\mathbb{R}_{\texttt{ReplayD}}$).

- Secondly, if $\texttt{c}_u \neq \texttt{c}_v$ and $\texttt{play}_u \in \{\texttt{W}, \texttt{F}\}$, then u can execute $\mathbb{R}_{\texttt{ReplayUp}}$, and $\mathbb{R}_{\texttt{ReWin}}$ if necessary, and then arrives in the previous situation. Therefore, we must count u as a node that might receive the token.

- Finally, if $\texttt{c}_u = \texttt{c}_v$ and $\texttt{play}_u \in \{\texttt{P}, \texttt{L}\}$, then u can execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, and then arrives in the previous situation.

Note that if $\mathtt{c}_u = \mathtt{c}_v$ and $\mathtt{play}_u \in \{\mathtt{W}, \mathtt{F}\}$ then u cannot execute $\mathbb{R}_{\mathtt{Win}}$ nor $\mathbb{R}_{\mathtt{FakeWin}}$ due to $\mathtt{play}_u$, and cannot execute $\mathbb{R}_{\mathtt{ReplayUp}}$ since $\mathtt{tok}_v \in \{\bot, \uparrow\}$. This is described in Figure 5.24.

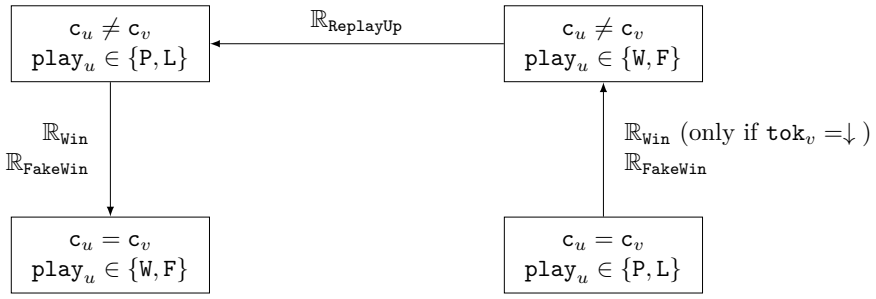

Figure 5.24: Evolution of the state of a child of node v with $\mathtt{tok}_v \notin \{\bot, \uparrow\}$.

All the transitions presented in the diagram of Figure 5.24 bring the node that executes the corresponding rule closer to the stable state where $\mathtt{c}_u = \mathtt{c}_v \wedge \mathtt{play}_u \in \{\mathtt{W}, \mathtt{F}\}$. We need to fully describe the process that leads to the passing of the token to one child. Therefore, we define $\mathtt{W}_{\mathtt{Ch}}$ such that any execution of a rule that corresponds to a transition presented in Figure 5.24 makes $\mathtt{W}_{\mathtt{Ch}}$ decrease on the parent of the node that executes it.

Yet, remark that we cannot treat $\mathbb{R}_{\mathtt{ReplayUp}}$ the same way as we treat $\mathbb{R}_{\mathtt{Win}}$ and $\mathbb{R}_{\mathtt{FakeWin}}$. Indeed, any execution of $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ by one node u requires that at least one parent v of u has its variable $\mathtt{tok}_v$ different from $\{\bot, \uparrow\}$, which means that any such execution corresponds to a transition on that diagram for that node v. On the contrary, one node u might execute $\mathbb{R}_{\mathtt{ReplayUp}}$ without any parent v such that $\mathtt{tok}_v \notin \{\bot, \uparrow\}$. Thus, although we treat some, we do not treat all possible executions of $\mathbb{R}_{\mathtt{ReplayUp}}$ with $\mathtt{W}_{\mathtt{Ch}}$. This means that, by the end of this section, we prove that any activation of $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ makes $\mathtt{W}$ decrease, and that some specific activation of $\mathbb{R}_{\mathtt{ReplayUp}}$ makes $\mathtt{W}$ decrease. This piece of knowledge will nevertheless be useful in the following sections until we prove that any activation of $\mathbb{R}_{\mathtt{ReplayUp}}$ makes $\mathtt{W}$ decrease. This is formally stated in Theorems 5.3 and 5.4.

Definitions Since children u of v move along the different boxes of Figure 5.24, we must design $\mathtt{W}_{\mathtt{Ch}}$ such that any transition taken by u makes $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') < \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$. Our solution is to attribute different coefficients to the different boxes: the further node u is from the stable configuration $\mathtt{c}_u = \mathtt{c}_v \wedge \mathtt{play}_u \in \{\mathtt{P}, \mathtt{L}\}$, the higher the coefficient. Somehow, the coefficient represents the number of transitions needed by one node to reach the stable configuration. The coefficients are the integers above and below the boxes in Figure 5.25.

In practice, to make the proof easier to approach, we define 3 sets on each node v, such that any transition taken by one child u of v brings u out of one of the sets. Those sets have a non-empty intersection, and are such that the coefficient attributed to one node is the number of sets it belongs to.

- One set includes all the nodes that are potential candidates to receiving the token from v, that is to say all the children u of v that are not in a stable state.
- One other set, included in the first one, includes all of those candidates that must execute $\mathbb{R}_{\mathtt{ReplayUp}}$ before they might receive the token from v.
- Finally, the last set, included in the second one, includes all of those candidates that must execute $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ before they can execute $\mathbb{R}_{\mathtt{ReplayUp}}$ and receive the token from v, that are also the candidates that are the same color as v.

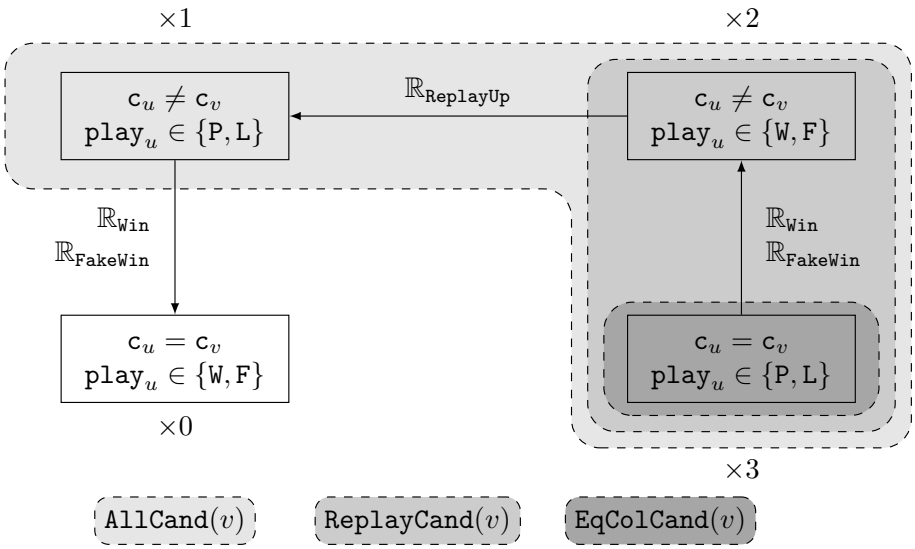

Figure 5.25: Weight associated to a child of node v with $\mathtt{tok}_v \notin \{\bot, \uparrow\}$.

Let us define those three sets, starting from the last, and smallest, one.

$$\mathtt{EqColCand}^\gamma(v) = \{u \in \mathcal{C}(v) \mid \mathtt{c}_u^\gamma = \mathtt{c}_v^\gamma \wedge \mathtt{play}_u^\gamma \in \{\mathtt{P},\mathtt{L}\}\} \tag{5.34}$$

$$\mathtt{ReplayCand}^\gamma(v) = \mathtt{EqColCand}^\gamma(v) \cup \{u \in \mathcal{C}(v) \mid \mathtt{c}_u^\gamma \neq \mathtt{c}_v^\gamma \wedge \mathtt{play}_u^\gamma \in \{\mathtt{W},\mathtt{F}\}\} \tag{5.35}$$

$$\mathtt{AllCand}^\gamma(v) = \mathtt{ReplayCand}^\gamma(v) \cup \{u \in \mathcal{C}(v) \mid \mathtt{c}_u^\gamma \neq \mathtt{c}_v^\gamma \wedge \mathtt{play}_u^\gamma \in \{\mathtt{P},\mathtt{L}\}\} \tag{5.36}$$

Finally, we can define $\mathtt{W}_{\mathtt{Ch}}(v,\gamma)$ such that the coefficients on Figure 5.25 are respected:

$$\mathtt{W}_{\mathtt{Ch}}(v,\gamma) = \begin{cases} 0 & \text{if } \mathtt{tok}_v^\gamma \in \{\bot,\uparrow\} \\ \left|\mathtt{AllCand}^\gamma(v)\right| + \left|\mathtt{ReplayCand}^\gamma(v)\right| + \left|\mathtt{EqColCand}^\gamma(v)\right| & \text{otherwise} \end{cases} \tag{5.37}$$

Proofs Lemmas 5.13, 5.14, and 5.15 state that $\forall v \in V$, the three sets defined previously can only decrease during an execution, as long as $\mathtt{tok}_v \notin \{\bot,\uparrow\}$. Basically, we prove that there is no other transition than the ones we represented in Figure 5.25. These three lemmas are one of the main arguments to establish that $\mathtt{W}_{\mathtt{Ch}}$ is a pre-admissible weight function.

Lemma 5.13

Let $cs = \gamma \to \gamma'$ be a computing step and let $v \in V$ be a node. If $\mathtt{tok}_v^\gamma \notin \{\bot,\uparrow\}$ then $\mathtt{AllCand}^{\gamma'}(v) \subseteq \mathtt{AllCand}^\gamma(v)$.

Proof: We prove the equivalent statement:

$$\mathcal{C}(v) \setminus \mathtt{AllCand}^\gamma(v) \subseteq \mathcal{C}(v) \setminus \mathtt{AllCand}^{\gamma'}(v).$$

Since $\mathtt{tok}_v^\gamma \notin \{\bot,\uparrow\}$, then according to Lemma 5.11, v does not update $\mathtt{c}_v$ during cs. Let us consider one child u of v such that $u \notin \mathtt{AllCand}^\gamma(v)$. By definition, $\mathtt{c}_u^\gamma = \mathtt{c}_v^\gamma \wedge \mathtt{play}_u^\gamma \in \{\mathtt{W},\mathtt{F}\}$.

During cs, u does not execute $\mathbb{R}_{\texttt{Win}}$ nor $\mathbb{R}_{\texttt{FakeWin}}$ since $\texttt{play}_u^{\gamma} \neq \texttt{P}$, and does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ since it has at least one parent. Furthermore, u does not execute $\mathbb{R}_{\texttt{ReplayUp}}$ during cs since $v \in \mathcal{P}_{\texttt{eq}}{}^{\gamma}(u) \wedge \texttt{tok}_v^{\gamma} = \uparrow$. Thus, $\texttt{c}_u^{\gamma'} = \texttt{c}_v^{\gamma'} \wedge \texttt{play}_u^{\gamma'} \in \{\texttt{W}, \texttt{F}\}$, in other words, $u \notin \texttt{AllCand}^{\gamma'}(v)$. As a result, $\mathcal{C}(v) \setminus \texttt{AllCand}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{AllCand}^{\gamma'}(v)$. ■

Lemma 5.14

Let $cs = \gamma \to \gamma'$ be a computing step and let $v \in V$ be a node. If $\texttt{tok}_v^{\gamma} \notin \{\bot, \uparrow\}$ then $\texttt{ReplayCand}^{\gamma'}(v) \subseteq \texttt{ReplayCand}^{\gamma}(v)$.

Proof: We prove the equivalent statement:

$$\mathcal{C}(v) \setminus \texttt{ReplayCand}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{ReplayCand}^{\gamma'}(v).$$

Since $\texttt{tok}_v^{\gamma} \notin \{\bot, \uparrow\}$, then according to Lemma 5.11, v does not update $\texttt{c}_v$ during cs. Let us consider one child u of v such that $u \notin \texttt{ReplayCand}^{\gamma}(v)$. If $u \notin \texttt{AllCand}^{\gamma}(v)$ then according to Lemma 5.13, $u \notin \texttt{AllCand}^{\gamma'}(v)$ and thus $u \notin \texttt{ReplayCand}^{\gamma'}(v)$. Let us suppose now that $u \in \texttt{AllCand}^{\gamma}(v) \setminus \texttt{ReplayCand}^{\gamma}(v)$, *i.e.* $\texttt{c}_u^{\gamma} \neq \texttt{c}_v^{\gamma} \wedge \texttt{play}_u^{\gamma} \in \{\texttt{P}, \texttt{L}\}$. If u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs, then $\texttt{c}_u^{\gamma'} = \texttt{c}_v^{\gamma'} \wedge \texttt{play}_u^{\gamma'} = \texttt{W}$ and thus $u \notin \texttt{ReplayCand}^{\gamma'}(v)$. Otherwise, since u has at least one parent it does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ and thus does not update $\texttt{c}_v$, and it can neither update $\texttt{c}_u$ to W or L, and thus $u \notin \texttt{ReplayCand}^{\gamma'}(v)$. As a result, $\mathcal{C}(v) \setminus \texttt{ReplayCand}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{ReplayCand}^{\gamma'}(v)$. ■

Lemma 5.15

Let $cs = \gamma \to \gamma'$ be a computing step and let $v \in V$ be a node. If $\texttt{tok}_v^{\gamma} \notin \{\bot, \uparrow\}$ then $\texttt{EqColCand}^{\gamma'}(v) \subseteq \texttt{EqColCand}^{\gamma}(v)$.

Proof: We prove the equivalent statement:

$$\mathcal{C}(v) \setminus \texttt{EqColCand}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{EqColCand}^{\gamma'}(v).$$

Since $\texttt{tok}_v^{\gamma} \notin \{\bot, \uparrow\}$, then according to Lemma 5.11, v does not update $\texttt{c}_v$ during cs. Let us consider one child u of v such that $u \notin \texttt{EqColCand}^{\gamma}(v)$. If $u \notin \texttt{ReplayCand}^{\gamma}(v)$ then according to Lemma 5.14, $u \notin \texttt{ReplayCand}^{\gamma'}(v)$ and thus $u \notin \texttt{EqColCand}^{\gamma'}(v)$.

Let us suppose now that $u \in \texttt{ReplayCand}^{\gamma}(v) \setminus \texttt{EqColCand}^{\gamma}(v)$, *i.e.* $\texttt{c}_u^{\gamma} \neq \texttt{c}_v^{\gamma} \wedge \texttt{play}_u^{\gamma} \in \{\texttt{W}, \texttt{F}\}$. Since u has at least one parent, it does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs, and since $\texttt{play}_u^{\gamma} \neq \texttt{P}$, u does not execute $\mathbb{R}_{\texttt{Win}}$ nor $\mathbb{R}_{\texttt{FakeWin}}$ during cs. Thus, we have $\texttt{c}_u^{\gamma'} \neq \texttt{c}_v^{\gamma'}$ and thus $u \notin \texttt{EqColCand}(v, \gamma')$. As a result, $\mathcal{C}(v) \setminus \texttt{EqColCand}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{EqColCand}^{\gamma'}(v)$. ■

Lemma 5.16 establishes that if one node executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, or $\mathbb{R}_{\texttt{ReplayUp}}$, then the weight of all its parents in any $\texttt{CD}^o$ decrease. Basically, we formally prove that any such activation corresponds to one of the transitions represented in Figure 5.25, and thus that it makes one of the three sets decrease.

Lemma 5.16

Let $cs = \gamma \to \gamma'$ be a computing step, and let $v \in V$ be a node. If v executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs, then $\forall p \in \mathcal{P}(v)$ such that $\texttt{tok}_p^{\gamma} \notin \{\bot, \uparrow\}$, we have $\texttt{W}_{\texttt{Ch}}(p, \gamma') < \texttt{W}_{\texttt{Ch}}(p, \gamma)$.

Proof: Let us consider $p \in \mathcal{P}(v)$ such that $\texttt{tok}_p^{\gamma} \notin \{\bot, \uparrow\}$. Lemma 5.11 guarantees that $\texttt{c}_p^{\gamma'} = \texttt{c}_p^{\gamma}$.

Suppose first that v executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs, which induces $\texttt{play}_v^{\gamma} = \texttt{P}$ and $\texttt{play}_v^{\gamma'} \in \{\texttt{W}, \texttt{F}\}$.

If $\mathtt{c}_v^\gamma = \mathtt{c}_p^\gamma$ then $v \in \mathtt{EqColCand}^\gamma(p)$. But since $\mathtt{play}_v^{\gamma'} \in \{\mathtt{W}, \mathtt{F}\}$, $v \notin \mathtt{EqColCand}^{\gamma'}(p)$, which according to Lemma 5.15 means that $\mathtt{EqColCand}^{\gamma'}(p) \subsetneq \mathtt{EqColCand}^\gamma(p)$. According to Lemmas 5.13 and 5.14, we obtain $\mathtt{W}_{\mathtt{Ch}}(p, \gamma') < \mathtt{W}_{\mathtt{Ch}}(p, \gamma)$.

Otherwise, $\mathtt{c}_v^\gamma \neq \mathtt{c}_p^\gamma$. Since $\mathtt{play}_v^\gamma = \mathtt{P}$, we have $v \in \mathtt{AllCand}^\gamma(p)$, and since $\mathtt{play}_v^\gamma \in \{\mathtt{W}, \mathtt{F}\}$ and v switches its color during cs, we also have $v \notin \mathtt{AllCand}^{\gamma'}(p)$. According to Lemma 5.13, $\mathtt{AllCand}^{\gamma'}(p) \subsetneq \mathtt{AllCand}^\gamma(p)$, and according to Lemmas 5.14 and 5.15, we obtain $\mathtt{W}_{\mathtt{Ch}}(p, \gamma') < \mathtt{W}_{\mathtt{Ch}}(p, \gamma)$.

Let us now suppose that v executes $\mathbb{R}_{\mathtt{ReplayUp}}$ during cs. Since $\mathtt{tok}_p^\gamma \notin \{\bot, \uparrow\}$, predicate $\mathtt{ReplayW}^\gamma(v)$ implies that $\mathtt{c}_p^\gamma \neq \mathtt{c}_v^\gamma$. Since v execute $\mathbb{R}_{\mathtt{ReplayUp}}$ during cs, we have $\mathtt{play}_v^\gamma = \mathtt{W}$, and thus $v \in \mathtt{ReplayCand}^\gamma(p)$, and we also know that v updates $\mathtt{play}_v$ to $\mathtt{P}$ and that it does not update its variable $\mathtt{c}_v$ which implies that $\mathtt{c}_v^{\gamma'} \neq \mathtt{c}_p^{\gamma'} \wedge \mathtt{play}_v^{\gamma'} = \mathtt{P}$. Thus, $v \notin \mathtt{ReplayCand}^{\gamma'}(p)$ so according to Lemma 5.14 means that $\mathtt{ReplayCand}^{\gamma'}(p) \subsetneq \mathtt{ReplayCand}^\gamma(p)$. According to Lemmas 5.13 and 5.15, we obtain $\mathtt{W}_{\mathtt{Ch}}(p, \gamma') < \mathtt{W}_{\mathtt{Ch}}(p, \gamma)$. ■

Lemma 5.17 establishes that the function $\mathtt{W}_{\mathtt{Ch}}$ is adapted to Definitions 5.8 and 5.20. Indeed, it might happen that one node sees its weight increase, but only if all of its parents see their weight decrease.

Lemma 5.17

Let $cs = \gamma \to \gamma'$ be a computing step and let $v \in V$ such that v does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs. If $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') > \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$ then $\forall p \in \mathcal{P}(v)$ such that $\mathtt{tok}_p^\gamma \notin \{\bot, \uparrow\}$, we have $\mathtt{W}_{\mathtt{Ch}}(p, \gamma') < \mathtt{W}_{\mathtt{Ch}}(p, \gamma)$, and there exists at least one such p.

Proof: According to Lemmas 5.13, 5.14, and 5.15, if $\mathtt{tok}_v^\gamma \in \{\bullet, \bigstar, \downdownarrows, \circlearrowleft, \downarrow, \circ\}$, then $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') \leq \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$. Furthermore, if $\mathtt{tok}_v^\gamma = \uparrow$, then $\mathtt{tok}_v^{\gamma'} \in \{\uparrow, \bot\}$ and thus $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') = \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$.

Finally, if $\mathtt{tok}_v^\gamma = \bot$, then $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') > \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$ is possible only if v executes $\mathbb{R}_{\mathtt{Win}}$ during cs, and thus Lemma 5.16 states that $\forall p \in \mathcal{P}(v)$ such that $\mathtt{tok}_p^\gamma \notin \{\bot, \uparrow\}$, we have $\mathtt{W}_{\mathtt{Ch}}(p, \gamma') < \mathtt{W}_{\mathtt{Ch}}(p, \gamma)$. ■

Theorem 5.3 establishes that $\mathtt{W}_{\mathtt{Ch}}$ is pre-admissible.

Theorem 5.3

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs. We have $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W}_{\mathtt{Ch}}(D_a^\gamma, \gamma)$.

Proof: Let $\mathcal{B} = v_1 v_2 \cdots v_k$ be a maximal branch of G_a, and let $v_1 v_2 \cdots v_j = D_a^{\gamma'}(\mathcal{B})$ be the projection of $D_a^{\gamma'}$ on $\mathcal{B}$. Let us consider the following cases:

- If $v_1 v_2 \cdots v_j$ is a branch of $D_a^\gamma(\mathcal{B})$ and $\forall i \in [1, j], \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma') = \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma)$.
  Then we have $\mathtt{W}_{\mathtt{Ch}}(v_1, \gamma')\mathtt{W}_{\mathtt{Ch}}(v_2, \gamma') \cdots \mathtt{W}_{\mathtt{Ch}}(v_j, \gamma') = \mathtt{W}_{\mathtt{Ch}}(v_1, \gamma)\mathtt{W}_{\mathtt{Ch}}(v_2, \gamma) \cdots \mathtt{W}_{\mathtt{Ch}}(v_j, \gamma)$ and thus $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \preceq_\alpha \mathtt{W}_{\mathtt{Ch}}(D_a^\gamma(\mathcal{B}), \gamma)$.
- If $v_1 v_2 \cdots v_j$ is a branch of $D_a^\gamma(\mathcal{B})$ and $\exists i \in [1, j] : \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma') \neq \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma)$.
  Let us consider the smallest i such that $\mathtt{W}_{\mathtt{Ch}}(v_i, \gamma') \neq \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma)$. According to Lemma 5.17, $\mathtt{W}_{\mathtt{Ch}}(v_i, \gamma') < \mathtt{W}_{\mathtt{Ch}}(v_i, \gamma)$, and as a consequence, $\mathtt{W}_{\mathtt{Ch}}(v_1, \gamma')\mathtt{W}_{\mathtt{Ch}}(v_2, \gamma') \cdots \mathtt{W}_{\mathtt{Ch}}(v_j, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(v_1, \gamma)\mathtt{W}_{\mathtt{Ch}}(v_2, \gamma) \cdots \mathtt{W}_{\mathtt{Ch}}(v_j, \gamma)$. Thus, we have $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \prec_\alpha \mathtt{W}_{\mathtt{Ch}}(D_a^\gamma(\mathcal{B}), \gamma)$.
- If $v_1 v_2 \cdots v_j$ is not a branch of D_a^γ.
  Let us consider $v_1 v_2 \cdots v_l = D_a^\gamma(\mathcal{B})$ the projection of D_a^γ on $\mathcal{B}$. Since $v_1 v_2 \cdots v_j$ is not a branch of D_a^γ we have $1 \leq l < j$, and thus $l+1 \leq j$. Remark first that since (v_l, v_{l+1}) is an edge in $D_a^{\gamma'}$, we have $\mathtt{tok}_{v_l}^{\gamma'} \notin \{\uparrow, \bot\}$, and thus $\mathtt{tok}_{v_l}^\gamma \neq \uparrow$. But $v_{l+1} \notin D_a^\gamma$, so $\mathtt{tok}_{v_{l+1}}^\gamma = \bot$, and since $v_{l+1} \in D_a^{\gamma'}$, $\mathtt{tok}_{v_{l+1}}^{\gamma'} \neq \bot$ so v_{l+1} executes $\mathbb{R}_{\mathtt{Win}}$ during cs. Thus, Lemma 5.16 applies, and as a consequence $\mathtt{W}_{\mathtt{Ch}}(v_l, \gamma') < \mathtt{W}_{\mathtt{Ch}}(v_l, \gamma)$.

Let us consider the smallest $i \leq l$ such that $\mathtt{W_{Ch}}(v_i, \gamma') \neq \mathtt{W_{Ch}}(v_i, \gamma)$. According to Lemma 5.17, $\mathtt{W_{Ch}}(v_i, \gamma') < \mathtt{W_{Ch}}(v_i, \gamma)$, and as a consequence,

$$\mathtt{W_{Ch}}(v_1, \gamma')\mathtt{W_{Ch}}(v_2, \gamma') \cdots \mathtt{W_{Ch}}(v_j, \gamma') \prec \mathtt{W_{Ch}}(v_1, \gamma)\mathtt{W_{Ch}}(v_2, \gamma) \cdots \mathtt{W_{Ch}}(v_l, \gamma).$$

Thus, we have $\mathtt{W_{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \prec_\alpha \mathtt{W_{Ch}}(D_a^{\gamma}(\mathcal{B}), \gamma)$.

We prove that for any maximal branch $v_1 v_2 \cdots v_k$ of G_t, $\mathtt{W_{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \preceq_\alpha \mathtt{W_{Ch}}(D_a^{\gamma}(\mathcal{B}), \gamma)$. By definition, we have $\mathtt{W_{Ch}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W_{Ch}}(D_a^{\gamma}, \gamma)$. ■

Theorem 5.4 and Lemma 5.18 prove that under certain circumstances, we are certain that the weight of one $\mathtt{CD}^o$ decreases. The first theorem focuses on the application of rules $\mathbb{R}_{\mathtt{Win}}$ and $\mathbb{R}_{\mathtt{FakeWin}}$, and will be useful to prove that $\mathtt{W}$ is an admissible weight function, but also to prove that the other, longer, prefixes of $\mathtt{W}$ are pre-admissible. Indeed, if in any computing step, one node executes $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$, we do not need to look beyond the first component to be assured that the weight does not increase, since $\preceq^k$ is a lexicographic order. The second lemma focuses on the length of the branches of a $\mathtt{CD}^o$, and is useful for us to write proofs in the following. Just as we explained before, we can now suppose in the following proofs of pre-admissibility that branches of $\mathtt{CD}^o$'s never increase, since it would make the first component of $\mathtt{W}$ decrease.

Theorem 5.4

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs. If $\exists v \in \mathbb{B}(D_a^\gamma)$ such that v executes $\mathbb{R}_{\mathtt{Win}}$, $\mathbb{R}_{\mathtt{FakeWin}}$, or $\mathbb{R}_{\mathtt{ReplayUp}}$ during cs, we have $\mathtt{W_{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W_{Ch}}(D_a^{\gamma}, \gamma)$.

Proof: Let us consider one node $v \in \mathbb{B}(D_a^\gamma)$ that executes $\mathbb{R}_{\mathtt{Win}}$, $\mathbb{R}_{\mathtt{FakeWin}}$, or $\mathbb{R}_{\mathtt{ReplayUp}}$ during cs. By definition, $\exists p \in \mathcal{P}(v) : p \in D_a^\gamma \wedge \mathtt{tok}_v^\gamma \neq \uparrow$. Let us consider $\mathcal{B} = v_1 v_2 \cdots v_k$ a branch of D_a^γ such that $v_k = p$. According to Lemma 5.10, $\mathcal{B}$ is a branch of $D_a^{\gamma'}$.

According to Lemma 5.16, $\mathtt{W_{Ch}}(p, \gamma') < \mathtt{W_{Ch}}(p, \gamma)$, and according to Theorem 5.3, $\mathtt{W_{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \preceq_\alpha \mathtt{W_{Ch}}(D_a^{\gamma}(\mathcal{B}), \gamma)$. Consequently, $\mathtt{W_{Ch}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \prec_\alpha \mathtt{W_{Ch}}(D_a^{\gamma}(\mathcal{B}), \gamma)$, and thus according to Theorem 5.3, $\mathtt{W_{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W_{Ch}}(D_a^{\gamma}, \gamma)$. ■

Lemma 5.18

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs. If there exists $\mathcal{B} = v_1 v_2 \cdots v_k$ a branch of G_a such that $D_a^{\gamma'}(\mathcal{B})$ is longer than $D_a^\gamma(\mathcal{B})$, then $\mathtt{W_{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W_{Ch}}(D_a^{\gamma}, \gamma)$.

Proof: Let us consider $v_1 v_2 \cdots v_j = D_a^\gamma(\mathcal{B})$. Let us prove that $v = v_{j+1}$ satisfies the conditions of Theorem 5.4.

Remark first that $\mathtt{tok}_{v_i}^\gamma \neq \uparrow$. Indeed, if $\mathtt{tok}_{v_i}^\gamma = \uparrow$ then $\mathtt{tok}_{v_i}^{\gamma'} \in \{\uparrow, \perp\}$, which is contradictory since $v \in \mathcal{C}_{D_a^{\gamma'}}(v_i)$. Then, we have $\mathtt{tok}_v^\gamma = \perp$ which implies $v \in \mathbb{B}(D_a^\gamma)$, otherwise we would have $v \in \mathcal{C}_{D_a^\gamma}(v_i)$. But since $v \in D_a^{\gamma'}$, we have $\mathtt{tok}_v^{\gamma'} \neq \perp$, which is possible only if v executes $\mathbb{R}_{\mathtt{Win}}$ during cs.

We can apply Theorem 5.4 to our situation, and consequently $\mathtt{W_{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W_{Ch}}(D_a^{\gamma}, \gamma)$. ■

5.4.2.5 Second component of $\mathtt{W}$: $\mathtt{W_{Er}}$ errors descending from v

Explanations One major quantity in our algorithm that should decrease during an execution is the number of nodes v such that $\mathtt{Er}(v)$. Indeed, we expect that after some convergence time, no node will execute $\mathbb{E}_{\mathtt{TrustChild}}$, and that our $\mathtt{CD}^o$ will have the expected

shape. Although that last sentence will be proven true in the following, the number of nodes v such that $\texttt{Er}(v)$ might occasionally increases. Indeed when a node executes $\mathbb{E}_{\texttt{TrustChild}}$, it might happen that it creates an error in some of its parents. For example, if node v has p as a parent, and $\texttt{tok}_v^\gamma = \bigstar, \texttt{tok}_p^\gamma = \downdownarrows$. We have $\neg\texttt{Er}^\gamma(p)$, but if in $\gamma \to \gamma', v$ executes $\mathbb{E}_{\texttt{TrustChild}}$, then $\texttt{tok}_v^{\gamma'} = \downarrow$ and thus $\texttt{Er}^{\gamma'}(p)$. In the described case, the number of nodes in error does not increase, but if node v has several parents p with $\texttt{tok}_p^\gamma \in \{\downdownarrows, \circlearrowleft\}$ then that number will actually increase.

One first idea to prevent this from happening is that each node counts the number of descendants it has that are in error. This almost works, but not totally. Indeed, suppose v has two parents p, that both have the same parent a, and we have $\texttt{tok}_v^\gamma = \bigstar$, $\texttt{tok}_p^\gamma = \downdownarrows$, $\texttt{tok}_a^\gamma = \downarrow$. In γ, a only has one descendant in error, but has two once v executes $\mathbb{E}_{\texttt{TrustChild}}$. The fact that errors can split through several branches forces us to design $\texttt{W}_{\texttt{Er}}$ such that it counts a descendant in error as many times as there exist paths to that node.

Fortunately, we can stop the count in our descendants as soon as we reach a node v such that $\texttt{tok}_v \in \{\bot, \uparrow, \downarrow\}$. Indeed, such nodes cannot execute $\mathbb{E}_{\texttt{TrustChild}}$, which implies that errors cannot ride up through such nodes.

There exists another situation that increases the number of nodes in error. Suppose one node v executes $\mathbb{R}_{\texttt{Win}}$ during $\gamma \to \gamma'$. We have $\texttt{W}_{\texttt{Er}}(v, \gamma) = 0$ by definition. Yet, it might happen that one child u of v is such that $\texttt{tok}_u^{\gamma'} \neq \bot$. It might even happen that $\texttt{W}_{\texttt{Er}}(u, \gamma') > 0$, with an arbitrary value. In such a situation, the weight of v, and thus $\texttt{W}_{\texttt{Er}}(D_a^\gamma)$ can increase arbitrarily. Hopefully, we proved in the previous section that at the same moment, $\texttt{W}_{\texttt{Ch}}(D_a^\gamma)$ decreases, and thus no problem is raised.

Definition According to what precedes, we define $\texttt{W}_{\texttt{Er}}$ the second component of our weight function by 0 on nodes v such that $\texttt{tok}_v \in \{\downarrow, \uparrow, \bot\}$, and by the number of nodes in error they can see within range, themselves included, otherwise.

$$\texttt{W}_{\texttt{Er}}(v, \gamma) = \begin{cases} 0 & \text{if } \texttt{tok}_v^\gamma \in \{\downarrow, \uparrow, \bot\} \\ \sum_{u \in \mathcal{C}(v)} \texttt{W}_{\texttt{Er}}(u, \gamma) + 1 & \text{if } \texttt{Er}^\gamma(v) \\ \sum_{u \in \mathcal{C}(v)} \texttt{W}_{\texttt{Er}}(u, \gamma) & \text{otherwise} \end{cases} \tag{5.38}$$

Proofs Lemma 5.19 describes under which conditions one node that is not in error before a computing step can become in error after that computing step. It states that either it makes $\texttt{W}_{\texttt{Ch}}$ decrease, either it comes from the fact that one child of that node executed $\mathbb{E}_{\texttt{TrustChild}}$, which does not lead to any increase of $\texttt{W}_{\texttt{Er}}$. This lemma is the main argument to establish that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})$ is a pre-admissible weight function.

Lemma 5.19

Let $v \in V$ be a node and let $cs = \gamma \to \gamma'$ be a computing step. If $\neg\texttt{Er}^\gamma(v)$ and $\texttt{Er}^{\gamma'}(v)$, then $\texttt{W}_{\texttt{Ch}}(v, \gamma') < \texttt{W}_{\texttt{Ch}}(v, \gamma)$ or v executes $\mathbb{R}_{\texttt{Win}}$ during cs or $\exists u \in \mathcal{C}(v)$ such that u executes $\mathbb{E}_{\texttt{TrustChild}}$ during cs.

Proof: We prove the equivalent statement: if $\neg\texttt{Er}^\gamma(v)$ then $\neg\texttt{Er}^{\gamma'}(v)$ or $\texttt{W}_{\texttt{Ch}}(v, \gamma') < \texttt{W}_{\texttt{Ch}}(v, \gamma)$ or, during cs, v executes $\mathbb{R}_{\texttt{Win}}$ or $\exists u \in \mathcal{C}(v)$ such that u executes $\mathbb{E}_{\texttt{TrustChild}}$.

Since $\neg\texttt{Er}^\gamma(v)$, only few cases must be considered.

1. If $\texttt{tok}_v^\gamma = \bot$ then either v executes $\mathbb{R}_{\texttt{Win}}$ during cs, either it does not, but then $\texttt{tok}_v^{\gamma'} = \bot$ and thus $\neg\texttt{Er}^{\gamma'}(v)$.
2. If $\texttt{tok}_v^\gamma = \uparrow$ then $\texttt{tok}_v^{\gamma'} \in \{\uparrow, \bot\}$ and thus $\neg\texttt{Er}^{\gamma'}(v)$.

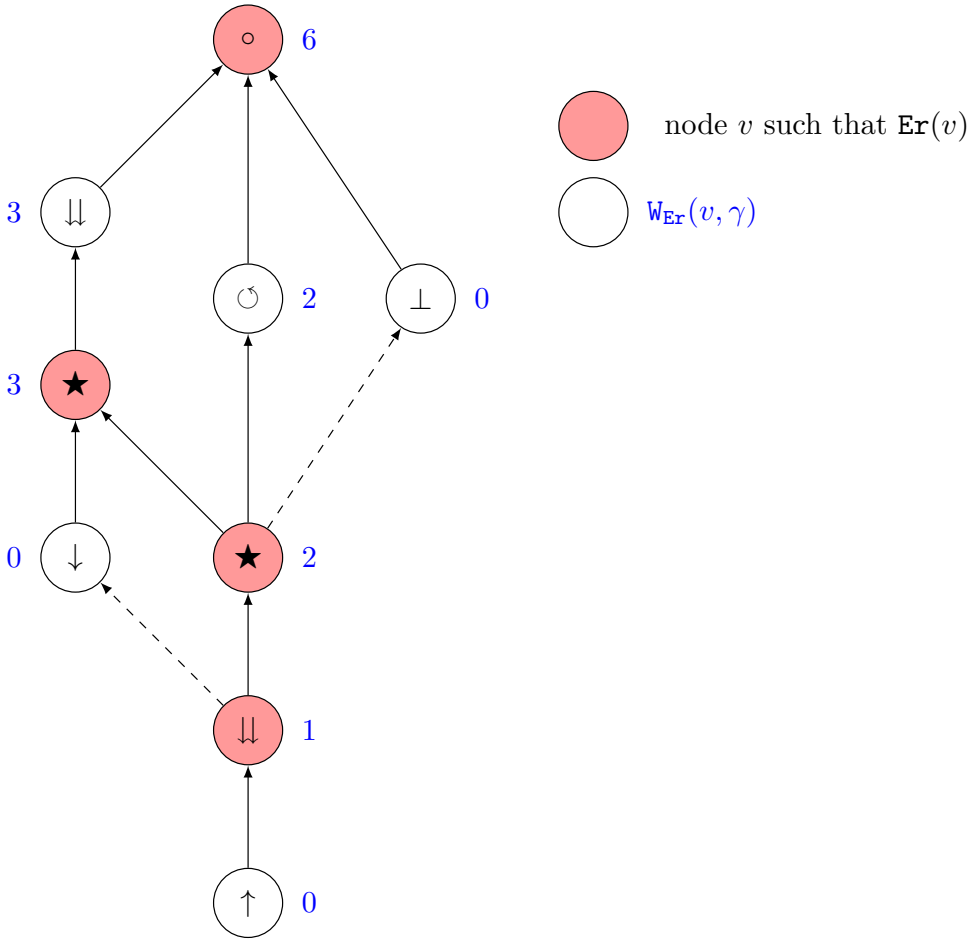

Figure 5.26: Example of the definition of $\mathtt{W}_{\mathtt{Er}}$ on a $\mathtt{D}^o$.

3. If $\mathtt{tok}_v^\gamma = \downarrow$ then if v does not execute $\mathbb{R}_{\mathtt{Receive}}$ during cs, $\mathtt{tok}_v^{\gamma'} = \downarrow$ and thus $\neg\mathtt{Er}^{\gamma'}(v)$. Otherwise, v executes $\mathbb{R}_{\mathtt{Receive}}$ during cs and then $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma \in \{\bot, \uparrow\}$. Either one child u of v executes $\mathbb{R}_{\mathtt{Win}}$, and then $\mathtt{W}_{\mathtt{Ch}}(v,\gamma') < \mathtt{W}_{\mathtt{Ch}}(v,\gamma)$ according to Lemma 5.16, either no one does, but then $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} \in \{\bot, \uparrow\}$, and since $\mathtt{tok}_v^{\gamma'} = \circ$, we have $\neg\mathtt{Er}^{\gamma'}(v)$.
4. If $\mathtt{tok}_v^\gamma = \bigstar \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma = \bot$ then either one child u of v executes $\mathbb{R}_{\mathtt{Win}}$, and then $\mathtt{W}_{\mathtt{Ch}}(v,\gamma') < \mathtt{W}_{\mathtt{Ch}}(v,\gamma)$ according to Lemma 5.16, either no one does, but then $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} = \bot$ which means that $\neg\mathtt{Er}^{\gamma'}(v)$.
5. If $\mathtt{tok}_v^\gamma = \bullet \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma = \bot$ then children u of v cannot execute $\mathbb{R}_{\mathtt{Win}}$ during cs: either u has one other parent p such that $\mathtt{tok}_p^\gamma = \downdownarrows$, and then $\neg\mathtt{OkNeg}^\gamma(u)$, either it has not, and thus it cannot execute $\mathbb{R}_{\mathtt{Win}}$ either. Thus, $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} = \bot$, so $\neg\mathtt{Er}^{\gamma'}(v)$.
6. If $\mathtt{tok}_v^\gamma = \downdownarrows \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma \in \{\bot, \bigstar\}$ then children u of v cannot execute $\mathbb{R}_{\mathtt{Nego}}$ since they have one parent v with $\mathtt{tok}_v^\gamma = \downdownarrows$. If no children u of v executes $\mathbb{E}_{\mathtt{TrustChild}}$ during cs, then $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} \in \{\bot, \bigstar\}$. Furthermore, $\mathtt{tok}_v^{\gamma'} \in \{\downdownarrows, \circlearrowleft\}$ since $\neg\mathtt{Er}^\gamma(v)$. Thus, if no children u of v executes $\mathbb{E}_{\mathtt{TrustChild}}$ during cs, $\neg\mathtt{Er}^{\gamma'}(v)$.
7. If $\mathtt{tok}_v^\gamma = \circlearrowleft \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^\gamma \in \{\bot, \bigstar\}$ then children u of v cannot execute $\mathbb{R}_{\mathtt{Nego}}$ since they have one parent v with $\mathtt{tok}_v^\gamma = \circlearrowleft$. If no children u of v executes $\mathbb{E}_{\mathtt{TrustChild}}$

during cs, then $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} \in \{\bot, \bigstar\}$. Furthermore, $\mathtt{tok}_v^{\gamma'} \in \{\circlearrowleft, \downarrow\}$. Thus, if no children u of v executes $\mathbb{E}_{\mathtt{TrustChild}}$ during cs, $\neg\mathtt{Er}^{\gamma'}(v)$.

8. If $\mathtt{tok}_v^{\gamma} = \circ \wedge \forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma} \in \{\bot, \uparrow\}$ then children u of v cannot execute $\mathbb{R}_{\mathtt{Win}}$ during cs: either u has one other parent p such that $\mathtt{tok}_p^{\gamma} = \downdownarrows$, and then $\neg\mathtt{OkNeg}^{\gamma}(u)$, either it has not, and thus it cannot execute $\mathbb{R}_{\mathtt{Win}}$ either. This implies that $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} \in \{\bot, \uparrow\}$ and thus $\neg\mathtt{Er}^{\gamma'}(v)$. ■

Theorem 5.5 establishes that $(\mathtt{W}_{\mathtt{Ch}}, \mathtt{W}_{\mathtt{Er}})$ is pre-admissible.

Theorem 5.5

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs.

We have $(\mathtt{W}_{\mathtt{Ch}}, \mathtt{W}_{\mathtt{Er}})(D_a^{\gamma'}, \gamma') \preceq^2 (\mathtt{W}_{\mathtt{Ch}}, \mathtt{W}_{\mathtt{Er}})(D_a^{\gamma}, \gamma)$.

Proof: According to Theorem 5.3, we already have $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$. Consequently, by definition of the lexicographic order, we only have to establish that if $\mathtt{W}_{\mathtt{Er}}(D_a^{\gamma}, \gamma) \prec \mathtt{W}_{\mathtt{Er}}(D_a^{\gamma'}, \gamma')$ then $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$.

Let us first remark that if there exists $\mathcal{B}$ a branch of G_a such that $D_a^{\gamma'}(\mathcal{B})$ is longer than $D_a^{\gamma}(\mathcal{B})$ then according to Lemma 5.18, $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$.

We suppose now that for any branch $\mathcal{B}$ of G_a, $D_a^{\gamma'}(\mathcal{B})$ is not longer than $D_a^{\gamma}(\mathcal{B})$. Let us establish that $\forall v \in D_a^{\gamma'}, \mathtt{W}_{\mathtt{Er}}(v, \gamma') \leq \mathtt{W}_{\mathtt{Er}}(v, \gamma)$, or $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$. Since the branches of $D_a^{\gamma'}$ are included in the branches of D_a^{γ}, this becomes $\mathtt{W}_{\mathtt{Er}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W}_{\mathtt{Er}}(D_a^{\gamma}, \gamma)$ or $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$, and thus this actually proves the theorem.

Let us prove that if $\exists v \in D_a^{\gamma'} : \mathtt{W}_{\mathtt{Er}}(v, \gamma') > \mathtt{W}_{\mathtt{Er}}(v, \gamma)$ then $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$, and let us consider one such v with highest depth. Let us first prove that $\mathtt{tok}_v^{\gamma} \notin \{\bot, \uparrow, \downarrow\}$. Since $v \in D_a^{\gamma}$, $\mathtt{tok}_v^{\gamma} \neq \bot$. If $\mathtt{tok}_v^{\gamma} = \uparrow$ then $\mathtt{tok}_v^{\gamma'} \in \{\uparrow, \bot\}$, and thus $\mathtt{W}_{\mathtt{Er}}(v, \gamma') = 0$ which is contradictory with our hypothesis, so $\mathtt{tok}_v^{\gamma} \neq \uparrow$. Finally, if $\mathtt{tok}_v^{\gamma} = \downarrow$ then $\mathtt{W}_{\mathtt{Er}}(v, \gamma') > 0$ is possible only if v executes $\mathbb{R}_{\mathtt{Receive}}$ during cs, which implies that $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma} \in \{\bot, \uparrow\}$. None of these children executes $\mathbb{R}_{\mathtt{Win}}$ during cs, because if one u does then the edge (v, u) belongs to $D_a^{\gamma'}$ (because $\mathtt{tok}_v^{\gamma'} = \circ$) while it does not belong to D_a^{γ}, which is contradictory with our hypothesis on branches. Consequently, $\forall u \in \mathcal{C}(v), \mathtt{tok}_u^{\gamma'} \in \{\bot, \uparrow\}$, and thus $\neg\mathtt{Er}^{\gamma'}(v)$, and $\mathtt{W}_{\mathtt{Er}}(v, \gamma') = 0$. Thus, we can indeed assume that $\mathtt{tok}_v^{\gamma} \notin \{\bot, \uparrow, \downarrow\}$.

Since all branches of $D_a^{\gamma'}$ are also branches of D_a^{γ}, and since v is one deepest node of $D_a^{\gamma'}$ such that $\mathtt{W}_{\mathtt{Er}}(v, \gamma') > \mathtt{W}_{\mathtt{Er}}(v, \gamma)$, we have $\forall u \in \mathcal{C}(v), \mathtt{W}_{\mathtt{Er}}(u, \gamma') \leq \mathtt{W}_{\mathtt{Er}}(u, \gamma)$. As a consequence, $\mathtt{W}_{\mathtt{Er}}(v, \gamma') > \mathtt{W}_{\mathtt{Er}}(v, \gamma)$ becomes possible only if $\neg\mathtt{Er}^{\gamma}(v)$ and $\mathtt{Er}^{\gamma'}(v)$.

According to Lemma 5.19, this is possible only if during cs, v executes $\mathbb{R}_{\mathtt{Win}}$, which cannot happen since $\mathtt{tok}_v^{\gamma} \neq \bot$, or if $\mathtt{W}_{\mathtt{Ch}}(v, \gamma') < \mathtt{W}_{\mathtt{Ch}}(v, \gamma)$ which implies that $\mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W}_{\mathtt{Ch}}(D_a^{\gamma}, \gamma)$, or if $\exists u \in \mathcal{C}(v)$ such that u executes $\mathbb{E}_{\mathtt{TrustChild}}$ during cs. In that last case, we have $\mathtt{Er}^{\gamma}(u)$ so $\mathtt{W}_{\mathtt{Er}}(u, \gamma) \geq 1$, and $\mathtt{tok}_u^{\gamma'} = \downarrow$ and so $\mathtt{W}_{\mathtt{Er}}(u, \gamma') = 0$. Since we took v one deepest node such that $\mathtt{W}_{\mathtt{Er}}(v, \gamma') > \mathtt{W}_{\mathtt{Er}}(v, \gamma)$, we have $\forall u \in \mathcal{C}(v), \mathtt{W}_{\mathtt{Er}}(u, \gamma') \leq \mathtt{W}_{\mathtt{Er}}(u, \gamma)$, so, by considering the child that executes $\mathbb{E}_{\mathtt{TrustChild}}$, we have $\sum_{u \in \mathcal{C}(v)} \mathtt{W}_{\mathtt{Er}}(u, \gamma') < \sum_{u \in \mathcal{C}(v)} \mathtt{W}_{\mathtt{Er}}(u, \gamma)$, and thus $\mathtt{W}_{\mathtt{Er}}(v, \gamma') \leq \mathtt{W}_{\mathtt{Er}}(v, \gamma)$. ■

Theorem 5.6 proves that under certain circumstances, we are certain that the weight of one $\mathtt{CD}^o$ decreases. It will be useful to prove that $\mathtt{W}$ is an admissible function, but also to prove that the other, longer, prefixes of $\mathtt{W}$ are pre-admissible. Indeed, if in any computing step one node executes $\mathbb{E}_{\mathtt{TrustChild}}$ then we do not need to look beyond the second component to be assured that the weight does not increase, since $\preceq^k$ is a lexicographic order.

Theorem 5.6

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a

does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ *during cs. If* $\exists v \in D_a^\gamma$ *such that* v *executes* $\mathbb{E}_{\texttt{TrustChild}}$ *during cs, then* $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^\gamma, \gamma)$.

Proof: Let us consider one such node $v \in D_a^\gamma$, and one branch $\mathcal{B}$ of G_a such that $v \in D_a^\gamma(\mathcal{B})$. If $v \notin D_a^{\gamma'}(\mathcal{B})$, then according to Lemma 5.18, we have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^\gamma, \gamma)$.

Else, remark that $\texttt{W}_{\texttt{Er}}(v, \gamma) > 1$ since $\texttt{Er}^\gamma(v)$, and $\texttt{W}_{\texttt{Er}}(v, \gamma') = 0$ since $\texttt{tok}_v^\gamma = \downarrow$. Thus, $\texttt{W}_{\texttt{Er}}(v, \gamma') < \texttt{W}_{\texttt{Er}}(v, \gamma)$, and so $\texttt{W}_{\texttt{Er}}(D_a^{\gamma'}(\mathcal{B}), \gamma') \prec_\alpha \texttt{W}_{\texttt{Er}}(D_a^\gamma(\mathcal{B}), \gamma)$ since $v \in D_a^{\gamma'}(\mathcal{B})$. Since $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \preceq^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^\gamma, \gamma)$, we conclude $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^\gamma, \gamma)$. ■

5.4.2.6 Third component of W: $\texttt{W}_{\texttt{Circ}}$, circulation of variable tok

Explanations Now that we have considered the cases where nodes execute $\mathbb{E}_{\texttt{TrustChild}}$, we can consider that variable tok is updated in accordance with Figure 5.7. The third component of the weight function should treat all the transitions of variable tok depicted in this diagram. We could think at first sight that since the diagram is circular, we will have difficulty defining a weight function that decreases along the diagram. Yet, if the algorithm behaves correctly, when a node v is updated from $\bullet$ to $\downarrow\downarrow$, one child of v is supposed to execute $\mathbb{R}_{\texttt{Win}}$ in the next computing steps, and then $\texttt{W}_{\texttt{Ch}}$ will decrease on the observed $\texttt{CD}^o$. Since $\texttt{W}_{\texttt{Ch}}$ has higher priority in the lexicographic order than $\texttt{W}_{\texttt{Circ}}$, we are going to design $\texttt{W}_{\texttt{Circ}}$ so that it increases exactly when $\texttt{W}_{\texttt{Ch}}$ decreases. Therefore, we can cycle as many times as necessary through the transition diagram of Figure 5.7 and having the 3-tuple $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})$ decrease at each step.

To do this, we must associate a different weight to a node v with $\texttt{tok}_v = \downarrow\downarrow$ depending on whether it still has one child that is going to execute $\mathbb{R}_{\texttt{Win}}$ (or $\mathbb{R}_{\texttt{FakeWin}}$) in the next computing steps. More precisely, if there still exists one child u of v that belongs to $\texttt{Players}(v)$, we associate a lower weight to v that if such child u does not exist. When v executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, $\texttt{W}_{\texttt{Circ}}$ increases on v, but $\texttt{W}_{\texttt{Ch}}$ decreases at the same time.

But since we work in the framework of self-stabilization, it might happen that registers are incorrectly initialized, and that when node v executes $\mathbb{R}_{\texttt{Give}}$, and updates $\texttt{tok}_v$ from $\bullet$ to $\downarrow\downarrow$, v immediately has no children susceptible to execute $\mathbb{R}_{\texttt{Win}}$. Indeed, if in the initial configuration, $\texttt{id}_v$ is inconsistent with the id of its children, it might happen that after some computations, all the children u of v such that $\texttt{play}_u = \texttt{P}$ eventually lose the negotiation. In such a situation, after v executes $\mathbb{R}_{\texttt{Give}}$, we arrive in a configuration in which $\texttt{W}_{\texttt{Circ}}(v, \gamma')$ is already at its high value. But this high value is precisely designed to be the highest assigned value of the diagram. To overcome that issue, we add one possible value for $\texttt{W}_{\texttt{Circ}}(v, \gamma)$ when $\texttt{tok}_v^\gamma = \bullet$, higher than the high value for nodes with $\texttt{tok}_v^\gamma = \downarrow\downarrow$. This value is taken by $\texttt{W}_{\texttt{Circ}}(v, \gamma)$ only if we detect that after few computations, we fall into a situation where no children of v can execute $\mathbb{R}_{\texttt{Win}}$. It is crucial that this situation cannot happen consecutively to an execution of $\mathbb{R}_{\texttt{ReNego}}$, otherwise we should also design two possible outcomes for the weight of some node v with $\texttt{tok}_v = \circ$. The predicate FullRound achieves that detection for node v such that $\texttt{tok}_v = \bullet$.

For the other values that $\texttt{tok}_v$ might take, we only associate integers that decrease along the classical scheme depicted in Figure 5.7. The definition of $\texttt{W}_{\texttt{Circ}}$ is summarized in Figure 5.27

Definitions The weight of one node v such that $\texttt{tok}_v = \downarrow\downarrow$ depends on if it has children susceptible to take the token. If it has, then we set its weight to one low value, so that when the child takes the token, $\texttt{W}_{\texttt{Ch}}$ decreases on v at the same time that we make $\texttt{W}_{\texttt{Circ}}$ increases on v. If it has not, then it must be a high value so that the weight of v can decrease through

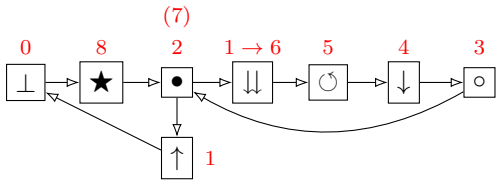

Figure 5.27: Value of $\mathtt{W}_{\mathtt{Circ}}$ depending on the variable $\mathtt{tok}_v$.

the transitions of Figure 5.27.

$$\mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}(v,\gamma) = \begin{cases} 6 & \text{if } \mathtt{Players}^{\gamma}(v) = \emptyset \\ 1 & \text{if } \mathtt{Players}^{\gamma}(v) \neq \emptyset \end{cases} \tag{5.39}$$

Let us now define the weight of one node v such that $\mathtt{tok}_v = \bullet$. The weight of such a node can also be taken between two values that are 2 and 7, so that it can easily be bigger than $\mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}$. We must design $\mathtt{W}^{\bullet}_{\mathtt{Circ}}$ such that, if after v executes $\mathbb{R}_{\mathtt{Give}}$ we have $\mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}(v,\gamma') = 6$ then we have $\mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma) = 7$ just before, otherwise the weight would increase. But we also require that we never have $\mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma) = 2 \wedge \mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma') = 7$. In other words, if $\mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}$ is set to 6 immediately after the execution of $\mathbb{R}_{\mathtt{Give}}$, it must be detected at the very beginning of the execution (or as soon as $\mathtt{tok}_v$ is set to $\bullet$).

The situation described just above happens if before v executes $\mathbb{R}_{\mathtt{Give}}$, all the nodes $u \in \mathtt{Players}^{\gamma}(v)$ execute $\mathbb{R}_{\mathtt{Lose}}$. In such a situation, no children u of v wins the negotiation, and v executes a full cycle of the cycle of Figure 5.27, which indeed forces us to attribute 7 to the weight of v. Hopefully, the design of our algorithm guarantees that this can only happen if the values of $\mathtt{id}_v$ and/or $\mathtt{id}_u$ are inconsistent. In other words, as soon as v executes one action, we guarantee that this situation cannot occur anymore, this will be proven in Lemma 5.23. Thus, we only have to deduce if $u \in \mathtt{Players}(v)$ will execute $\mathbb{R}_{\mathtt{Lose}}$ before any action of v, from the values of $\mathtt{id}_v$ and $\mathtt{id}_u$. This strongly depends on the value of $\mathtt{b}^{\gamma}_v$.

If $\mathtt{b}^{\gamma}_v \neq \bot$ then v announces one value that defines the partial winners of that turn of the negotiation. It the configuration is inconsistent, it is very possible that no children $u \in \mathtt{Players}^{\gamma}(v)$ has produced, neither will produce that answer. All of the nodes that have announced a different value than $\mathtt{b}^{\gamma}_v$, or that will announce a different value than $\mathtt{b}^{\gamma}_v$ will execute $\mathbb{R}_{\mathtt{Lose}}$, immediately for the first ones, and after one execution of $\mathbb{R}_{\mathtt{NewBit}}$ for the second ones

$$\mathtt{Losers}^{\gamma}_{\neg\bot}(v) = \{u \in \mathtt{Players}^{\gamma}(v), \begin{cases} \mathtt{ph}^{\gamma}_u = \mathtt{ph}^{\gamma}_v \\ \mathtt{b}^{\gamma}_u \neq \mathtt{b}^{\gamma}_v \end{cases} \vee \begin{cases} \mathtt{ph}^{\gamma}_u \neq \mathtt{ph}^{\gamma}_v \\ \mathtt{Bit}_u(\mathtt{ph}^{\gamma}_v) \neq \mathtt{b}^{\gamma}_v \end{cases} \} \tag{5.40}$$

If $\mathtt{b}^{\gamma}_v = \bot$ then v is waiting for all of its children $u \in \mathtt{Players}^{\gamma}(v)$ to announce $\mathtt{Bit}_u(\mathtt{ph}^{\gamma}_v)$. Therefore, no children $u \in \mathtt{Players}^{\gamma}(v)$ can execute $\mathbb{R}_{\mathtt{Lose}}$ before an action of v. We define the set $\mathtt{Losers}(v)$ that depends on the value of $\mathtt{b}_v$.

$$\mathtt{Losers}^{\gamma}(v) = \begin{cases} \mathtt{Losers}^{\gamma}_{\neg\bot}(v) & \text{if } \mathtt{b}^{\gamma}_v \neq \bot \\ \emptyset & \text{if } \mathtt{b}^{\gamma}_v = \bot \end{cases} \tag{5.41}$$

We are now tempted to say that $\mathtt{W}^{\gamma}_{\mathtt{Circ}}(v) = 7$ if and only if $\mathtt{Losers}^{\gamma}(v) = \mathtt{Players}^{\gamma}(v)$. Yet, one subtlety must be treated before. Indeed, when all the children u of v have received the token, then u will not execute $\mathbb{R}_{\mathtt{Give}}$ but $\mathbb{R}_{\mathtt{OfferUp}}$, and thus we do not have to reason

based on the constraint of $\mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}$. Moreover, in that situation, we must not attribute a weight of 7 to v since it would be an increase when v executes $\mathbb{R}_{\mathtt{ReNego}}$, and updates $\mathtt{tok}_v$ from $\circ$ to $\bullet$. This situation cannot happen as long as there remains players, because after the execution of $\mathtt{StartNego}(v)$, we have $\mathtt{Losers}(v) = \emptyset$. The only case to consider is therefore when no node children of v is susceptible to receive the token, and in that situation we force $\mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma)$ to be 2.

In order to make the proof easier to approach, we first define the set $\mathtt{Candidates}(v)$ that contains all the nodes that still have to negotiate and win the token before the circulation from v is over. This set reminds one of the different cases depicted in Figure 5.24.

$$\mathtt{Candidates}^{\gamma}(v) = \{u \in \mathcal{C}_{\mathtt{neq}}{}^{\gamma}(v) : \mathtt{tok}^{\gamma}_u = \bot \wedge \mathtt{play}^{\gamma}_u \in \{\mathtt{P},\mathtt{L}\}\} \tag{5.42}$$

We can now define the main predicate if that section, `FullRound`, which decides whether v has to execute one full cycle before any of its children receives the token.

$$\mathtt{FullRound}^{\gamma}(v) \equiv (\mathtt{Losers}^{\gamma}(v) = \mathtt{Players}^{\gamma}(v)) \wedge (\mathtt{Candidates}^{\gamma}(v) \neq \emptyset) \tag{5.43}$$

We deduce function $\mathtt{W}^{\bullet}_{\mathtt{Circ}}$ that allows us to define $\mathtt{W}_{\mathtt{Circ}}$.

$$\mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma) = \begin{cases} 7 & \text{if } \mathtt{FullRound}^{\gamma}(v) \\ 2 & \text{otherwise} \end{cases} \tag{5.44}$$

$$\mathtt{W}_{\mathtt{Circ}}(v,\gamma) = \begin{cases} 0 & \text{if } \mathtt{tok}^{\gamma}_v = \bot \\ 8 & \text{if } \mathtt{tok}^{\gamma}_v = \bigstar \\ \mathtt{W}^{\bullet}_{\mathtt{Circ}}(v,\gamma) & \text{if } \mathtt{tok}^{\gamma}_v = \bullet \\ \mathtt{W}^{\Downarrow\Downarrow}_{\mathtt{Circ}}(v,\gamma) & \text{if } \mathtt{tok}^{\gamma}_v = \Downarrow\Downarrow \\ 5 & \text{if } \mathtt{tok}^{\gamma}_v = \circlearrowleft \\ 4 & \text{if } \mathtt{tok}^{\gamma}_v = \downarrow \\ 3 & \text{if } \mathtt{tok}^{\gamma}_v = \circ \\ 1 & \text{if } \mathtt{tok}^{\gamma}_v = \uparrow \end{cases} \tag{5.45}$$

Proofs Lemma 5.20 establishes that actions that make the negotiation progress are non simultaneous on the parent and on its children.

Lemma 5.20

Let γ be a configuration and let $v \in V$ be a node such that $\mathtt{Synch}^{\gamma}(v) \wedge (\mathtt{PhComplete}^{\gamma}(v) \vee \mathtt{NextPlay}^{\gamma}(v))$.

Then $\forall u \in \mathtt{Players}^{\gamma}(v), \neg\mathtt{LosePar}^{\gamma}(u,v) \wedge \neg\mathtt{AnswerPar}^{\gamma}(u,v)$

Proof: Since we are only interested in properties at γ, no confusion can be made, and thus we do not precise the configuration in which the predicates and sets are evaluated in that proof.

By definition of $\mathtt{Synch}(v)$, we have $\mathtt{ph}_u = \mathtt{ph}_v$. Consequently, we have:

$$\mathtt{LosePar}(u,v) \equiv \mathtt{b}_v \neq \mathtt{b}_u \wedge \mathtt{b}_v \neq \bot, \text{ and}$$

$$\mathtt{AnswerPar}(u,v) \equiv \mathtt{ph}_v = 1 \wedge \mathtt{b}_v = \bot \wedge \mathtt{b}_u = \bot.$$

Let us first suppose that $\mathtt{PhComplete}(v)$. Then by definition, $\mathtt{b}_u = \mathtt{b}_v$, and thus $\neg\mathtt{LosePar}(u,v)$. Furthermore, we also have $\mathtt{ph}_v \neq 1 \vee \mathtt{b}_v \neq \bot$, which can be rewritten $\neg(\mathtt{ph}_v = 1 \wedge \mathtt{b}_v = \bot)$ and thus $\neg\mathtt{AnswerPar}(u,v)$.

Let us now suppose that $\mathtt{NextPlay}(v)$. We deduce that $\mathtt{b}_v = \bot$, and thus $\neg\mathtt{LosePar}(u,v)$. Furthermore, we have $\mathtt{AnswerPar}(u,v)$ only if $\mathtt{ph}_v = 1 \wedge \mathtt{b}_u = \bot$. But if $\mathtt{ph}_v = 1$ then $\mathtt{NextPlay}(v)$ requires that $\mathtt{b}_u \neq \bot$, which is contradictory. Thus, $\neg\mathtt{AnswerPar}(u,v)$. ■

Lemmas 5.21 and 5.22 prove that the definition of predicate $\texttt{Losers}$ is adapted to the algorithm. We prove that all the nodes that execute $\mathbb{R}_{\texttt{Lose}}$ were actually counted in the set $\texttt{Losers}$, and that execution of $\mathbb{R}_{\texttt{NewBit}}$ does not bring any new node in the set $\texttt{Losers}$.

Lemma 5.21

Let $cs = \gamma \to \gamma'$ be a computing step, and let $v \in V$ be a node such that $\texttt{tok}_v^\gamma = \bullet$. $\forall u \in \texttt{Players}^\gamma(v)$, if u executes $\mathbb{R}_{\texttt{Lose}}$ during cs, then $u \in \texttt{Losers}^\gamma(v)$

Proof: Let u be one such node. Since u executes $\mathbb{R}_{\texttt{Lose}}$ during cs, $\texttt{OkNeg}^\gamma(u) = \texttt{true}$ which means that v is the only parent of u such that $\texttt{tok}_v = \bullet$.

Thus, we have $\texttt{ph}_v^\gamma = \texttt{ph}_u^\gamma \wedge \texttt{b}_v^\gamma \neq \texttt{b}_u^\gamma \wedge \texttt{b}_v^\gamma \neq \bot$. Then $u \in \texttt{Losers}_{\neg\bot}^\gamma(v) = \texttt{Losers}^\gamma(v)$. ■

Lemma 5.22

Let $cs = \gamma \to \gamma'$ be a computing step, and let $v \in V$ be a node such that $\texttt{tok}_v^\gamma = \bullet$ and such that v is not activated during cs.

Let $u \in \texttt{Players}^\gamma(v)$ be a node that executes $\mathbb{R}_{\texttt{NewBit}}$ during cs. If $u \in \texttt{Losers}^{\gamma'}(v)$ then $u \in \texttt{Losers}^\gamma(v)$.

Proof: Since u executes $\mathbb{R}_{\texttt{NewBit}}$ during cs, $\texttt{OkNeg}^\gamma(v) = \texttt{true}$ so v is the only parent of u such that $\texttt{tok}_p v^\gamma = \bullet$. Thus, the execution of $\texttt{Announce}$ produces $\texttt{ph}_u^{\gamma'} = \texttt{ph}_v^\gamma = \texttt{ph}_v^{\gamma'}$ (because v is not activated during cs). Since $u \in \texttt{Losers}^{\gamma'}(v)$, we necessarily have $\texttt{b}_v^{\gamma'} \neq \bot$ and $\texttt{b}_u^{\gamma'} \neq \texttt{b}_v^{\gamma'}$.

Since v is not activated during cs, this means that $\texttt{b}_v^\gamma \neq \bot$ and that $\texttt{b}_u^{\gamma'} \neq \texttt{b}_v^\gamma$. But by definition of $\texttt{Announce}$, we have $\texttt{b}_u^{\gamma'} = \texttt{Bit}_u(\texttt{ph}_v^\gamma)$, and then we conclude that $\texttt{Bit}_u(\texttt{ph}_v^\gamma) \neq \texttt{b}_v^\gamma \wedge \texttt{ph}_u^\gamma \neq \texttt{ph}_v^\gamma \wedge \texttt{b}_v^\gamma \neq \bot$ which means that $u \in \texttt{Losers}^\gamma(v)$. ■

Lemma 5.23 proves that the definition of $\texttt{FullRound}$ is adapted to our algorithm. Indeed, we establish that one node v such that $\texttt{tok}_v = \bullet$ both before and after a computing step does not see its weight by $\texttt{W}_{\texttt{Circ}}$ increase, unless it makes decrease one higher component of $\texttt{W}$ at the same time.

Lemma 5.23

Let $cs = \gamma \to \gamma'$ be a computing step, and let $v \in V$ be a node such that $\texttt{tok}_v^\gamma = \texttt{tok}_v^{\gamma'} = \bullet$. Suppose that $\neg\texttt{FullRound}^\gamma(v)$. Then $\neg\texttt{FullRound}^{\gamma'}(v)$ or there exists one child u of v that executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, $\mathbb{R}_{\texttt{ReplayUp}}$, or $\mathbb{E}_{\texttt{TrustChild}}$ during cs.

Proof: Let us first remark that by hypothesis, v does not execute $\mathbb{E}_{\texttt{TrustChild}}$, $\mathbb{R}_{\texttt{Give}}$, or $\mathbb{R}_{\texttt{OfferUp}}$, since it would update $\texttt{tok}_v$.

Suppose first that $\texttt{Losers}^\gamma(v) \neq \texttt{Players}^\gamma(v)$, and let us consider the different cases depending on which rule v executes during cs.

- If v is not activated during cs, then let us consider one node $u \in \texttt{Players}^\gamma(v) \setminus \texttt{Losers}^\gamma(v)$. If u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ or $\mathbb{E}_{\texttt{TrustChild}}$, then we obtain the theorem. Lemma 5.21 states that u does not execute $\mathbb{R}_{\texttt{Lose}}$ during cs. Lemma 5.22 states that if u execute $\mathbb{R}_{\texttt{NewBit}}$ during cs, then $u \in \texttt{Players}^{\gamma'}(v) \setminus \texttt{Losers}^{\gamma'}(v)$, and the same happens if u is not activated during cs. Thus, either u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ or $\mathbb{E}_{\texttt{TrustChild}}$, either $\texttt{Losers}^{\gamma'}(v) \neq \texttt{Players}^{\gamma'}(v)$, and then $\neg\texttt{FullRound}^{\gamma'}(v)$.
- If v executes $\mathbb{R}_{\texttt{maxPos}}$ or $\mathbb{R}_{\texttt{NewPh}}$ during cs then according to Lemma 5.20, nodes $u \in \texttt{Players}^\gamma(v)$ do not execute $\mathbb{R}_{\texttt{Lose}}$ nor $\mathbb{R}_{\texttt{NewBit}}$. If one such node executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ or $\mathbb{E}_{\texttt{TrustChild}}$, then we obtain the theorem. Thus, we can now suppose that $\forall u \in \texttt{Players}^\gamma(v)$, u is not activated during cs. As a corollary, we have $\texttt{Players}^{\gamma'}(v) = \texttt{Players}^\gamma(v)$.

- If v executes $\mathbb{R}_{\texttt{maxPos}}$ during cs then $\texttt{MaxPos}(v)$ updates $\texttt{b}_v$ to the maximal value of $\texttt{b}_u$ where $u \in \texttt{Players}^{\gamma}(v)$. Let u be one node with maximal value of $\texttt{b}$ among $\texttt{Players}^{\gamma}(v)$. Since $u \in \texttt{Players}^{\gamma'}(v)$ and $\texttt{ph}_u^{\gamma'} = \texttt{ph}_v^{\gamma'} \wedge \texttt{b}_u^{\gamma'} = \texttt{b}_v^{\gamma'}$, we have $u \in \texttt{Players}^{\gamma'}(v) \setminus \texttt{Losers}^{\gamma'}(v)$ and thus $\neg\texttt{FullRound}^{\gamma'}(v)$.
- If v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs then $\texttt{PhasePlus}(v)$ updates $\texttt{b}_v$ to $\perp$, which means that $\texttt{Losers}^{\gamma'}(v) = \emptyset$. Since v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs then $\texttt{Players}^{\gamma}(v) \neq \emptyset$, and by what precedes, $\texttt{Players}^{\gamma'}(v) \neq \emptyset$, which implies that $\texttt{Players}^{\gamma'}(v) \neq \texttt{Losers}^{\gamma'}(v)$ and thus $\neg\texttt{FullRound}^{\gamma'}(v)$.

Let us now suppose that $\texttt{Losers}^{\gamma}(v) = \texttt{Players}^{\gamma}(v)$, which implies $\texttt{Candidates}^{\gamma}(v) = \emptyset$. Since $\texttt{Players}^{\gamma}(v) \subseteq \texttt{Candidates}^{\gamma}(v)$, we have $\texttt{Players}^{\gamma}(v) = \emptyset$, so v cannot execute $\mathbb{R}_{\texttt{maxPos}}$ or $\mathbb{R}_{\texttt{NewPh}}$. Thus, v is not activated during cs (recall that by hypothesis, $\texttt{tok}_v^{\gamma'} = \bullet$). Let us consider one node $u \in \mathcal{C}(v)$. Since $\texttt{Candidates}^{\gamma}(v) = \emptyset$, either $\texttt{c}_u^{\gamma} = \texttt{c}_v^{\gamma}$, either $\texttt{tok}_u^{\gamma} \neq \perp$, either $\texttt{play}_u^{\gamma} \in \{\texttt{W}, \texttt{F}\}$.

- If $\texttt{c}_u^{\gamma} = \texttt{c}_v^{\gamma}$, then either u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, and then the theorem is established, either not, and then $\texttt{c}_u^{\gamma'} = \texttt{c}_v^{\gamma'}$ so $u \notin \texttt{Candidates}^{\gamma'}(v)$.
- If $\texttt{c}_u^{\gamma} \neq \texttt{c}_v^{\gamma} \wedge \texttt{tok}_u^{\gamma} \neq \perp$, then u does not execute $\mathbb{R}_{\texttt{Return}}$ during cs since v is a parent of u and $\texttt{tok}_v^{\gamma} \notin \{\perp, \downarrow, \uparrow, \circ\}$. Thus, $\texttt{tok}_u^{\gamma'} \neq \perp$, and so $u \notin \texttt{Candidates}^{\gamma'}(v)$.
- If $\texttt{c}_u^{\gamma} \neq \texttt{c}_v^{\gamma} \wedge \texttt{tok}_u^{\gamma} = \perp \wedge \texttt{play}_u^{\gamma} \in \{\texttt{W}, \texttt{F}\}$ then either u executes $\mathbb{R}_{\texttt{ReplayUp}}$ and then the theorem is established, either it does not and then $\texttt{play}_u^{\gamma'} \in \{\texttt{W}, \texttt{F}\}$ so $u \notin \texttt{Candidates}^{\gamma'}(v)$.

Thus, either one child u of v executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, or $\mathbb{R}_{\texttt{ReplayUp}}$, either not but then $\forall u \in \mathcal{C}(v), u \notin \texttt{Candidates}^{\gamma'}(v)$ so $\texttt{Candidates}^{\gamma'}(v) = \emptyset$ and thus $\neg\texttt{FullRound}^{\gamma'}(v)$. ■

Lemma 5.24 establishes under which conditions we can have an increase of $\texttt{W}_{\texttt{Circ}}$ on one node v. It proves that if this happens, then we fall into the conditions of Theorem 5.4 or of Theorem 5.6. This lemma, which largely uses Lemma 5.23, is the main argument to establish that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})$ is pre-admissible.

Lemma 5.24

Let $cs = \gamma \to \gamma'$ be a computing step, let a be an anchor at γ', and let $v \in D_a^{\gamma}$ be a node.

If $\texttt{W}_{\texttt{Circ}}(v, \gamma') > \texttt{W}_{\texttt{Circ}}(v, \gamma)$ then $\texttt{W}_{\texttt{Ch}}(v, \gamma') < \texttt{W}_{\texttt{Ch}}(v, \gamma)$, or v executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{E}_{\texttt{TrustChild}}$, or $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs, or $\texttt{tok}_v^{\gamma} \notin \{\perp, \uparrow, \downarrow\}$ and one child u of v executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, $\mathbb{R}_{\texttt{ReplayUp}}$ or $\mathbb{E}_{\texttt{TrustChild}}$ during cs.

Proof: Let us consider different cases depending on the value of $\texttt{tok}_v^{\gamma}$:

- If $\texttt{tok}_v^{\gamma} = \perp$, then $\texttt{W}_{\texttt{Circ}}(v, \gamma') > \texttt{W}_{\texttt{Circ}}(v, \gamma)$ if and only if v executes $\mathbb{R}_{\texttt{Win}}$ during cs.
- If $\texttt{tok}_v^{\gamma} = \bigstar$, then $\texttt{W}_{\texttt{Circ}}(v, \gamma') \leq \texttt{W}_{\texttt{Circ}}(v, \gamma)$.
- If $\texttt{tok}_v^{\gamma} = \bullet \wedge \texttt{FullRound}^{\gamma}(v)$ then $\texttt{W}_{\texttt{Circ}}(v, \gamma') \leq \texttt{W}_{\texttt{Circ}}(v, \gamma)$.
- If $\texttt{tok}_v^{\gamma} = \bullet \wedge \neg\texttt{FullRound}^{\gamma}(v) \wedge \texttt{tok}_v^{\gamma'} = \bullet$ then according to Lemma 5.23 either $\neg\texttt{FullRound}^{\gamma'}(v)$ and thus $\texttt{W}_{\texttt{Circ}}(v, \gamma') = \texttt{W}_{\texttt{Circ}}(v, \gamma)$, either there exists one child u of v that executes $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, $\mathbb{R}_{\texttt{ReplayUp}}$, or $\mathbb{E}_{\texttt{TrustChild}}$ during cs.
- If $\texttt{tok}_v^{\gamma} = \bullet \wedge \neg\texttt{FullRound}^{\gamma}(v) \wedge \texttt{tok}_v^{\gamma'} \neq \bullet$ then either v executes $\mathbb{E}_{\texttt{TrustChild}}$, either v executes $\mathbb{R}_{\texttt{OfferUp}}$ and then $\texttt{W}_{\texttt{Circ}}(v, \gamma') = 1 < \texttt{W}_{\texttt{Circ}}(v, \gamma)$, either v executes $\mathbb{R}_{\texttt{Give}}$. Let us consider that last case.

  Remark first that if $\texttt{Candidates}^{\gamma}(v) = \emptyset$ then $\neg\texttt{Give}^{\gamma}(v)$, and thus v cannot execute $\mathbb{R}_{\texttt{Give}}$ during cs. As a consequence, we have $\texttt{Candidates}^{\gamma}(v) \neq \emptyset$. Then, since $\neg\texttt{FullRound}^{\gamma}(v)$ we must have $\texttt{Losers}^{\gamma}(v) \neq \texttt{Players}^{\gamma}(v)$, which is possible only if $\texttt{Players}^{\gamma}(v) \neq \emptyset$ (recall that $\texttt{Losers}^{\gamma}(v) \subseteq \texttt{Players}^{\gamma}(v)$).

Thus, we necessarily have $\texttt{Players}^{\gamma}(v) \neq \emptyset$, and actually since v executes $\mathbb{R}_{\texttt{Give}}$ during cs, there $\left|\texttt{Players}^{\gamma}(v)\right| = 1$. Let u be that node in $\texttt{Players}^{\gamma}(v)$. If u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, we have our result, let us now consider that it does not. Since $\texttt{Losers}^{\gamma}(v) \neq \texttt{Players}^{\gamma}(v)$, $u \notin \texttt{Losers}^{\gamma}(v)$ and thus, according to Lemma 5.21, u does not execute $\mathbb{R}_{\texttt{Lose}}$ during cs. As a consequence, u either executes $\mathbb{R}_{\texttt{NewBit}}$, either is not activated during cs. In both cases, variables $\texttt{tok}$, $\texttt{c}$, and $\texttt{play}$ are not updated on u during cs, and thus $u \in \texttt{Players}^{\gamma'}(v)$, so $\texttt{W}_{\texttt{Circ}}(v,\gamma') = 1 < \texttt{W}_{\texttt{Circ}}(v,\gamma)$.

- If $\texttt{tok}_v^{\gamma} = \downdownarrows \wedge \texttt{Players}^{\gamma}(v) = \emptyset$ then $\texttt{W}_{\texttt{Circ}}(v,\gamma') \leq \texttt{W}_{\texttt{Circ}}(v,\gamma)$.
- If $\texttt{tok}_v^{\gamma} = \downdownarrows \wedge \texttt{Players}^{\gamma}(v) \neq \emptyset$ then let us consider $u \in \texttt{Players}^{\gamma}(v)$. By definition, $u \in \mathcal{C}_{\texttt{neq}}{}^{\gamma}(v) \wedge (\texttt{play}_u^{\gamma} = \texttt{P} \wedge \texttt{tok}_u^{\gamma} \neq \bigstar)$. As a consequence, v does not execute $\mathbb{R}_{\texttt{NewPlay}}$ during cs. If v executes $\mathbb{E}_{\texttt{TrustChild}}$ during cs, then the desired property holds.
  Let us now suppose that v does not execute any rule during cs. Since $\texttt{tok}_v^{\gamma} = \downdownarrows$, u cannot execute $\mathbb{R}_{\texttt{Lose}}$ or $\mathbb{R}_{\texttt{NewBit}}$ during cs, since it requires that one other parent p of u is such that $\texttt{tok}_p^{\gamma} = \bullet$, and then $\neg\texttt{OkNeg}^{\gamma}(u)$. Thus, either u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs and then our property holds, either it does not, but then it is not activated and thus $u \in \texttt{Players}^{\gamma'}(v)$ which means that $\texttt{W}_{\texttt{Circ}}^{\gamma'} = 1 = \texttt{W}_{\texttt{Circ}}(v,\gamma)$.
- If $\texttt{tok}_v^{\gamma} = \circlearrowleft$, then $\texttt{tok}_v^{\gamma'} \in \{\circlearrowleft, \downarrow\}$ and thus $\texttt{W}_{\texttt{Circ}}(v,\gamma') \leq \texttt{W}_{\texttt{Circ}}(v,\gamma)$.
- If $\texttt{tok}_v^{\gamma} = \downarrow$ then $\texttt{tok}_v^{\gamma'} \in \{\downarrow, \circ\}$ and thus $\texttt{W}_{\texttt{Circ}}(v,\gamma') \leq \texttt{W}_{\texttt{Circ}}(v,\gamma)$.
- If $\texttt{tok}_v^{\gamma} = \circ$, then $\texttt{W}_{\texttt{Circ}}(v,\gamma') > \texttt{W}_{\texttt{Circ}}(v,\gamma)$ only if v executes $\mathbb{E}_{\texttt{TrustChild}}$ or $\mathbb{R}_{\texttt{ReNego}}$ during cs. If v executes $\mathbb{E}_{\texttt{TrustChild}}$, then our property holds, let us now suppose that v executes $\mathbb{R}_{\texttt{ReNego}}$ during cs.
  We have $\forall u \in \mathcal{C}_{\texttt{neq}}{}^{\gamma}(v), \texttt{play}_u^{\gamma} \neq \texttt{L}$. Let us first prove that $\forall u \in \mathcal{C}_{\texttt{neq}}{}^{\gamma'}(v), \texttt{play}_u^{\gamma'} \neq \texttt{L}$, and let us consider $u \in \mathcal{C}^{\gamma}(v)$. Since v executes $\mathbb{R}_{\texttt{ReNego}}$ during cs, we have $\texttt{tok}_u^{\gamma} = \bot$. If $\texttt{c}_u^{\gamma} = \texttt{c}_v^{\gamma}$ then u switches its color only if it executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, and thus the property holds. Otherwise if $\texttt{c}_u^{\gamma} = \texttt{c}_v^{\gamma}$ then $\texttt{play}_u^{\gamma} \neq \texttt{L}$ and since $\texttt{tok}_v^{\gamma} = \circ$ u cannot execute $\mathbb{R}_{\texttt{Lose}}$ during cs, and thus $\texttt{play}_u^{\gamma'} \neq \texttt{L}$.
  Since $\texttt{b}_v^{\gamma'} = \bot$, we have $\texttt{Losers}^{\gamma'}(v) = \emptyset$. Thus, either $\exists u \in \texttt{Players}^{\gamma'}(v)$ and thus $\texttt{Losers}^{\gamma'} \neq \texttt{Players}^{\gamma'}(v)$ so $\neg\texttt{FullRound}^{\gamma'}(v)$, either $\texttt{Players}^{\gamma'}(v) = \emptyset$ but then according to what we established just above, we have $\texttt{Candidates}^{\gamma'}(v) = \emptyset$ and once again $\neg\texttt{FullRound}^{\gamma'}(v)$. In both cases, $\texttt{W}_{\texttt{Circ}}^{\gamma'}(v) = 2 < \texttt{W}_{\texttt{Circ}}^{\gamma}(v)$.
- If $\texttt{tok}_v^{\gamma} = \uparrow$ then either $v \neq r$ and thus $\texttt{tok}_v^{\gamma} \in \{\uparrow, \bot\}$ so $\texttt{W}_{\texttt{Circ}}(v,\gamma') \leq \texttt{W}_{\texttt{Circ}}(v,\gamma)$, either $v = r$ and thus, if activated, v executes $\mathbb{R}_{\texttt{NewDFS}}^{r}$ and the property holds. ■

Theorem 5.7 establishes that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})$ is pre-admissible.

Theorem 5.7

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}_{\texttt{NewDFS}}^{r}$ during cs.
We have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D_a^{\gamma'}, \gamma') \preceq^3 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D_a^{\gamma}, \gamma)$.

Proof: According to Theorem 5.5, we already have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \preceq (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_t^{\gamma}, \gamma)$. Consequently, we only have to prove that if $\texttt{W}_{\texttt{Circ}}(D_a^{\gamma'}, \gamma') > \texttt{W}_{\texttt{Circ}}(D_a^{\gamma}, \gamma)$, then we have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma}, \gamma)$.

In the same way as we did in the proof of Theorem 5.5, we can suppose that for any branch $\mathcal{B}$ of G_a, $D_a^{\gamma'}(\mathcal{B})$ is not longer than $D_a^{\gamma}(\mathcal{B})$. In such circumstances, we have $\texttt{W}_{\texttt{Circ}}(D_a^{\gamma'}, \gamma') > \texttt{W}_{\texttt{Circ}}(D_a^{\gamma}, \gamma)$ only if there exists a branch $\mathcal{B}$ of G_a such that $\exists v \in D_a^{\gamma}(\mathcal{B}) : \texttt{W}_{\texttt{Circ}}(v,\gamma') > \texttt{W}_{\texttt{Circ}}(v,\gamma)$. Let us now apply Lemma 5.24. If v executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{E}_{\texttt{TrustChild}}$ (it cannot execute $\mathbb{R}_{\texttt{NewDFS}}^{r}$ by hypothesis), then Theorem 5.3 or Theorem 5.5 guarantees that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma}, \gamma)$.

Otherwise, $\texttt{tok}_v^{\gamma} \notin \{\bot, \uparrow, \downarrow\}$ and $\exists u \in \mathcal{C}(v)$ that executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ or $\mathbb{E}_{\texttt{TrustChild}}$, and once again Theorem 5.3 or Theorem 5.5 guarantees that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma'}, \gamma') \prec^2 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}})(D_a^{\gamma}, \gamma)$.

■

Theorem 5.8 proves that under certain circumstances, we are certain that the weight of one $\texttt{CD}^o$ decreases. It will be useful to prove that $\texttt{W}$ is an admissible function, but also to prove that the other, longer, prefixes of $\texttt{W}$ are pre-admissible. Indeed, if in any computing step one node updates its variable $\texttt{tok}$ then we do not need to look beyond the third component to be assured that the weight does not increase, since $\preceq^k$ is a lexicographic order.

Theorem 5.8

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs.

If $\exists v \in D^\gamma_a$ such that v executes one rule among $\mathbb{R}_{\texttt{Give}}$, $\mathbb{R}_{\texttt{NewPlay}}$, $\mathbb{R}_{\texttt{Drop}}$, $\mathbb{R}_{\texttt{Nego}}$, $\mathbb{R}_{\texttt{OfferUp}}$, $\mathbb{R}_{\texttt{Receive}}$, $\mathbb{R}_{\texttt{Return}}$, $\mathbb{R}_{\texttt{ReNego}}$ during cs, then $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^{\gamma'}_a, \gamma') \prec^3 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^\gamma_a, \gamma)$.

Proof: Let us consider one such node $v \in D^\gamma_a$, and one branch $\mathcal{B}$ of G_a such that $v \in D^\gamma_a(\mathcal{B})$. If $v \notin D^{\gamma'}_a(\mathcal{B})$ then according to Lemma 5.18 we have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^{\gamma'}_a, \gamma') \prec^3 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^\gamma_a, \gamma)$.

Else, since any of these rules guarantee that $\texttt{W}_{\texttt{Circ}}(v, \gamma') \neq \texttt{W}_{\texttt{Circ}}(v, \gamma)$ and since we established in Theorem 5.7 that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^{\gamma'}_a, \gamma') \preceq^3 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^\gamma_a, \gamma)$, we have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^{\gamma'}_a, \gamma') \prec^3 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})(D^\gamma_a, \gamma)$. ■

5.4.2.7 Fourth component of W: $\texttt{W}_{\texttt{Play}}$, circulation of variable play on children

Explanations The three weight functions that we previously defined, $\texttt{W}_{\texttt{Ch}}$, $\texttt{W}_{\texttt{Er}}$, and $\texttt{W}_{\texttt{Circ}}$ totally embrace the rules that affect $\texttt{tok}_v$ and $\texttt{c}_v$. Thus, we can now consider that both these variable are constant on nodes, and focus on the two other variables that are $\texttt{play}$ and $\texttt{id}$. In this section, we only treat variable $\texttt{play}$. Remark that the rules $\mathbb{R}_{\texttt{Win}}$, $\mathbb{R}_{\texttt{FakeWin}}$, and $\mathbb{R}_{\texttt{Return}}$ have an impact on variable $\texttt{play}$, but were already treated in Sections 5.4.2.4 and 5.4.2.6. Thus, although we discussed rule $\mathbb{R}_{\texttt{ReplayUp}}$ in Section 5.4.2.4, we did not treat all the cases where this rule might be executed, and thus we must consider it in this section. Consequently, the rules that interest us in this section are $\mathbb{R}_{\texttt{Lose}}$, $\mathbb{R}_{\texttt{ReplayD}}$, $\mathbb{R}_{\texttt{ReplayUp}}$, $\mathbb{R}_{\texttt{Fake}}$, and $\mathbb{R}_{\texttt{ReWin}}$.

Depending on its variable $\texttt{tok}_v$, it happens that one node v is waiting for some of its children to update their variable $\texttt{play}$, before it executes a rule that updates $\texttt{tok}_v$ to the next state. For example, if $\texttt{tok}_v = \downdownarrows$, then v does not execute $\mathbb{R}_{\texttt{NewPlay}}$ until all of its children u such that $\texttt{tok}_u = \texttt{P}$ updates $\texttt{play}_u$ to $\texttt{L}$, $\texttt{W}$, or $\texttt{F}$. In order to prove that our algorithm progresses, and that circulation might terminate, we need to prove that the number of children u of v such that $\texttt{play}_u = \texttt{P}$ never increases, at least as long as $\texttt{tok}_v = \downdownarrows$. Indeed, as soon as $\texttt{tok}_v$ is updated, then we already know that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}})$ decreases. Actually, when $\texttt{tok}_v = \downdownarrows$, children u of v such that $\texttt{play}_u = \texttt{P}$ will update $\texttt{play}_u$ with rule $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, both cases have already been treated in Section 5.4.2.4, so we do not need to elaborate any further on this case.

There exists other situations where one node v waits until its children u have some specific values for their variable $\texttt{play}_u$ before v itself updates $\texttt{tok}_v$. For each of those situations, we want to define $\texttt{W}_{\texttt{Play}}$ such that any time one child u of v updates its variable $\texttt{play}_u$ in a sense that allows v to update $\texttt{tok}_v$, $\texttt{W}_{\texttt{Play}}$ decreases on v. Thus, we define $\texttt{W}_{\texttt{Play}}$ such that each node v counts the number of children it has that prevent it from updating its variable $\texttt{tok}_v$. In order to have a function that is pre-admissible, we also require that $\texttt{W}_{\texttt{Play}}$ never increases, and thus that no action taken by one node u might add u to the set of children that prevent one of its parents v to update $\texttt{tok}_v$.

The situations where one node v waits for its children u to update $\texttt{play}_u$ occurs when $\texttt{tok}_v \in \{\bullet, \circlearrowleft, \circ, \uparrow\}$. Let us give some details:

- If $\texttt{tok}_v = \bullet$ and $\texttt{b}_v \neq \bot$, then v waits for some of its children u to execute $\mathbb{R}_{\texttt{Lose}}$ and update $\texttt{play}_u$ from P to L.

  Although rule $\mathbb{R}_{\texttt{Lose}}$ could be treated in the following section that focuses on negotiation, we treat it here for two reasons. First, it concerns variable play as well, and thus it remains consistent here. Second, having already treated executions of rule $\mathbb{R}_{\texttt{Lose}}$ will help us greatly in the section on negotiation.

- If $\texttt{tok}_v = \circ$ or $\texttt{tok}_v = \circlearrowleft$, then v does not execute $\mathbb{R}_{\texttt{Drop}}$ until all of its children u such that $\texttt{tok}_u = \texttt{L}$ execute $\mathbb{R}_{\texttt{ReplayD}}$ and update $\texttt{play}_u$ to P.

- If $\texttt{tok}_v = \uparrow$, then before executing $\mathbb{R}_{\texttt{Return}}$, v waits for all of its children u such that $\texttt{play}_u = \texttt{W}$ execute $\mathbb{R}_{\texttt{ReplayUp}}$ or $\mathbb{R}_{\texttt{Fake}}$, and update $\texttt{play}_u$ from W to P or F, depending on whether u still has parents in any $\texttt{CD}^o$.

From what precedes, if one node u executes $\mathbb{R}_{\texttt{Lose}}$, $\mathbb{R}_{\texttt{ReplayD}}$, $\mathbb{R}_{\texttt{ReplayUp}}$, or $\mathbb{R}_{\texttt{Fake}}$, it will make decrease the weight of the $\texttt{CD}^o$'s that contain the parents v of u that have the corresponding value of $\texttt{tok}_v$. Indeed, all of these four rules are conditioned by the existence of at least one parent v with the adapted value of $\texttt{tok}_v$.

Remark that we did not mentioned rule $\mathbb{R}_{\texttt{ReWin}}$. It is not difficult to define a weight function that decreases on v each time one child u of v execute $\mathbb{R}_{\texttt{ReWin}}$. The difficulty comes from proving that the weight function we defined that way is pre-admissible. Indeed, $\mathbb{R}_{\texttt{ReWin}}$ might be executed in very various situations: as soon as u has one no parent v such that $\texttt{tok}_v = \uparrow$. Unfortunately, there exists situations in which we can give no guarantee that there won't be executions of $\mathbb{R}_{\texttt{Fake}}$ that cancels executions of $\mathbb{R}_{\texttt{ReWin}}$, and thus we cannot treat both $\mathbb{R}_{\texttt{Fake}}$ and $\mathbb{R}_{\texttt{ReWin}}$ at the same layer of W.

However, the design of the algorithm allows us design $\texttt{W}_{\texttt{Play}}$ that is pre-admissible and such that it decreases any time one node u executes $\mathbb{R}_{\texttt{Lose}}$, $\mathbb{R}_{\texttt{ReplayD}}$, $\mathbb{R}_{\texttt{ReplayUp}}$, or $\mathbb{R}_{\texttt{Fake}}$. This is stated in Theorems 5.9 and 5.10.

Figure 5.28 summarizes the different transitions that might be taken by variable $\texttt{play}_u$ on one node u.

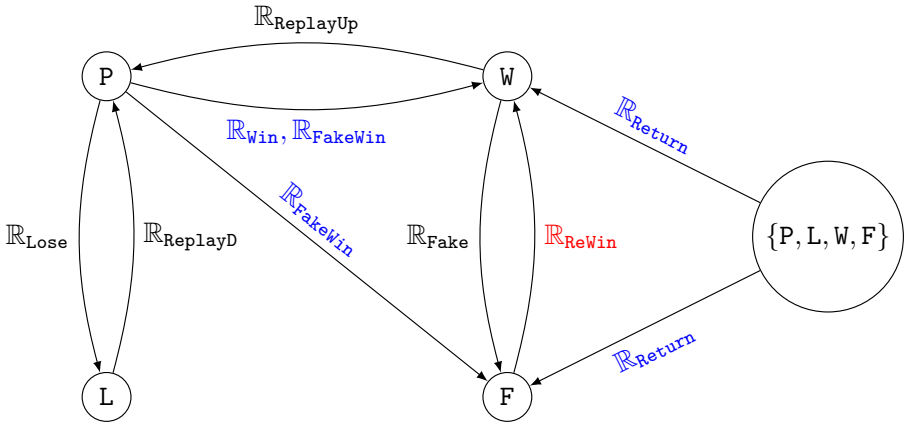

Figure 5.28: Diagram of variable $\texttt{play}_u$.

We represented in blue the rules that have already been treated in the previous sections, and in red the rule $\mathbb{R}_{\texttt{ReWin}}$ that is not treated in that section.

Definitions Let us first define the different sets that contain the children u of v that we are going to count in the different cases described above.

If $\texttt{tok}_v^\gamma = \bullet$, then we want to count children $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v)$ that might execute $\mathbb{R}_{\texttt{Lose}}$. One node u can execute $\mathbb{R}_{\texttt{Lose}}$ only if $\texttt{play}_u^\gamma = \texttt{P}$, so nodes $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) : \texttt{play}_u^\gamma = \texttt{P}$ must obviously be counted. Yet, we cannot count only those children of v. Indeed, it might happen that one node $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) : \texttt{play}_u^\gamma = \texttt{W}$ execute $\mathbb{R}_{\texttt{ReplayUp}}$, which produces $\texttt{play}_u^{\gamma'} = \texttt{P}$, and would make $\texttt{W}_{\texttt{Play}}$ increase on v. Although it would at the same time make $\texttt{W}_{\texttt{Play}}$ decrease on one other node w such that $\texttt{play}_w = \uparrow$, since we require that W decrease on each branch of each $\texttt{CD}^o$, we must prevent this from happening. Thus, we must include nodes $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) : \texttt{play}_u^\gamma = \texttt{W}$ in our count for $\texttt{tok}_v = \bullet$. For the same reason, due to rule $\mathbb{R}_{\texttt{ReWin}}$, we must also include nodes $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) : \texttt{play}_u^\gamma = \texttt{F}$ in that set. This leads to the following predicate:

$$\texttt{ChLose}^\gamma(v) = \{u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) \mid \texttt{tok}_u^\gamma = \bot \wedge \texttt{play}_u^\gamma \neq \texttt{L}\} \tag{5.46}$$

If $\texttt{tok}_v^\gamma = \circlearrowleft$ or $\texttt{tok}_v^\gamma = \circ$, then we want to count children $u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v)$ that might execute $\mathbb{R}_{\texttt{ReplayD}}$. One node u can execute $\mathbb{R}_{\texttt{ReplayD}}$ only if $\texttt{play}_u^\gamma = \texttt{L}$. Furthermore, due to predicate OkNeg, no node u that has a parent v with $\texttt{tok}_v^\gamma \in \{\circlearrowleft, \circ\}$ can execute $\mathbb{R}_{\texttt{Lose}}$. This leads to the following predicate:

$$\texttt{ChReplay}^\gamma(v) = \{u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v) \mid \texttt{tok}_u^\gamma = \bot \wedge \texttt{play}_u^\gamma = \texttt{L}\} \tag{5.47}$$

If $\texttt{tok}_v^\gamma = \uparrow$ then we want to count children $u \in {\mathcal{C}_{\texttt{eq}}}^\gamma(v)$ that might execute $\mathbb{R}_{\texttt{ReplayUp}}$ or $\mathbb{R}_{\texttt{Fake}}$. One node might execute $\mathbb{R}_{\texttt{ReplayUp}}$ or $\mathbb{R}_{\texttt{Fake}}$ only if $\texttt{play}_u^\gamma = \texttt{W}$. One node u might updates its variable $\texttt{play}_u$ from P to W only if it executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, cases that have been treated previously. Furthermore, due to predicate StopFaking, no node u that has a parent $v \in \mathcal{P}_{\texttt{eq}}(u)$ such that $\texttt{tok}_v = \uparrow$ can execute $\mathbb{R}_{\texttt{ReWin}}$, and thus update its variable $\texttt{play}_u$ from F to W. This leads to the following predicate:

$$\texttt{ChEnd}^\gamma(v) = \{u \in {\mathcal{C}_{\texttt{eq}}}^\gamma(v) \mid \texttt{tok}_u^\gamma = \bot \wedge \texttt{play}_u^\gamma = \texttt{W}\} \tag{5.48}$$

We can now define $\texttt{W}_{\texttt{Play}}(v, \gamma)$ depending on the value of $\texttt{tok}_v^\gamma$.

$$\texttt{W}_{\texttt{Play}}(v,\gamma) = \begin{cases} |\texttt{ChLose}^\gamma(v)| & \text{if } \texttt{tok}_v^\gamma = \bullet \\ |\texttt{ChReplay}^\gamma(v)| & \text{if } \texttt{tok}_v^\gamma \in \{\circ, \circlearrowleft\} \\ |\texttt{ChEnd}^\gamma(v)| & \text{if } \texttt{tok}_v^\gamma = \uparrow \\ 0 & \text{if } \texttt{tok}_v \in \{\bot, \downdownarrows, \downarrow, \bigstar\} \end{cases} \tag{5.49}$$

Proofs Lemmas 5.25, 5.26 and 5.27 prove that the definitions of ChLose, ChReplay, and ChEnd are consistent with our algorithm. For $\texttt{ChLose}(v)$ and $\texttt{ChReplay}(v)$ we prove in Lemmas 5.25 and 5.26 that both these sets do not increase unless it is by one node $u \in \mathcal{C}(v)$ that executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, which makes decrease one more important quantity. We cannot use the same reasoning for $\texttt{ChEnd}(v)$. Indeed, since we consider one node v such that $\texttt{tok}_v = \uparrow$, $\texttt{W}_{\texttt{Ch}}(v)$ is not affected by any action taken by one child $u \in \mathcal{C}(v)$. Thus, we prove in Lemmas 5.27 that $\texttt{ChEnd}(v)$ never increases, as long as $\texttt{tok}_v = \uparrow$.

We also establish in these lemmas some particular cases which necessarily decrease $\texttt{W}_{\texttt{Play}}$.

Lemma 5.25

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^\gamma = \bullet$ and $\texttt{tok}_v^{\gamma'} = \bullet$.

If $\forall u \in \mathcal{C}(v)$, u does not execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, then $\texttt{ChLose}^{\gamma'}(v) \subseteq \texttt{ChLose}^\gamma(v)$. Furthermore, if $\exists u \in {\mathcal{C}_{\texttt{neq}}}^\gamma(v)$ that executes $\mathbb{R}_{\texttt{Lose}}$ during cs, then $\texttt{W}_{\texttt{Play}}(v,\gamma') < \texttt{W}_{\texttt{Play}}(v,\gamma)$.

Proof: For the first part, we prove the equivalent statement, under the same conditions:

$$\mathcal{C}(v) \setminus \texttt{ChLose}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{ChLose}^{\gamma'}(v)$$

Let us consider $u \in \mathcal{C}(v) \setminus \texttt{ChLose}^{\gamma}(v)$. If $\texttt{tok}_u^{\gamma} \neq \bot$ then since $\texttt{tok}_v^{\gamma} = \bullet$, u does not execute $\mathbb{R}_{\texttt{Return}}$ and thus $\texttt{tok}_u^{\gamma'} \neq \bot$ so $u \notin \texttt{ChLose}^{\gamma'}(v)$. Otherwise, $\texttt{tok}_u^{\gamma} = \bot$. If $u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma}(v)$, then since u does not execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, $u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma'}(v)$ and thus $u \notin \texttt{ChLose}^{\gamma'}(v)$. Otherwise, $u \in {\mathcal{C}_{\texttt{neq}}}^{\gamma}(v)$, and thus $\texttt{play}_u^{\gamma} = \texttt{L}$. Since $v \in {\mathcal{P}_{\texttt{neq}}}^{\gamma}(v)$ and $\texttt{tok}_v = \bullet$, $\neg\texttt{ReplayL}^{\gamma}(u)$ so u does not execute $\mathbb{R}_{\texttt{ReplayD}}$ so $\texttt{play}_u^{\gamma'} = \texttt{L}$ and thus $u \notin \texttt{ChLose}^{\gamma'}(v)$.

Let us now prove the second part. By definition of $\mathbb{R}_{\texttt{Lose}}$ we have $\texttt{play}_u^{\gamma} = \texttt{P} \wedge \texttt{tok}_v^{\gamma} = \bot$, and thus, $u \in \texttt{ChLose}^{\gamma}(v)$. But since u executes $\mathbb{R}_{\texttt{Lose}}$ during cs, $\texttt{play}_u^{\gamma'} \neq \texttt{P}$ and thus $u \notin \texttt{ChLose}^{\gamma'}(v)$. Then, what precedes implies that $\texttt{ChLose}^{\gamma'}(v) \subsetneq \texttt{ChLose}^{\gamma}(v)$, which leads to $\texttt{W}_{\texttt{Play}}(v, \gamma') < \texttt{W}_{\texttt{Play}}(v, \gamma)$. ■

Lemma 5.26

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} \in \{\circlearrowleft, \circ\}$ and $\texttt{tok}_v^{\gamma'} \in \{\circlearrowleft, \circ\}$.

If $\forall u \in \mathcal{C}(v)$, u does not execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, then $\texttt{ChReplay}^{\gamma'}(v) \subseteq \texttt{ChReplay}^{\gamma}(v)$. Furthermore, if $\exists u \in {\mathcal{C}_{\texttt{neq}}}^{\gamma}(v)$ that executes $\mathbb{R}_{\texttt{ReplayD}}$ during cs, then $\texttt{W}_{\texttt{Play}}(v, \gamma') < \texttt{W}_{\texttt{Play}}(v, \gamma)$.

Proof: For the first part, we prove the equivalent statement, under the same conditions:

$$\mathcal{C}(v) \setminus \texttt{ChReplay}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{ChReplay}^{\gamma'}(v)$$

Let us consider $u \in \mathcal{C}(v) \setminus \texttt{ChReplay}^{\gamma}(v)$. If $\texttt{tok}_u^{\gamma} \neq \bot$ then since $\texttt{tok}_v^{\gamma} = \bullet$, u does not execute $\mathbb{R}_{\texttt{Return}}$ and thus $\texttt{tok}_u^{\gamma'} \neq \bot$ so $u \notin \texttt{ChReplay}^{\gamma'}(v)$. Otherwise, $\texttt{tok}_u^{\gamma} = \bot$. If $u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma}(v)$, then since u does not execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, then $u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma'}(v)$ and thus $u \notin \texttt{ChReplay}^{\gamma'}(v)$. Otherwise, $u \in {\mathcal{C}_{\texttt{neq}}}^{\gamma}(v)$, and thus $\texttt{play}_u^{\gamma} \in \{\texttt{P}, \texttt{W}, \texttt{F}\}$. If $\texttt{play}_u^{\gamma} \in \{\texttt{W}, \texttt{F}\}$ no rule can induce $\texttt{play}_u^{\gamma'} = \texttt{L}$. If $\texttt{play}_u^{\gamma} = \texttt{P}$, then since $v \in \texttt{ParNeg}^{\gamma}(u) \wedge \texttt{tok}_v^{\gamma} \neq \bullet$, then due to $\texttt{OkNeg}$ u cannot execute $\mathbb{R}_{\texttt{Lose}}$ during cs, and thus $u \notin \texttt{ChReplay}^{\gamma'}(v)$.

Let us now prove the second part. By definition of $\mathbb{R}_{\texttt{ReplayD}}$ we have $\texttt{play}_u^{\gamma} = \texttt{L} \wedge \texttt{tok}_v^{\gamma} = \bot$, and thus, $u \in \texttt{ChReplay}^{\gamma}(v)$. But since u executes $\mathbb{R}_{\texttt{ReplayD}}$ during cs, $\texttt{play}_u^{\gamma'} \neq \texttt{L}$ and thus $u \notin \texttt{ChReplay}^{\gamma'}(v)$. Then, what precedes implies that $\texttt{ChReplay}^{\gamma'}(v) \subsetneq \texttt{ChReplay}^{\gamma}(v)$, which leads to $\texttt{W}_{\texttt{Play}}(v, \gamma') < \texttt{W}_{\texttt{Play}}(v, \gamma)$. ■

Lemma 5.27

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} = \uparrow$ and $\texttt{tok}_v^{\gamma'} = \uparrow$.

We have $\texttt{ChEnd}^{\gamma'}(v) \subseteq \texttt{ChEnd}^{\gamma}(v)$. Furthermore, if $\exists u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma}(v)$ that executes $\mathbb{R}_{\texttt{Fake}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs, then $\texttt{W}_{\texttt{Play}}(v, \gamma') < \texttt{W}_{\texttt{Play}}(v, \gamma)$.

Proof: For the first part, we prove the equivalent statement:

$$\mathcal{C}(v) \setminus \texttt{ChEnd}^{\gamma}(v) \subseteq \mathcal{C}(v) \setminus \texttt{ChEnd}^{\gamma'}(v)$$

Let us consider $u \in \mathcal{C}(v) \setminus \texttt{ChEnd}^{\gamma}(v)$. If $u \in {\mathcal{C}_{\texttt{neq}}}^{\gamma}(v)$ we have $u \in \texttt{ChEnd}^{\gamma'}(v)$ only if u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ to switch its color. If u executes $\mathbb{R}_{\texttt{Win}}$ then $\texttt{tok}_u^{\gamma'} = \bigstar$ so $u \notin \texttt{ChEnd}^{\gamma'}(v)$. If u executes $\mathbb{R}_{\texttt{FakeWin}}$ then since $v \in {\mathcal{P}_{\texttt{neq}}}^{\gamma}(v) \wedge \texttt{tok}_v^{\gamma} = \uparrow$, the execution of $\texttt{FakeWin}(v)$ induces $\texttt{play}_u^{\gamma'} = \texttt{F}$, and then $u \notin \texttt{ChEnd}^{\gamma'}(v)$.

Suppose now $u \in {\mathcal{C}_{\texttt{eq}}}^{\gamma}(v)$. If $\texttt{tok}_u^{\gamma} \neq \bot$ then $u \in \texttt{ChEnd}^{\gamma'}(v)$ only if u executes $\mathbb{R}_{\texttt{Return}}$ during cs, but if it does since $v \in {\mathcal{P}_{\texttt{eq}}}^{\gamma}(v) \wedge \texttt{tok}_v^{\gamma} = \uparrow$, the execution of $\texttt{Drop}(v)$ induces $\texttt{play}_u^{\gamma'} = \texttt{F}$, and then $u \notin \texttt{ChEnd}^{\gamma'}(v)$.

Suppose now $\texttt{tok}_u^\gamma = \bot$. Then we have $\texttt{play}_u^\gamma \neq \texttt{W}$. Remark first that u cannot execute $\mathbb{R}_{\texttt{Return}}$ during cs since $\texttt{tok}_u^\gamma = \bot$. If $\texttt{play}_u^\gamma = \texttt{L}$ then we cannot have $\texttt{play}_u^{\gamma'} = \texttt{W}$, necessary condition to have $u \in \texttt{ChEnd}^{\gamma'}(v)$. If $\texttt{play}_u^\gamma = \texttt{P}$, then we have $\texttt{play}_u^{\gamma'} = \texttt{W}$ only if u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs, but if it does, then it switches its color$\texttt{c}_u$, so $u \in \mathcal{C}_{\texttt{neq}}{}^\gamma(v)$ and then $u \notin \texttt{ChEnd}^{\gamma'}(v)$. Finally, if $\texttt{play}_u^\gamma = \texttt{F}$ then u cannot execute $\mathbb{R}_{\texttt{ReWin}}$ since $v \in \mathcal{P}_{\texttt{eq}}{}^\gamma(v) \wedge \texttt{tok}_v^\gamma = \uparrow$, we have $\neg\texttt{FakeWin}^\gamma(u)$, and thus $u \notin \texttt{ChEnd}^{\gamma'}(v)$.

Let us now prove the second part. By definition of $\mathbb{R}_{\texttt{Fake}}$ and $\mathbb{R}_{\texttt{ReplayUp}}$ we have $\texttt{play}_u^\gamma = \texttt{W} \wedge \texttt{tok}_v^\gamma = \bot$, and thus, $u \in \texttt{ChEnd}^\gamma(v)$. But since u executes $\mathbb{R}_{\texttt{Fake}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs, $\texttt{play}_u^{\gamma'} \neq \texttt{W}$ and thus $u \notin \texttt{ChEnd}^{\gamma'}(v)$. Then, what precedes implies that $\texttt{ChEnd}^{\gamma'}(v) \subsetneq \texttt{ChEnd}^\gamma(v)$, which leads to $\texttt{W}_{\texttt{Play}}(v,\gamma') < \texttt{W}_{\texttt{Play}}(v,\gamma)$. ■

Lemma 5.28 combines the results of Lemmas 5.25, 5.26 and 5.27 and proves that $\texttt{W}_{\texttt{Play}}$ can decrease on one node only if the previous components of $\texttt{W}$ decrease on the $\texttt{CD}^o$'s that include that node.

Lemma 5.28

Let $cs = \gamma \to \gamma'$ be a computing step, let a be an anchor at γ' that does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs, and let $v \in D_a^\gamma$ be a node.

If $\texttt{W}_{\texttt{Play}}(v,\gamma') > \texttt{W}_{\texttt{Play}}(v,\gamma)$ then $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^{\gamma'},\gamma') \prec^3 (\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^\gamma,\gamma)$.

Proof: Let us first eliminate the cases in which the decrease of $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})$ is immediate. According to Theorem 5.4 we can assume that, if $\texttt{tok}_v^\gamma \neq \uparrow$, then $\forall u \in \mathcal{C}(v)$, u does not execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$. Furthermore, according to Theorems 5.6 and 5.8, we can assume that v does not execute any rule among $\mathbb{E}_{\texttt{TrustChild}}$, $\mathbb{R}_{\texttt{Give}}$, $\mathbb{R}_{\texttt{NewPlay}}$, $\mathbb{R}_{\texttt{Drop}}$, $\mathbb{R}_{\texttt{Nego}}$, $\mathbb{R}_{\texttt{OfferUp}}$, $\mathbb{R}_{\texttt{Receive}}$, $\mathbb{R}_{\texttt{Return}}$, and $\mathbb{R}_{\texttt{ReNego}}$ during cs.

Furthermore since $v \in D_a^\gamma$, it is obvious that $\texttt{tok}_v^\gamma \neq \bot$ and thus that v cannot execute $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$.

As a direct consequence, we have $\texttt{tok}_v^{\gamma'} = \texttt{tok}_v^\gamma$ and $\texttt{c}_v^{\gamma'} = \texttt{c}_v^\gamma$. Since $\texttt{tok}_v^{\gamma'} = \texttt{tok}_v^\gamma$ we can have $\texttt{W}_{\texttt{Play}}(v,\gamma') > \texttt{W}_{\texttt{Play}}(v,\gamma)$ only if $\texttt{tok}_v^\gamma \in \{\circ, \circlearrowleft, \uparrow, \bullet\}$. Since $\texttt{tok}_v^\gamma = \texttt{tok}_v^{\gamma'}$ and for the cases where $\texttt{tok}_v^\gamma \in \{\bullet, \circlearrowleft, \circ\}$, we have no children $u \in \mathcal{C}(v)$ executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs, then we fall in the preconditions of Lemmas 5.25, 5.26, and 5.27. Whatever the value of $\texttt{tok}_v^\gamma$, the corresponding lemma guarantees that $\texttt{W}_{\texttt{Play}}(v,\gamma') \leq \texttt{W}_{\texttt{Play}}(v,\gamma)$. ■

Theorem 5.9 establishes that $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}},\texttt{W}_{\texttt{Play}})$ is pre-admissible.

Theorem 5.9

Let $cs = \gamma \to \gamma'$ be a computing step, and let a be an anchor at γ', and suppose that a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs.

We have $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}},\texttt{W}_{\texttt{Play}})(D_a^{\gamma'},\gamma') \preceq^4 (\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}},\texttt{W}_{\texttt{Play}})(D_a^\gamma,\gamma)$.

Proof: Theorem 5.7 establishes $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^{\gamma'},\gamma') \preceq^3 (\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^\gamma,\gamma)$. Suppose now $\texttt{W}_{\texttt{Play}}(D_a^{\gamma'},\gamma') > \texttt{W}_{\texttt{Play}}(D_a^\gamma,\gamma)$, and prove $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^{\gamma'},\gamma') \prec^3 (\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^\gamma,\gamma)$.

In the same way as we did in the proof of Theorem 5.5, we can suppose that for any branch $\mathcal{B}$ of G_a, $D_a^{\gamma'}(\mathcal{B})$ is not longer than $D_a^\gamma(\mathcal{B})$.

In such circumstances, we have $\texttt{W}_{\texttt{Play}}(D_a^{\gamma'},\gamma') > \texttt{W}_{\texttt{Play}}(D_a^\gamma,\gamma)$ only if there exists a branch $\mathcal{B}$ of G_a such that $\exists v \in D_a^\gamma(\mathcal{B}) : \texttt{W}_{\texttt{Play}}(v,\gamma') > \texttt{W}_{\texttt{Play}}(v,\gamma)$. But then, according to Lemma 5.28, we have $(\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^{\gamma'},\gamma') \prec^3 (\texttt{W}_{\texttt{Ch}},\texttt{W}_{\texttt{Er}},\texttt{W}_{\texttt{Circ}})(D_a^\gamma,\gamma)$. ■

Theorem 5.10 proves that under certain circumstances, we are certain that the weight of one $\texttt{CD}^o$ decreases. It will be useful to prove that $\texttt{W}$ is admissible, but also to prove that it is pre-admissible.

Theorem 5.10

Let $cs = \gamma \to \gamma'$ be a computing step, let a be an anchor at γ' that does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs.
If $\exists v \in D^\gamma_a$ and $u \in \mathcal{C}_{G_a}(v)$ such that:

- $\texttt{tok}^\gamma_v = \bullet$ *and* $\texttt{c}^\gamma_u \neq \texttt{c}^\gamma_v$ *and u executes $\mathbb{R}_{\texttt{Lose}}$ during cs, or*
- $\texttt{tok}^\gamma_v \in \{\circlearrowleft, \circ\}$ *and* $\texttt{c}^\gamma_u \neq \texttt{c}^\gamma_v$ *and u executes $\mathbb{R}_{\texttt{ReplayD}}$ during cs, or*
- $\texttt{tok}^\gamma_v = \uparrow$ *and* $\texttt{c}^\gamma_u = \texttt{c}^\gamma_v$ *and u executes $\mathbb{R}_{\texttt{Fake}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs,*

we have $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^{\gamma'}_a, \gamma') \prec^4 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^\gamma_a, \gamma)$.

Proof: Let us consider such nodes v and u, and one branch $\mathcal{B}$ of G_a such that $v \in D^\gamma_a(\mathcal{B})$. If $v \notin D^{\gamma'}_a(\mathcal{B})$, or if $\texttt{tok}^{\gamma'}_v \neq \texttt{tok}^\gamma_v$, or if $\texttt{tok}^\gamma_v \in \{\bullet, \circlearrowleft, \circ\}$ and one child u of v executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$, then we according to Lemma 5.18, or Theorems 5.6 and 5.8, or Theorem 5.4, we have

$$(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^{\gamma'}_a, \gamma') \prec^4 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^\gamma_a, \gamma)$$

We can now suppose that $v \in D^{\gamma'}_a(\mathcal{B})$ and $\texttt{tok}^{\gamma'}_v = \texttt{tok}^\gamma_v$ and if $\texttt{tok}^\gamma_v \in \{\bullet, \circlearrowleft, \circ\}$ then no child of v executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$. Depending on which of the three situations has to be considered, Lemma 5.25, Lemma 5.26, or Lemma 5.27 applies and guarantees that $\texttt{W}_{\texttt{Play}}(v, \gamma') < \texttt{W}_{\texttt{Play}}(v, \gamma)$. From Theorem 5.9 we conclude

$$(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^{\gamma'}_a, \gamma') \prec^4 (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(D^\gamma_a, \gamma).$$

■

5.4.2.8 Fifth component of W: $\texttt{W}_{\texttt{Nego}}$ negotiation between one parent and its children

Explanations Only one aspect of the algorithm has not been treated yet: the negotiation between one node v that has the token $\texttt{tok}_v = \bullet$ and its children $u \in \texttt{Players}(v)$. This negotiation occurs only if there are several nodes in $\texttt{Players}(v)$, otherwise v decides that the negotiation is terminated, and executes $\mathbb{R}_{\texttt{Give}}$ or $\mathbb{R}_{\texttt{OfferUp}}$ depending on whether there exist some candidates for the token or not. Furthermore, let us remark that if $\texttt{Players}(v)$ evolves during an execution, whether by an addition (with $\mathbb{R}_{\texttt{ReplayUp}}$) or a deletion (with $\mathbb{R}_{\texttt{Lose}}$ for example), then according to Theorems 5.4 and 5.10 the weight of the $\texttt{CD}^o$ which contains v decreases. Thus, we can suppose in the following that we work under the following hypothesis: $\texttt{Players}(v)$ is constant, and there are at least two nodes in it.

Due to the impact of $\texttt{PhasePlus}(v)$ on variable $\texttt{ph}_v$, we cannot guarantee at first sight that the negotiation does not cycle. Indeed, if all the children u of v declare that they cannot give an answer for $\texttt{ph}_v$, then v decides to restart from $\texttt{ph}_v = 1$ so it does not eliminate all of its children. But as soon as v can restart, we must prove that it does not restart indefinitely, what would create a livelock in the negotiation phase.

To overcome this problem, we statically determine the first value $\texttt{ph}_v$ which, when taken by v, will push one child $u \in \texttt{Players}(v)$ to execute $\mathbb{R}_{\texttt{Lose}}$. Such values exist since we supposed that there are at least two nodes in $\texttt{Players}(v)$ and that the identifiers are globally unique. If we prove that ultimately, some node u can only execute $\mathbb{R}_{\texttt{Lose}}$ and thus makes $\texttt{W}_{\texttt{Play}}$ decrease, we prove that the negotiation is livelock-free. Unfortunately, it is not trivial to determine the value of the first incoming value of $\texttt{ph}_v$ which will induce an execution of $\mathbb{R}_{\texttt{Lose}}$. Basically, we are looking for the lowest value greater or equal to $\texttt{ph}_v$ which induces different answers on two children $u \in \texttt{Players}(v)$.

One first subtlety comes with the fact that, since we work in the framework of self-stabilization, there might not be such greater value. Indeed, if the initial value of $\texttt{ph}_v$ is different from 1, and if all the values of $\texttt{ph}_v$ which might differentiate two children $u \in \texttt{Players}(v)$ are less than this initial value, then the differentiation will not happen before a reset of $\texttt{ph}_v$. Under such circumstances, v executes a full round of all the possible value for $\texttt{ph}$, increasing as long as its children u produce answers different from $\texttt{b}_u = \bot$, and when they all, at the same moment by hypothesis, cannot, v restart from $\texttt{ph}_v = 1$. Then, v will increase $\texttt{ph}_v$ until it reaches one value which differentiates two of its children $u \in \texttt{Players}(v)$.

One other, trickier, subtlety coming from the self-stabilizing framework, is that the registers of the children might be incorrectly initialized. If, due to an incorrect initialization, we miss an occasion to differentiate two children in $\texttt{Players}(v)$, a similar scheme will happen. In such situation, v increases $\texttt{ph}_v$ until it reaches the next value which differentiates two of its children of $\texttt{Players}(v)$. It might even happen that the initial value of $\texttt{ph}_v$ is the only value which can differentiate any two children of $\texttt{Players}(v)$. In this situation, node v will execute a full round of all the possible value for $\texttt{ph}_v$, increasing as long as all of its children produce answers different from $\texttt{b}_u = \bot$, then restarting from $\texttt{ph}_v = 1$ until it reaches again the initial value which differentiates its children. On the contrary, an incorrect initialization of the values of $\texttt{b}_u$ on some node $u \in \texttt{Players}(v)$ might implies that u executes $\mathbb{R}_{\texttt{Lose}}$ while it is not supposed to, according to its actual $\texttt{id}$. That last case will be said to be a false positive.

Incorrect initializations of variable $\texttt{b}_u$ might only happen at the very beginning of the execution, and might be taken into account only if u does not update it before any action if its parent, *i.e.* if $\neg\texttt{AnswerPar}(u, v)$. Then, as soon as u takes a step, either it executes $\mathbb{R}_{\texttt{Lose}}$, either it updates $\texttt{b}_u$ according to its identifier, and then we will not face any incorrect initialization anymore. Thus, we have to consider the possibilities of wrong initializations only in few cases, basically only for the current value of $\texttt{ph}_v$ and only if $\neg\texttt{AnswerPar}(u, v)$. For any other situation, $\texttt{Bit}_u(\texttt{ph})$ is the adapted value, which describes the value of $\texttt{b}_u$ taken into account during the negotiation.

To summarize, we can distinguish three situations.

- The first, simplest, situation, is when the current value of $\texttt{ph}_v$ actually differentiates at least two children of $\texttt{Players}(v)$. This situation includes false positive, when $\texttt{ph}_v$ is not meant to play this role regarding to identifiers, and excludes situations where an incorrect initialization of the $\texttt{b}_u$ prevents the expected differentiation to actually happen.

- The second situation is when the current value of $\texttt{ph}_v$ does not differentiate two children of $\texttt{Players}(v)$, but there exists a value greater than $\texttt{ph}_v$ which does. In such a situation, we only have to wait for $\texttt{ph}_v$ to increase until it reaches the first such value, and we fall in the previous situation.

- The third and last situation is when neither the current value of $\texttt{ph}_v$, neither any value greater than it, can differentiate two children of $\texttt{Players}(v)$. This situation includes, as an extreme case, the situation where the current value of $\texttt{ph}_v$ is the only one which might differentiate two children of $\texttt{Players}(v)$, but an incorrect initialization of some $\texttt{b}_u$ prevents the differentiation from being effective. In this third situation, we wait for $\texttt{ph}_v$ to increase until the value for which all children of $\texttt{Players}(v)$, at the same moment, answer $\bot$. At this moment, $\texttt{ph}_v$ is set to 1 and then we fall in one of the previous situations.

We want to design a function dealing with as many aspects, or as many rules, of the algorithm as possible. Thus, we intend to define $\texttt{W}_{\texttt{Nego}}$ such that any action involving $\mathbb{R}_{\texttt{maxPos}}$,

$\mathbb{R}_{\texttt{NewPh}}$, or $\mathbb{R}_{\texttt{NewBit}}$, makes $\texttt{W}_{\texttt{Nego}}$ decrease. It is pretty natural to treat $\mathbb{R}_{\texttt{NewPh}}$ since it increases (or reduces down to 1) the value of $\texttt{ph}_v$, and thus bring closer to the expected value. Since any two execution of $\mathbb{R}_{\texttt{NewPh}}$ are separated by an execution of $\mathbb{R}_{\texttt{maxPos}}$, which updates $\texttt{b}_v$ from $\bot$ to a value different from $\bot$, we will not have much trouble to use those properties in the design of $\texttt{W}_{\texttt{Nego}}$. Finally, we want $\texttt{W}_{\texttt{Nego}}$ to decrease any time one child $u \in \texttt{Players}(v)$ executes $\mathbb{R}_{\texttt{NewBit}}$. In other words, we must, in a certain way, count the number of children of v which can execute $\mathbb{R}_{\texttt{NewBit}}$. But any execution of $\mathbb{R}_{\texttt{NewPh}}$ recreates the conditions under which all the nodes $u \in \texttt{Players}(v)$ can execute $\mathbb{R}_{\texttt{NewBit}}$, and thus if we simply decrease $\texttt{W}_{\texttt{Nego}}$ by one for the execution of $\mathbb{R}_{\texttt{NewPh}}$, but it increases by $|\texttt{Players}(v)|$ at the same time, we fail. Two options are left to us. First, we can define $\texttt{W}_{\texttt{Nego}}$ as a 2-tuple, the first component focuses on the progress of $\texttt{ph}_v$, and the second one focuses on the decreasing of the number of nodes which can execute $\mathbb{R}_{\texttt{NewBit}}$. The other option is to deal this in one single function, and to give some coefficients to the different components involved. In a certain sense, it boils down to the same as the previous option, but instead of using the lexicographic order, we define manually high coefficients on the first component so that it all works the same with one integer value. To avoid making this proof even more tedious, we chose the second option.

Definitions Before properly defining $\texttt{W}_{\texttt{Nego}}$, let us first formally determine the three different situations which were described above, whether we reached the expected value for $\texttt{ph}_v$, or whether not but there we will reach this value with only increasing $\texttt{ph}_v$, or whether we will have to reset $\texttt{ph}_v$ to 1 before we reach it.

Recall that to decide if the current value of $\texttt{ph}_v$ will differentiate two children $u \in \texttt{Players}(v)$, we cannot simply rely on the values of $\texttt{Bit}_u(\texttt{ph}_v)$. Indeed, we must take into account that the register of $\texttt{b}_v$ and/or $\texttt{b}_u$ might be incorrectly initialized. If this happens, and if the current state of one node u can be interpreted as an answer by its parent, then it does not update it, and thus $\texttt{b}_u$ is the value used to decide what the future of the negotiation is. Otherwise, u has to first execute $\mathbb{R}_{\texttt{NewBit}}$ before anything else, and then erase its previous state and only $\texttt{Bit}_u(\texttt{ph}_v)$ is relevant. Thus, we first formally define what actual value will be taken as the answer by the parent v of u for the current step of the negotiation, which depends for each node u on whether it will produce a new answer to its parent, or not.

$$\texttt{Answer}^\gamma(u,v) = \begin{cases} \texttt{Bit}_u(\texttt{ph}_v^\gamma) & \text{if } \texttt{AnswerPar}^\gamma(u,v) \\ \texttt{b}_u^\gamma & \text{otherwise} \end{cases} \tag{5.50}$$

Let us now detail the situations in which the current value of $\texttt{ph}_v$ differentiates two children $u \in \texttt{Players}^\gamma(v)$. The first, most natural, situation is when two different nodes will produce different values for $\texttt{b}$ in response to $\texttt{ph}_v^\gamma$. If this happens, then after v executes $\mathbb{R}_{\texttt{maxPos}}$ (or before, if we already have $\texttt{b}_v^\gamma \neq \bot$, at least one of them will be in a situation where it must execute $\mathbb{R}_{\texttt{Lose}}$.

$$\texttt{TwoDiffer}^\gamma(v) \equiv \exists u_1, u_2 \in \texttt{Players}^\gamma(v) : \texttt{Answer}^\gamma(u_1, v) \neq \texttt{Answer}^\gamma(u_2, v) \tag{5.51}$$

The previous captures almost all the situations where one node is close to execute $\mathbb{R}_{\texttt{Lose}}$. There exists one other, unusual, situation, which is not described here, which is when all the nodes $u \in \texttt{Players}^\gamma(v)$ will produce the same value for $\texttt{ph}_v$, but due to an incorrect initialization of variables, v has already picked a value, which is different from the common value proposed by its children. In this situation, all the children will actually execute $\mathbb{R}_{\texttt{Lose}}$ at very short term.

$$\texttt{OneDiffers}^\gamma(v) \equiv \texttt{b}_v^\gamma \neq \bot \wedge \exists u \in \texttt{Players}^\gamma(v) : \texttt{Answer}^\gamma(u, v) \neq \texttt{b}_v^\gamma \tag{5.52}$$

We can now define the first of our three cases, which corresponds to the situation where one node $u \in \texttt{Players}^\gamma$ will execute $\mathbb{R}_{\texttt{Lose}}$ before any update of $\texttt{ph}_v$.

$$\texttt{Finished}^\gamma(v) \equiv \texttt{TwoDiffer}^\gamma(v) \vee \texttt{OneDiffers}^\gamma(v) \tag{5.53}$$

Let us now discuss the two other situations, in which the current value $\texttt{ph}_v^\gamma$ does not differentiate any two children of $\texttt{Players}^\gamma(v)$. We need to determine whether v will reset $\texttt{ph}_v$ to 1 before we reach a value which differentiates two of its children of $\texttt{Players}^\gamma(v)$, or whether not. We could think, at first, that it is sufficient to determine whether there exists one value greater than $\texttt{ph}_v^\gamma$ which differentiates two nodes of $\texttt{Players}^\gamma(v)$ or not. If there is not such greater value, $\texttt{ph}_v$ will indeed be reset to 1 before we terminate the negotiation phase. Yet, it might happen that, although there exists such a greater value, we are nevertheless forced to reset $\texttt{ph}_v$ to 1 before we reach this greater value. Indeed, it could happen that, due to an incorrect initialization of variables $\texttt{b}_u$, all the children $u \in \texttt{Players}^\gamma(v)$ produce $\perp$ as an answer for $\texttt{ph}_v^\gamma$ while there exists a greater value which would differentiate two of them. If this happens, then the execution of $\texttt{PhasePlus}(v)$ will reset $\texttt{ph}_v = 1$. Hopefully, this is can happen only due to incorrect initialization of variables, and thus can only happen for the initial value of $\texttt{ph}_v$. Remark that if only few of the nodes produce $\perp$ as an answer for $\texttt{ph}_v^\gamma$ then we have $\texttt{Finished}^\gamma(v)$. Let us define the predicate $\texttt{ForcedReset}(v)$ which corresponds to configurations where an execution of $\mathbb{R}_{\texttt{NewPh}}$ by v will set $\texttt{ph}_v$ to 1.

$$\texttt{ForcedReset}^\gamma(v) \equiv \forall u \in \texttt{Players}^\gamma(v), \texttt{Answer}^\gamma(u, v) = \perp \tag{5.54}$$

If $\texttt{ForcedReset}^\gamma(v)$, then we are sure that v will first reset $\texttt{ph}_v = 1$. But if not, we must determine whether there exists one value greater than $\texttt{ph}_v^\gamma$ which differentiates two nodes of $\texttt{Players}^\gamma(v)$ or not. Let us first define $\texttt{Separators}^\gamma(v)$ the set of all the values for $\texttt{ph}_v$ which can differentiate two children $u \in \texttt{Players}^\gamma(v)$. Remark that this set is defined statically: it does not depend at all on the values of $\texttt{ph}_v, \texttt{b}_v, \texttt{ph}_u, \texttt{b}_u$, and only relies on the values of the different identifiers of children of v in $\texttt{Players}^\gamma(v)$. Yet, we consider this set only if $\neg\texttt{Finished}^\gamma(v)$, that is to say if we must execute $\mathbb{R}_{\texttt{NewPh}}$ at least once before reaching the value that differentiates children of v. By definition, after v executes $\mathbb{R}_{\texttt{NewPh}}$, we necessarily have $\texttt{AnswerPar}(u, v)$ for all children $u \in \texttt{Players}^\gamma(v)$. Thus, in such situations, $\texttt{Bit}_u$ is the relevant tool to decide whether some value for $\texttt{ph}_v$ will differentiate children of v. Remark also that $\texttt{Separators}^\gamma(v)$ only depends on $\texttt{Players}^\gamma(v)$, which will be supposed constant in the following. Consequently, we will suppose $\texttt{Separators}^\gamma(v)$ constant too. Finally, since we are interested in considering the negotiation phase, v has at least two children, and then $\texttt{Separators}^\gamma(v)$ contains at least one value.

$$\texttt{Separators}^\gamma(v) = \{\texttt{ph} \in [0, \lceil \log n \rceil] \mid \exists u_1, u_2 \in \texttt{Players}^\gamma(v) : \texttt{Bit}_{u_1}(\texttt{ph}) \neq \texttt{Bit}_{u_2}(\texttt{ph})\} \tag{5.55}$$

Let us now define the two other predicates, which correspond to situations where node v will not reset $\texttt{ph}_v$ to 1 before it reaches a value which differentiates two of its children, and to the situation where it will.

$$\texttt{WontReset}^\gamma(v) \equiv \neg\texttt{Finished}^\gamma(v) \wedge \begin{cases} \exists i > \texttt{ph}_v^\gamma : i \in \texttt{Separators}^\gamma(v) \\ \wedge \\ \neg\texttt{ForcedReset}^\gamma(v) \end{cases} \tag{5.56}$$

$$\texttt{WillReset}^\gamma(v) \equiv \neg\texttt{Finished}^\gamma(v) \wedge \begin{cases} \forall i > \texttt{ph}_v^\gamma, i \notin \texttt{Separators}^\gamma(v) \\ \vee \\ \texttt{ForcedReset}^\gamma(v) \end{cases} \tag{5.57}$$

Remark that the three predicates $\texttt{Finished}$, $\texttt{WontReset}$, and $\texttt{WillReset}$ are mutually exclusive, and that $\forall \gamma \in \Gamma, \forall v \in V, \texttt{Finished}^\gamma(v) \vee \texttt{WontReset}^\gamma(v) \vee \texttt{WillReset}^\gamma(v)$. This justifies the use of these three predicates to distinguish cases in any situation.

Let us now define, for each of these three cases, the value of $\texttt{ph}$ which, when reach by $\texttt{ph}_v$, will provoke one execution of $\mathbb{R}_{\texttt{Lose}}$ on one child $u \in \texttt{Players}^\gamma(v)$.

$$\texttt{SepVal}^\gamma(v) = \begin{cases} \texttt{ph}_v^\gamma & \text{if } \texttt{Finished}^\gamma(v) \\ \min\{i \mid i \in \texttt{Separators}^\gamma(v) \wedge i > \texttt{ph}_v^\gamma\} & \text{if } \texttt{WontReset}^\gamma(v) \\ \min\{i \mid i \in \texttt{Separators}^\gamma(v)\} & \text{if } \texttt{WillReset}^\gamma(v) \end{cases} \tag{5.58}$$

Finally, we are able to define one first aspect of $\texttt{W}_{\texttt{Nego}}$, which is the distance between $\texttt{ph}_v^\gamma$ and $\texttt{SepVal}^\gamma(v)$, which corresponds to the number of executions of $\mathbb{R}_{\texttt{NewPh}}$ required for v. This definition is very simple in both situations $\texttt{Finished}^\gamma(v)$ and $\texttt{WontReset}^\gamma(v)$. It is slightly more difficult when $\texttt{WillReset}^\gamma(v)$ since we must determine how high $\texttt{ph}_v$ will reach before it is reset to 1. Two situations might occur.

If, due to an incorrect initialization of variables, all the children u of $\texttt{Players}^\gamma(v)$ produce $\bot$ as an answer for $\texttt{ph}_v^\gamma$, then the next execution of $\mathbb{R}_{\texttt{NewPh}}$ by v will set $\texttt{ph}_v$ to 1 and then $\texttt{ph}_v^\gamma$ is the maximal value which will be reached before $\texttt{ph}_v$ is reset to 1.

Otherwise, there will be at least one execution of $\mathbb{R}_{\texttt{NewPh}}$ by v before we reach the maximal value, and thus only the functions $\texttt{Bit}_u$ are relevant. In this last case, the maximal value reached before $\texttt{ph}_v$ is reset to 1 is the lowest value such that, when reached, all the children u of $\texttt{Players}^\gamma(v)$ produce $\bot$ as an answer.

$$\texttt{MaxPh}^\gamma(v) = \begin{cases} \texttt{ph}_v^\gamma & \text{if } \texttt{ForcedReset}^\gamma(v) \\ \min\{i > \texttt{ph}_v^\gamma \mid \forall u \in \texttt{Players}^\gamma(v) : \texttt{Bit}_u(i) = \bot\} & \text{if } \neg\texttt{ForcedReset}^\gamma(v) \end{cases} \tag{5.59}$$

Finally, the definition of $\texttt{SepDist}(v)$ which is the number of executions of $\mathbb{R}_{\texttt{NewPh}}$ v has to make before we have $\texttt{ph}_v = \texttt{SepVal}^\gamma(v)$.

$$\texttt{SepDist}^\gamma(v) = \begin{cases} 0 & \text{if } \texttt{Finished}^\gamma(v) \\ \texttt{SepVal}^\gamma(v) - \texttt{ph}_v^\gamma & \text{if } \texttt{WontReset}^\gamma(v) \\ \texttt{SepVal}^\gamma(v) + (\texttt{MaxPh}^\gamma(v) - \texttt{ph}_v^\gamma) & \text{if } \texttt{WillReset}^\gamma(v) \end{cases} \tag{5.60}$$

We now formally define the weight of one node v. The goal is to count the number of actions each node, among v and its children in $\texttt{Players}^\gamma(v)$, can take before the execution of $\mathbb{R}_{\texttt{Lose}}$ is necessary.

Node v will execute two actions for each increase of $\texttt{ph}_v$, since it must execute both $\mathbb{R}_{\texttt{NewPh}}$, and $\mathbb{R}_{\texttt{maxPos}}$ for each step. A node $u \in \texttt{Players}^\gamma(v)$ executes $\mathbb{R}_{\texttt{Lose}}$ when $\texttt{b}_v \neq \bot$, which means immediately after one execution of $\mathbb{R}_{\texttt{maxPos}}$, and not of $\mathbb{R}_{\texttt{NewPh}}$. Thus, v executes a sequence of $\mathbb{R}_{\texttt{NewPh}}; \mathbb{R}_{\texttt{maxPos}}$ in this order, which might be preceded by one execution of $\mathbb{R}_{\texttt{maxPos}}$ if necessary.

$$\texttt{ParSteps}^\gamma(v) = 2 \times \texttt{SepDist}^\gamma(v) + \begin{cases} 1 & \text{if } \texttt{b}_v^\gamma = \bot \\ 0 & \text{otherwise} \end{cases} \tag{5.61}$$

One children $u \in \texttt{Players}^\gamma(v)$ only executes $\mathbb{R}_{\texttt{NewBit}}$, by hypothesis. Every time it executes $\mathbb{R}_{\texttt{NewBit}}$, u sets $\texttt{ph}_u$ at the current value of $\texttt{ph}_v$, which will be updated exactly $\texttt{SepDist}^\gamma(v)$ times. Thus, u executes $\mathbb{R}_{\texttt{NewBit}}$ once for each updated value of $\texttt{ph}_v$, and execute one more action if, in the initial configuration, u has to produce an answer for $\texttt{ph}_v$.

$$\texttt{ChSteps}^\gamma(u, v) = \texttt{SepDist}^\gamma(v) + \begin{cases} 1 & \text{if } \texttt{AnswerPar}^\gamma(u, v) \\ 0 & \text{otherwise} \end{cases} \tag{5.62}$$

We finally define $\mathtt{W_{Nego}}$.

$$\mathtt{W_{Nego}}(v,\gamma) = \begin{cases} \mathtt{ParSteps}^{\gamma}(v) + \sum_{u \in \mathtt{Players}^{\gamma}(v)} \mathtt{ChSteps}^{\gamma}(u,v) & \text{if } \mathtt{tok}_v^{\gamma} = \bullet \\ 0 & \text{otherwise} \end{cases} \tag{5.63}$$

Preliminaries Our overall goal is to prove that $\mathtt{W_{Nego}}$ behaves as expected, which is proving that associated with the four previous components, it is pre-admissible, and that in some specific cases, it is a decreasing function. Before we formally establish this, we need to establish some basic, yet numerous, properties which describe how the different predicates and sets we defined above behave during an execution. Indeed, since the definition of $\mathtt{W_{Nego}}$ strongly depends on `Finished`, `WontReset`, and `WillReset`, among other things. In this preliminary, we focus on establishing properties of stability for those three predicates, depending on the activation of node v. To achieve this, we first prove some stability properties on other, more basic, predicates, namely `AnswerPar`, `Answer`, `ForcedReset`. Some of those preliminary lemmas will be reused as well in the demonstrations of the next section.

Our goal is to prove that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})$ is pre-admissible. Thus, according to Theorem 5.9, most of the proof will be done under the hypothesis that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})$ is constant. Actually, this is hypothesis is explicitly used only in the proof of Lemma 5.29, which states that $\mathtt{Players}(v)$ and $\mathtt{Separators}(v)$ are constant. In all the other lemmas, this hypothesis on $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})$ is actually used through Lemma 5.29: we suppose $\mathtt{Players}(v)$ and $\mathtt{Separators}(v)$ are constant.

Lemma 5.29

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma)$.

Then $\mathtt{Players}^{\gamma'}(v) = \mathtt{Players}^{\gamma}(v) \wedge \mathtt{Separators}^{\gamma'}(v) = \mathtt{Separators}^{\gamma}(v)$.

Proof: Let us first remark that $\mathtt{c}_v^{\gamma'} = \mathtt{c}_v^{\gamma}$ according to the rules. Let us first consider one node $u \in \mathtt{Players}^{\gamma}(v)$, *i.e.* $\mathtt{tok}_u^{\gamma} = \bot \wedge \mathtt{c}_u^{\gamma} \neq \mathtt{c}_v^{\gamma} \wedge \mathtt{play}_u^{\gamma} = \mathtt{P}$. If u executes $\mathbb{R}_{\mathtt{Lose}}$ then according to Lemma 5.25, we have $\mathtt{W_{Play}}(v,\gamma') < \mathtt{W_{Play}}(v,\gamma)$ which cannot happen by hypothesis. If u executes $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ then according to Lemma 5.16, we have $\mathtt{W_{Ch}}(v,\gamma') < \mathtt{W_{Ch}}(v,\gamma)$ which cannot happen by hypothesis. Thus, u can only execute $\mathbb{R}_{\mathtt{NewBit}}$, and whatever happens, $u \in \mathtt{Players}^{\gamma'}(v)$.

Let us now consider one node $u \notin \mathtt{Players}^{\gamma}(v)$. If $\mathtt{c}_u^{\gamma} = \mathtt{c}_v^{\gamma}$ then we can have $u \in \mathtt{Players}^{\gamma'}(v)$ only if $\mathtt{c}_u^{\gamma'} \neq \mathtt{c}_v^{\gamma'}$ which is possible only if u executes $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ but then according to Lemma 5.16, we have $\mathtt{W_{Ch}}(v,\gamma') < \mathtt{W_{Ch}}(v,\gamma)$ which cannot happen by hypothesis. Otherwise, if $\mathtt{tok}_u^{\gamma} \neq \bot$ then we can have $u \in \mathtt{Players}^{\gamma'}(v)$ only if $\mathtt{tok}_u^{\gamma'} = \bot$, which is possible only if u executes $\mathbb{R}_{\mathtt{Return}}$, but then the execution of $\mathtt{Drop}(u)$ implies that $\mathtt{play}_u^{\gamma'} = \mathtt{F}$ and thus $u \notin \mathtt{Players}^{\gamma'}(v)$. Finally, if $\mathtt{c}_u^{\gamma} \neq \mathtt{c}_v^{\gamma}$ and $\mathtt{tok}_u^{\gamma} = \bot$, then $\mathtt{play}_u^{\gamma} \neq \mathtt{P}$. We can have $u \in \mathtt{Players}^{\gamma'}(v)$ only if $\mathtt{play}_u^{\gamma'} = \mathtt{P}$, which is possible only if u executes $\mathbb{R}_{\mathtt{ReplayD}}$ or $\mathbb{R}_{\mathtt{ReplayUp}}$. Due to its parent v, u cannot execute $\mathbb{R}_{\mathtt{ReplayD}}$, and if it executes $\mathbb{R}_{\mathtt{ReplayD}}$, then according to Lemma 5.16, we have $\mathtt{W_{Ch}}(v,\gamma') < \mathtt{W_{Ch}}(v,\gamma)$ which cannot happen by hypothesis. As a consequence, in any case, $u \notin \mathtt{Players}^{\gamma'}(v)$.

We just established that $u \in \mathtt{Players}^{\gamma'}(v) \iff u \in \mathtt{Players}^{\gamma}(v)$, which means that $\mathtt{Players}^{\gamma'}(v) = \mathtt{Players}^{\gamma}(v)$. Since $\mathtt{Separators}^{\gamma}(v)$ only depends on $\mathtt{Players}^{\gamma}(v)$, we can conclude that $\mathtt{Separators}^{\gamma'}(v) = \mathtt{Separators}^{\gamma}(v)$ as well. ■

From now on, since all the following lemmas of this section are under, at least, the hypothesis of Lemma 5.29, we allow ourselves to simply write $\mathtt{Players}(v)$ and $\mathtt{Separators}(v)$ without referring to the configuration in which the set is evaluated.

Lemma 5.30 proves that, unless node v asks for one new information, which concerns one new value for $\mathtt{ph}$, then the value which will be taken as an answer for the negotiation does not evolve. It partly justifies the definition of $\mathtt{Answer}$.

Lemma 5.30

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^\gamma = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.

If v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs then $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Answer}^{\gamma}(u, v)$.

Proof: Let us consider $u \in \mathtt{Players}(v)$. If u is activated, then according to Theorems 5.4 and 5.10, u executes $\mathbb{R}_{\mathtt{NewBit}}$. Therefore, $\mathtt{OkNeg}^\gamma(u)$ so v is the only parent of u with $\mathtt{tok}_v \notin \{\bot, \uparrow, \downarrow\}$, and, on the other hand, $\mathtt{AnswerPar}^\gamma(u, v)$. Therefore, by definition of the action $\mathtt{Announce}(u)$, $\mathtt{Answer}^{\gamma'}(u, v) = \mathtt{b}_u^{\gamma'} = \mathtt{Bit}_u(\mathtt{ph}_v^\gamma) = \mathtt{Answer}^\gamma(u, v)$. Suppose now that u is not activated during cs. If v is not activated either, then we obviously have $\mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Answer}^\gamma(u, v)$. Let us rather suppose that v executes $\mathbb{R}_{\mathtt{maxPos}}$ during cs.

According to Lemma 5.20, we have $\neg\mathtt{AnswerPar}^\gamma(u, v)$, so $\mathtt{Answer}^\gamma(u, v) = \mathtt{b}_u^\gamma$, and since u is not activated during cs we even have $\mathtt{b}_u^{\gamma'} = \mathtt{b}_u^\gamma$. By definition, we have $\mathtt{Synch}^\gamma(v)$ so $\mathtt{ph}_u^\gamma = \mathtt{ph}_v^\gamma$, which implies, according to $\mathtt{NextPlay}(v)$, that $\mathtt{ph}_u^{\gamma'} = \mathtt{ph}_v^{\gamma'}$. Since $\mathtt{b}_v^{\gamma'} \neq \bot$, we have $\neg\mathtt{AnswerPar}^{\gamma'}(u, v)$, so $\mathtt{Answer}^{\gamma'}(u, v) = \mathtt{b}_u^{\gamma'} = \mathtt{b}_u^\gamma = \mathtt{Answer}^\gamma(u, v)$. ■

Lemma 5.32 proves that once the predicate $\mathtt{Finished}$ evaluates to $\mathtt{true}$, then it does in all the following configurations too. In order to prove it, we first establish Lemma 5.31 which proves that when we have $\mathtt{Finished}^\gamma(v)$, node v cannot execute $\mathbb{R}_{\mathtt{NewPh}}$, which justifies the name of this predicate. Lemma 5.32 is the first fundamental lemma which will be helpful in the next section.

Lemma 5.31

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^\gamma = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.

Then $\mathtt{Finished}^\gamma(v) \Rightarrow \neg(\mathtt{Synch}^\gamma(v) \wedge \mathtt{PhComplete}^\gamma(v))$.

Proof: Suppose that $\mathtt{Finished}^\gamma(v) \wedge \mathtt{Synch}^\gamma(v)$ and prove that $\neg\mathtt{PhComplete}^\gamma(v)$. Whether we have $\mathtt{TwoDiffer}^\gamma(v)$ or $\mathtt{OneDiffers}^\gamma(v)$, there exists $u \in \mathtt{Players}(v) : \mathtt{Answer}^\gamma(u, v) \neq \mathtt{b}_v^\gamma$. If $\mathtt{AnswerPar}^\gamma(u, v)$ then according to Lemma 5.20, $\neg\mathtt{PhComplete}^\gamma(v)$. Otherwise, $\neg\mathtt{AnswerPar}^\gamma(u, v)$ and then $\mathtt{b}_u^\gamma \neq \mathtt{b}_v^\gamma$ and once again, $\neg\mathtt{PhComplete}^\gamma(v)$. ■

Lemma 5.32

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^\gamma = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.

Then $(\mathtt{TwoDiffer}^\gamma(v) \Rightarrow \mathtt{TwoDiffer}^{\gamma'}(v)) \wedge (\mathtt{OneDiffers}^\gamma(v) \Rightarrow \mathtt{OneDiffers}^{\gamma'}(v))$, and thus $\mathtt{Finished}^\gamma(v) \Rightarrow \mathtt{Finished}^{\gamma'}(v)$.

Proof: If $\mathtt{TwoDiffer}^\gamma(v)$ then $\mathtt{Finished}^\gamma(v)$ so according to Lemma 5.31, v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs, and thus according to Lemma 5.30, $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Answer}^\gamma(u, v)$. Thus, let us consider $u_1, u_2 \in \mathtt{Players}(v)$ such that $\mathtt{Answer}^\gamma(u_1, v) \neq \mathtt{Answer}^\gamma(u_2, v)$. Immediately, we have $\mathtt{Answer}^\gamma(u_1, v) \neq \mathtt{Answer}^\gamma(u_2, v)$ and thus $\mathtt{TwoDiffer}^{\gamma'}(v)$.

Similarly, if $\mathtt{OneDiffers}^\gamma(v)$, then $\mathtt{Finished}^\gamma(v)$ so according to Lemma 5.31, u does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs, and thus $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Answer}^\gamma(u, v)$. By definition of $\mathtt{OneDiffers}^\gamma(v)$, we have $\mathtt{b}_v^\gamma \neq \bot$, so v does not execute $\mathbb{R}_{\mathtt{maxPos}}$ during cs,

which implies that v is not activated during cs. Consequently, $\mathtt{b}_v^{\gamma'} = \mathtt{b}_v^{\gamma} \neq \bot$, and since $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u,v) = \mathtt{Answer}^{\gamma}(u,v)$, then $\mathtt{OneDiffers}^{\gamma'}(v)$.

Since $\mathtt{Finished} = \mathtt{TwoDiffer} \vee \mathtt{OneDiffers}$, we deduce $\mathtt{Finished}^{\gamma}(v) \Rightarrow \mathtt{Finished}^{\gamma'}(v)$. ■

The two following lemmas prove that, when v does not execute $\mathbb{R}_{\mathtt{NewPh}}$, then the predicates `ForcedReset` and `Finished` remain constant.

Lemma 5.33 proves that, as long as v does not update $\mathtt{ph}_v$, then whether the next execution of $\mathbb{R}_{\mathtt{NewPh}}$ will reset $\mathtt{ph}_v = 1$ or not, $\mathtt{ForcedReset}(v)$ keeps the same boolean value. It, partly, justifies the definition of `ForcedReset`.

Lemma 5.33

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma)$.
If v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs then $\mathtt{ForcedReset}^{\gamma'}(v) \iff \mathtt{ForcedReset}^{\gamma}(v)$.

Proof: Since `ForcedReset` only depends on `Answer`, this is a direct consequence of Lemma 5.30 ■

Lemma 5.34 establishes that, as long as v does not update $\mathtt{ph}_v$, then whether we have reached the final value for $\mathtt{ph}_v$ or not, $\mathtt{Finished}(v)$ keeps the same boolean value. In a certain sense, it specifies Lemma 5.32.

Lemma 5.34

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v,\gamma)$.
If v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs then $\mathtt{Finished}^{\gamma'}(v) \iff \mathtt{Finished}^{\gamma}(v)$.

Proof: Since v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs, $\mathtt{ph}_v^{\gamma'} = \mathtt{ph}_v^{\gamma}$. According to Lemma 5.30, we have $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u,v) = \mathtt{Answer}^{\gamma}(u,v)$.

If v is not activated during cs, then $\mathtt{b}_v^{\gamma'} = \mathtt{b}_v^{\gamma}$ and thus we have $\mathtt{Finished}^{\gamma'}(v) \iff \mathtt{Finished}^{\gamma}(v)$. Now suppose that v executes $\mathbb{R}_{\mathtt{maxPos}}$ during cs, which means $\mathtt{b}_v^{\gamma} = \bot$ and $\mathtt{b}_v^{\gamma'} \neq \bot$.

Since $\mathtt{b}_v^{\gamma'} \neq \bot$, $\mathtt{OneDiffers}^{\gamma'}(v) \equiv \exists u \in \mathtt{Players}(v) : \mathtt{Answer}^{\gamma'}(u,v) \neq \mathtt{b}_v^{\gamma'}$. But since $\exists u \in \mathtt{Players}(v) : \mathtt{Answer}^{\gamma'}(v) = \mathtt{b}_v^{\gamma'}$ due to $\mathtt{MaxPos}(v)$, we have $\mathtt{OneDiffers}^{\gamma'}(v) \equiv \exists u_1, u_2 \in \mathtt{Players}(v) : \mathtt{Answer}^{\gamma'}(u_1,v) \neq \mathtt{Answer}^{\gamma'}(u_2,v)$. Thus, $\mathtt{OneDiffers}^{\gamma'}(v) \iff \mathtt{TwoDiffer}^{\gamma'}(v)$. Furthermore, since $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u,v) = \mathtt{Answer}^{\gamma}(u,v)$ we have $\mathtt{TwoDiffer}^{\gamma}(v) \iff \mathtt{TwoDiffer}^{\gamma'}(v)$.

Finally, since $\mathtt{b}_v^{\gamma} = \bot$ we have $\neg\mathtt{OneDiffers}^{\gamma}(v)$, so

$$\begin{aligned} \mathtt{Finished}^{\gamma}(v) &\iff \mathtt{TwoDiffer}^{\gamma}(v) \\ &\iff \mathtt{TwoDiffer}^{\gamma'}(v) \\ &\iff \mathtt{OneDiffers}^{\gamma'}(v) \vee \mathtt{TwoDiffer}^{\gamma'}(v) \\ &\iff \mathtt{Finished}^{\gamma'}(v). \end{aligned}$$

■

The four following lemmas give information on how an execution of $\mathbb{R}_{\mathtt{NewPh}}$ might alter some of the predicates we consider here, or establish relations between them. Lemma 5.35 proves that one node v which executes $\mathbb{R}_{\mathtt{NewPh}}$ resets $\mathtt{ph}_v = 1$ if and only if it was actually detected by `ForcedReset` before its action.

Lemma 5.35

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and

$\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.
If v *executes* $\mathbb{R}_{\texttt{NewPh}}$ *during* cs *then* $\texttt{ph}_v^{\gamma'} = 1 \iff \texttt{ForcedReset}^{\gamma}(v)$.

Proof: According to Lemma 5.20, $\forall u \in \texttt{Players}(v), \neg\texttt{AnswerPar}^{\gamma}(u, v)$. Consequently, $\forall u \in \texttt{Players}(v), \texttt{Answer}^{\gamma}(u, v) = \texttt{b}_u^{\gamma}$. Thus, $\texttt{ForcedReset}^{\gamma}(v) \equiv \forall u \in \texttt{Players}(v), \texttt{b}_u^{\gamma} = \bot$, which is exactly the condition which decides whether $\texttt{ph}_v$ is set to 1 by $\texttt{PhasePlus}(v)$. ■

Lemma 5.36 states that if node v verifies `WontReset`, then it will not reset $\texttt{ph}_v = 1$, as we expect.

Lemma 5.36

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and* $\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.
If $\texttt{WontReset}^{\gamma}(v)$ *then* v *does not reset* $\texttt{ph}_v = 1$ *during* cs.

Proof: If v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs, then this result is immediate. If v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs, then since $\texttt{WontReset}^{\gamma}(v)$, we have $\neg\texttt{ForcedReset}^{\gamma}(v)$, and thus according to Lemma 5.35, v does not reset $\texttt{ph}_v = 1$ during cs. ■

Lemma 5.37 states that if v executes $\mathbb{R}_{\texttt{NewPh}}$, *i.e.* if it asks for one new information to its children, then all of them will have to produce a new answer, so after this we can trust the values of $\texttt{Bit}_u$ in the negotiation process. Associated to Lemma 5.30 it fully justifies the definition of `Answer`.

Lemma 5.37

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and* $\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.
If v *executes* $\mathbb{R}_{\texttt{NewPh}}$ *during* cs *then* $\forall u \in \texttt{Players}(v), \texttt{AnswerPar}^{\gamma'}(u, v)$.

Proof: Let us consider one node $u \in \texttt{Players}(v)$. According to Lemma 5.20, u is not activated during cs. Furthermore, since v executes $\mathbb{R}_{\texttt{NewPh}}$, we deduce $\texttt{ph}_u^{\gamma'} = \texttt{ph}_u^{\gamma} = \texttt{ph}_v^{\gamma}$ and $\texttt{b}_u^{\gamma'} = \texttt{b}_u^{\gamma} = \texttt{b}_v^{\gamma}$. Two cases must be considered.

If $\texttt{b}_u^{\gamma} \neq \bot$ then $\texttt{PhasePlus}(v)$ updates $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1 \neq \texttt{ph}_v^{\gamma} = \texttt{ph}_u^{\gamma'}$, which reduces to $\texttt{ph}_u^{\gamma'} \neq \texttt{ph}_v^{\gamma'}$ and consequently $\texttt{AnswerPar}^{\gamma'}(u, v)$.

Otherwise, if $\texttt{b}_u^{\gamma} = \bot$, since $\forall u' \in \texttt{Players}(v), \texttt{b}_{u'}^{\gamma} = \texttt{b}_v$, we have $\forall u' \in \texttt{Players}(v), \texttt{b}_{u'}^{\gamma} = \bot$, so $\texttt{PhasePlus}(v)$ sets $\texttt{ph}_v^{\gamma'} = 1$ and $\texttt{b}_v^{\gamma'} = \bot$. Since we also have $\texttt{b}_u^{\gamma'} = \bot$, we conclude $\texttt{AnswerPar}^{\gamma'}(u, v)$. ■

Lemma 5.38 states that if one node v executes $\mathbb{R}_{\texttt{NewPh}}$, then we have reached the value on which the negotiation is finished if and only if this value belongs to the set $\texttt{Separators}(v)$. Associated with Lemmas 5.31 and 5.34 it fully specifies the situations where a computing step leads to a configuration which satisfies `Finished`.

Lemma 5.38

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and* $\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(v, \gamma)$.
If v *executes* $\mathbb{R}_{\texttt{NewPh}}$ *during* cs *then* $\texttt{Finished}^{\gamma'}(v) \iff \texttt{ph}_v^{\gamma'} \in \texttt{Separators}(v)$.

Proof: Remark first that $\texttt{b}_v^{\gamma'} = \bot$, and thus $\neg\texttt{OneDiffers}^{\gamma'}(v)$.

By definition $\forall u \in \texttt{Players}(v), \texttt{ph}_u^{\gamma} = \texttt{ph}_v^{\gamma} \wedge \texttt{b}_u^{\gamma} = \texttt{b}_v^{\gamma}$. According to Lemma 5.20, children u of v are not activated during cs, so $\forall u \in \texttt{Players}(v), \texttt{ph}_u^{\gamma'} = \texttt{ph}_v^{\gamma} \wedge \texttt{b}_u^{\gamma'} = \texttt{b}_v^{\gamma}$.

Furthermore, according to Lemma 5.37, $\forall u \in \texttt{Players}(v), \texttt{AnswerPar}^{\gamma'}(v)$. Consequently, $\forall u \in \texttt{Players}(v), \texttt{Answer}^{\gamma'}(u,v) = \texttt{Bit}_u(\texttt{ph}_v^{\gamma'})$. Since $\neg\texttt{OneDiffers}^{\gamma'}(v)$, we obtain:

$$\begin{aligned}\texttt{Finished}^{\gamma'}(v) &\iff \texttt{TwoDiffer}^{\gamma'}(v)\\ &\iff \exists u_1,u_2 : \texttt{Bit}_{u_1}(\texttt{ph}_v^{\gamma'}) \neq \texttt{Bit}_{u_1}(\texttt{ph}_v^{\gamma'})\\ &\iff \texttt{ph}_v^{\gamma'} \in \texttt{Separators}(v).\end{aligned}$$ ■

The two following lemmas, the last of this section, and establish how situations where $\neg\texttt{Finished}(v)$ can evolve.

Lemma 5.39 states that if we are in a configuration such that we will reach the terminal value for $\texttt{ph}_v$ without resetting $\texttt{ph}_v = 1$, then it remains true. Associated with Lemmas 5.34 and 5.38, it fully describes what configuration is reached after a computing step which starts with $\texttt{WontReset}(v)$.

Lemma 5.39

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} = \bullet$ and $\texttt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v,\gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v,\gamma)$.
Then $\texttt{WontReset}^{\gamma}(v) \Rightarrow \texttt{WontReset}^{\gamma'}(v) \vee \texttt{Finished}^{\gamma'}(v)$.

Proof: Recall that $\texttt{WontReset}^{\gamma}(v)$ implies $\exists i > \texttt{ph}_v^{\gamma} : i \in \texttt{Separators}(v)$ and $\neg\texttt{ForcedReset}^{\gamma}(v)$.

Suppose first that v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs. According to Lemma 5.33, we have $\neg\texttt{ForcedReset}^{\gamma'}(v)$. Since $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma}$, we also have $\exists i > \texttt{ph}_v^{\gamma'} : i \in \texttt{Separators}(v)$. Consequently, $\texttt{WontReset}^{\gamma'}(v) \vee \texttt{Finished}^{\gamma'}(v)$.

Now suppose that v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs. According to Lemma 5.36, $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma}+1$, and, by definition, $\texttt{b}_v^{\gamma'} = \bot$.

If $\texttt{ph}_v^{\gamma'} \in \texttt{Separators}(v)$ then according to Lemma 5.38 $\texttt{Finished}^{\gamma'}(v)$ and thus we obtain the desired result.

Let us now suppose that $\texttt{ph}_v^{\gamma'} \notin \texttt{Separators}(v)$. By definition of $\texttt{WontReset}^{\gamma}(v)$, we have $\exists i > \texttt{ph}_v^{\gamma} : i \in \texttt{Separators}(v)$. Since $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1 \notin \texttt{Separators}(v)$, we deduce $\exists i > \texttt{ph}_v^{\gamma'} : i \in \texttt{Separators}(v)$. We now finish the proof by establishing $\neg\texttt{ForcedReset}^{\gamma'}(v)$.

Since $\exists i > \texttt{ph}_v^{\gamma'} : i \in \texttt{Separators}(v)$, we deduce $\exists u_1,u_2 \in \texttt{Players}(v) : \texttt{Bit}_{u_1}(i) \neq \texttt{Bit}_{u_2}(i)$. At least one of them, let us say u, is such that $\texttt{Bit}_u(i) \neq \bot$. By definition of $\texttt{Bit}_u$, since $\texttt{ph}_v^{\gamma'} < i$, $\texttt{Bit}_u(\texttt{ph}_v^{\gamma'}) \neq \bot$, and by Lemma 5.37, we deduce $\texttt{Answer}^{\gamma'}(u,v) \neq \bot$. Thus, $\neg\texttt{ForcedReset}^{\gamma'}(v)$, and finally $\texttt{WontReset}^{\gamma'}(v) \vee \texttt{Finished}^{\gamma'}(v)$. ■

Lemma 5.40 states that if we are in a configuration such that we need to reset $\texttt{ph}_v = 1$ before reaching the terminal value for $\texttt{ph}_v$, then this remains true after a computing step, unless we do reset $\texttt{ph}_v = 1$ during this computing step. Associated with Lemmas 5.35 and 5.38, it fully describes what configuration is reached after a computing step which starts with $\texttt{WillReset}(v)$.

Lemma 5.40

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} = \bullet$ and $\texttt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v,\gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v,\gamma)$.
If $\texttt{WillReset}^{\gamma}(v)$, we have:
$\neg\texttt{WillReset}^{\gamma'}(v) \iff$ v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs and $\texttt{ph}_v^{\gamma'} = 1$.

Proof: Suppose that $\texttt{WillReset}^{\gamma}(v)$, *i.e.* $\neg\texttt{Finished}^{\gamma}(v) \wedge (\forall i > \texttt{ph}_v^{\gamma}, i \notin \texttt{Separators}(v) \vee \texttt{ForcedReset}^{\gamma}(v))$.

If v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs then according to Lemma 5.34, $\neg\texttt{Finished}^{\gamma'}(v)$, and according to Lemma 5.33, $\texttt{ForcedReset}^{\gamma'}(v) \iff \texttt{ForcedReset}^{\gamma}(v)$. Finally, since

$\mathtt{ph}_v^{\gamma'} = \mathtt{ph}_v^{\gamma}$, we have $\forall i > \mathtt{ph}_v^{\gamma'}, i \notin \mathtt{Separators}(v) \iff \forall i > \mathtt{ph}_v^{\gamma}, i \notin \mathtt{Separators}(v)$. Consequently, we have $\mathtt{WillReset}^{\gamma'}(v)$.

Suppose now that v executes $\mathbb{R}_{\mathtt{NewPh}}$ during cs. According to Lemma 5.37 $\forall u \in \mathtt{Players}(v), \mathtt{AnswerPar}^{\gamma'}(u, v)$ which implies $\forall u \in \mathtt{Players}(u, v), \mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Bit}_u(\mathtt{ph}_v^{\gamma'})$.

Let us first prove that if $\mathtt{ph}_v^{\gamma'} \neq 1$ then $\mathtt{WillReset}^{\gamma'}(v)$. Remark that $\neg\mathtt{ForcedReset}^{\gamma}(v)$. Indeed, according to Lemma 5.20 we have $\forall u \in \mathtt{Players}(v), \neg\mathtt{AnswerPar}^{\gamma}(u, v)$, and thus if $\mathtt{ForcedReset}^{\gamma}(v)$ then $\forall u \in \mathtt{Players}(v), \mathtt{b}_u^{\gamma} = \bot$ and thus the execution of $\mathtt{PhasePlus}(v)$ set $\mathtt{ph}_v$ to 1. But now, $\mathtt{WillReset}^{\gamma}(v) \iff \neg\mathtt{Finished}^{\gamma'}(v) \wedge \forall i > \mathtt{ph}_v^{\gamma}, i \notin \mathtt{Separators}(v)$. Thus, $\mathtt{ph}_v^{\gamma'} \notin \mathtt{Separators}(v)$, so according to Lemma 5.38, $\neg\mathtt{Finished}^{\gamma'}(v)$. Furthermore, we have $\forall i > \mathtt{ph}_v^{\gamma'}, i \notin \mathtt{Separators}(v)$, and thus $\mathtt{WillReset}^{\gamma'}(v)$.

Let us now prove that if $\mathtt{ph}_v^{\gamma'} = 1$ then $\neg\mathtt{WillReset}^{\gamma'}(v)$. If $1 \in \mathtt{Separators}(v)$ then according to Lemma 5.38, $\mathtt{Finished}^{\gamma'}(v)$ and thus $\neg\mathtt{WillReset}^{\gamma'}(v)$. Otherwise, $1 \notin \mathtt{Separators}(v)$, and since $\mathtt{Separators}(v) \neq \emptyset, \exists i > \mathtt{ph}_v^{\gamma'} : i \in \mathtt{Separators}(v)$. Let us now prove that $\neg\mathtt{ForcedReset}^{\gamma'}(v)$ to finish the proof. Since v executes $\mathbb{R}_{\mathtt{NewPh}}$ during cs, $\mathtt{ForcedReset}^{\gamma'}(v) \iff \forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(1) = \bot$, which cannot be by hypothesis on the identifiers. Thus, $\exists i > \mathtt{ph}_v^{\gamma'} : i \in \mathtt{Separators}(v) \wedge \neg\mathtt{ForcedReset}^{\gamma'}(v)$ so $\mathtt{Finished}^{\gamma'}(v) \vee \mathtt{WontReset}^{\gamma'}(v)$. ■

Proofs In this section, we finally address the function $\mathtt{W}_{\mathtt{Nego}}$, the lemmas established in the previous section are going to be extremely useful.

One first step is to establish that what we defined as the last value for $\mathtt{ph}_v$ during the negotiation phase, $\mathtt{SepVal}(v)$, is consistent through an execution. We prove in Lemma 5.42 that this value remains constant. Let us first prove Lemma 5.41 which states that unless we reset $\mathtt{ph}_v = 1$, then the maximum value for $\mathtt{ph}_v$ which we will reach before the reset remains constant as well.

Lemma 5.41

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\mathtt{W}_{\mathtt{Ch}}, \mathtt{W}_{\mathtt{Er}}, \mathtt{W}_{\mathtt{Circ}}, \mathtt{W}_{\mathtt{Play}})(v, \gamma') = (\mathtt{W}_{\mathtt{Ch}}, \mathtt{W}_{\mathtt{Er}}, \mathtt{W}_{\mathtt{Circ}}, \mathtt{W}_{\mathtt{Play}})(v, \gamma)$.

If v does not reset $\mathtt{ph}_v = 1$ during cs, we have $\mathtt{MaxPh}^{\gamma'}(v) = \mathtt{MaxPh}^{\gamma}(v)$.

Proof: If v does not execute $\mathbb{R}_{\mathtt{NewPh}}$ during cs then $\mathtt{ph}_v^{\gamma'} = \mathtt{ph}_v^{\gamma}$, and according to Lemma 5.33 $\mathtt{ForcedReset}^{\gamma'}(v) \iff \mathtt{ForcedReset}^{\gamma}(v)$, and thus $\mathtt{MaxPh}^{\gamma'}(v) = \mathtt{MaxPh}^{\gamma}(v)$.

Suppose now that v executes $\mathbb{R}_{\mathtt{NewPh}}$ during cs. By hypothesis, $\mathtt{ph}_v$ is not reset to 1 during cs, so $\mathtt{ph}_v^{\gamma'} = \mathtt{ph}_v^{\gamma} + 1$, and according to Lemma 5.35 we also have $\neg\mathtt{ForcedReset}^{\gamma}(v)$. Thus, we deduce $\mathtt{MaxPh}^{\gamma}(v) = \min\{i \geq \mathtt{ph}_v^{\gamma} \mid \forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(i) = \bot\}$. On the other hand, since v executes $\mathbb{R}_{\mathtt{NewPh}}$ during cs, we know by Lemma 5.37 that $\forall u \in \mathtt{Players}(v), \mathtt{AnswerPar}^{\gamma'}(u, v)$, which implies $\forall u \in \mathtt{Players}(v), \mathtt{Answer}^{\gamma'}(u, v) = \mathtt{Bit}_u(\mathtt{ph}_v^{\gamma'})$. Thus, $\mathtt{ForcedReset}^{\gamma'}(v) \equiv \forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(\mathtt{ph}_v^{\gamma'}) = \bot$, which is equivalent to $\mathtt{MaxPh}^{\gamma}(v) = \mathtt{ph}_v^{\gamma'}$.

If $\mathtt{ForcedReset}^{\gamma'}(v)$ then immediately, we conclude $\mathtt{MaxPh}^{\gamma'}(v) = \mathtt{MaxPh}^{\gamma}(v)$. Let us suppose $\neg\mathtt{ForcedReset}^{\gamma'}(v)$, which implies $\mathtt{MaxPh}^{\gamma'}(v) = \min\{i \geq \mathtt{ph}_v^{\gamma'} \mid \forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(i) = \bot\}$. But we also know $\neg(\forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(\mathtt{ph}_v^{\gamma'}) = \bot)$, so we deduce $\mathtt{MaxPh}^{\gamma'}(v) = \min\{i \geq \mathtt{ph}_v^{\gamma'} - 1 \mid \forall u \in \mathtt{Players}(v), \mathtt{Bit}_u(i) = \bot\}$, and as a consequence, $\mathtt{MaxPh}^{\gamma'}(v) = \mathtt{MaxPh}^{\gamma}(v)$. ■

Lemma 5.42

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\mathtt{tok}_v^{\gamma} = \bullet$ and

$\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$. *Then* $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$.

Proof: Let us distinguish three cases:

- If $\texttt{Finished}^{\gamma}(v)$, then according to Lemma 5.31, v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs, and thus $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma}$. Furthermore, according to Corollary 5.32, $\texttt{Finished}^{\gamma'}(v)$. Thus, we have $\texttt{SepVal}^{\gamma'}(v) = \texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} = \texttt{SepVal}^{\gamma}(v)$.
- If $\texttt{WontReset}^{\gamma}(v)$, two cases must be considered.

  If v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs, then $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma}$, and according to Lemma 5.34, $\neg\texttt{Finished}^{\gamma'}(v)$, and according to Lemma 5.30, $\forall u \in \texttt{Players}(v), \texttt{Answer}^{\gamma'}(u, v) = \texttt{Answer}^{\gamma}(u, v)$. Thus, we have $\texttt{WontReset}^{\gamma'}(v)$, and thus $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$.

  If v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs, then by definition of $\texttt{WontReset}^{\gamma}(v)$ the execution of $\texttt{PhasePlus}(v)$ sets $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1$ and $\texttt{b}_v^{\gamma'} = \bot$. According to Lemma 5.37, $\forall u \in \texttt{Players}(v), \texttt{AnswerPar}^{\gamma'}(u, v)$, which implies $\forall u \in \texttt{Players}(v), \texttt{Answer}^{\gamma'}(u, v) = \texttt{Bit}_u(\texttt{ph}_v^{\gamma'})$.
  - If $\texttt{ph}_v^{\gamma'} \in \texttt{Separators}(v)$, then, by definition, $\texttt{SepVal}^{\gamma}(v) = \texttt{ph}_v^{\gamma'}$. Furthermore, according to Lemma 5.38, $\texttt{Finished}^{\gamma'}(v)$, and thus $\texttt{SepVal}^{\gamma'}(v) = \texttt{ph}_v^{\gamma'}$. This prove $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$.
  - If $\texttt{ph}_v^{\gamma'} \notin \texttt{Separators}(v)$ then according to Lemmas 5.39 and 5.38 $\texttt{WontReset}^{\gamma'}(v)$. Furthermore, since $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1 \notin \texttt{Separators}(v)$, we deduce $\min\{i \mid i \in \texttt{Separators}(v) \wedge i > \texttt{ph}_v^{\gamma}\} = \min\{i \mid i \in \texttt{Separators}(v) \wedge i > \texttt{ph}_v^{\gamma'}\}$. In other words, $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$
- If $\texttt{WillReset}^{\gamma}(v)$, let us consider two cases.

  If v executes $\mathbb{R}_{\texttt{NewPh}}$ and sets $\texttt{ph}_v^{\gamma'} = 1$, then according to Lemma 5.40, $\texttt{Finished}^{\gamma'}(v) \vee \texttt{WontReset}^{\gamma'}(v)$, and according to Lemma 5.38, $\texttt{Finished}^{\gamma'}(v) \iff 1 \in \texttt{Separators}(v)$. Thus, if $1 \in \texttt{Separators}(v)$ then $\texttt{SepVal}^{\gamma}(v) = 1$ and $\texttt{SepVal}^{\gamma'}(v) = \texttt{ph}_v^{\gamma'} = 1$, and thus $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$. Otherwise, if $1 \notin \texttt{Separators}(v)$, then $\neg\texttt{Finished}^{\gamma'}(v)$, so $\texttt{WontReset}^{\gamma'}(v)$. We have

  $$\begin{aligned}\texttt{SepVal}^{\gamma}(v) &= \min\{i \mid i \in \texttt{Separators}(v)\}\\ &= \min\{i \mid i \in (\texttt{Separators}(v) \setminus \{1\})\}\\ &= \min\{i \mid i \in \texttt{Separators}(v) \wedge i > \texttt{ph}_v^{\gamma'}\}\\ &= \texttt{SepVal}^{\gamma'}(v)\end{aligned}$$

  and thus $\texttt{SepVal}^{\gamma'}(v) = \texttt{SepVal}^{\gamma}(v)$

  Let us now suppose that v does not execute $\mathbb{R}_{\texttt{NewPh}}$, or that, if it does, then it does not set $\texttt{ph}_v = 1$. According to Lemma 5.40, $\texttt{WillReset}^{\gamma'}(v)$, and we deduce $\texttt{SepVal}^{\gamma'}(v) = \min\{i \mid i \in \texttt{Separators}(v)\} = \texttt{SepVal}^{\gamma}(v)$. ■

From now on, for all the lemmas of this section which are stated under, at least, the hypotheses of Lemma 5.42, which by the way are the same as the hypothesis of Lemma 5.29, we allow ourselves to simply write $\texttt{SepVal}(v)$ without referring to the configuration in which this function is evaluated.

This being established, we are going to follow the different components of $\texttt{W}_{\texttt{Nego}}$ introduced for its definition: $\texttt{SepDist}$, $\texttt{ParSteps}$, $\texttt{ChSteps}$, and finished $\texttt{W}_{\texttt{Nego}}$ itself. For each of those four functions, we prove that it corresponds to a non-increasing function, and for each of them we present the cases where we can guarantee its decrease.

Lemma 5.43

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and*

$\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$.
Then $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$.

Proof: Let us consider three cases.

- If $\texttt{Finished}^{\gamma}(v)$ then according to Corollary 5.32, $\texttt{Finished}^{\gamma'}(v)$ and thus $\texttt{SepDist}^{\gamma'}(v) = \texttt{SepDist}^{\gamma}(v) = 0$.
- If $\texttt{WontReset}^{\gamma}(v)$, then $\texttt{SepDist}^{\gamma}(v) = \texttt{SepVal}(v) - \texttt{ph}_v^{\gamma}$. If $\texttt{Finished}^{\gamma'}(v)$ then $\texttt{SepDist}^{\gamma'}(v) = 0 \leq \texttt{SepDist}^{\gamma}(v)$. Otherwise, according to Lemma 5.39 $\texttt{WontReset}^{\gamma'}(v)$. Furthermore, according to Lemma 5.36, $\texttt{ph}_v^{\gamma'} \geq \texttt{ph}_v^{\gamma}$, so $\texttt{SepVal}(v) - \texttt{ph}_v^{\gamma'} \leq \texttt{SepVal}(v) - \texttt{ph}_v^{\gamma}$. Consequently, we have $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$.
- If $\texttt{WillReset}^{\gamma}(v)$, then $\texttt{SepDist}^{\gamma}(v) = \texttt{SepVal}(v) + (\texttt{MaxPh}^{\gamma}(v) - \texttt{ph}_v^{\gamma})$.
  If v does not reset $\texttt{ph}_v^{\gamma'} = 1$ during cs, then according to Lemma 5.40 $\texttt{WillReset}^{\gamma'}(v)$, and according to Lemma 5.41 $\texttt{MaxPh}^{\gamma'}(v) = \texttt{MaxPh}^{\gamma}(v)$. Furthermore, by hypothesis $\texttt{ph}_v^{\gamma'} \geq \texttt{ph}_v^{\gamma}$, so $\texttt{SepVal}(v) + (\texttt{MaxPh}^{\gamma'}(v) - \texttt{ph}_v^{\gamma'}) \leq \texttt{SepVal}(v) + (\texttt{MaxPh}^{\gamma}(v) - \texttt{ph}_v^{\gamma}$. Thus, we deduce $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$.
  Otherwise, if v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs, and sets $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1$, then according to Lemma 5.40 $\neg\texttt{WillReset}^{\gamma'}(v)$, and thus we deduce $\texttt{SepDist}^{\gamma}(v) \leq \texttt{SepVal}(v)$. Since by definition $\texttt{MaxPh}^{\gamma}(v) - \texttt{ph}_v^{\gamma} \geq 0$, we have $\texttt{SepDist}^{\gamma}(v) \geq \texttt{SepVal}(v)$. By transitivity, $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$. ■

Lemma 5.44

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and* $\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$.
If v *executes* $\mathbb{R}_{\texttt{NewPh}}$ *during* cs *we have* $\texttt{SepDist}^{\gamma'}(v) < \texttt{SepDist}^{\gamma}(v)$.

Proof: Let us consider three cases.

- If $\texttt{Finished}^{\gamma}(v)$ then according to Lemma 5.31, v does not execute $\mathbb{R}_{\texttt{NewPh}}$ during cs.
- If $\texttt{WontReset}^{\gamma}(v)$ then if $\texttt{Finished}^{\gamma'}(v)$, we have $\texttt{SepDist}^{\gamma'}(v) = 0 < \texttt{SepDist}^{\gamma}(v)$. Indeed, since $\texttt{WontReset}^{\gamma}(v)$, we have $\texttt{SepVal}(v) > \texttt{ph}_v^{\gamma}$, and so $\texttt{SepDist}^{\gamma}(v) > 0$.
  Let us now suppose that $\neg\texttt{Finished}^{\gamma'}(v)$. According to Lemma 5.39 $\texttt{WontReset}^{\gamma'}(v)$. Furthermore, according to Lemma 5.36, $\texttt{ph}_v^{\gamma'} > \texttt{ph}_v^{\gamma}$, and thus $\texttt{SepVal}(v) - \texttt{ph}_v^{\gamma'} < \texttt{SepVal}(v) - \texttt{ph}_v^{\gamma}$. Consequently, $\texttt{SepDist}^{\gamma'}(v) < \texttt{SepDist}^{\gamma}(v)$.
- If $\texttt{WillReset}^{\gamma}(v)$ then let us consider two options.
  If $\texttt{ph}_v^{\gamma'} = \texttt{ph}_v^{\gamma} + 1$ then according to Lemma 5.40 $\texttt{WillReset}^{\gamma'}(v)$, and according to Lemma 5.41 $\texttt{MaxPh}^{\gamma'}(v) = \texttt{MaxPh}^{\gamma}(v)$. Thus, we deduce $\texttt{SepVal}(v) + (\texttt{MaxPh}^{\gamma'}(v) - \texttt{ph}_v^{\gamma'}) < \texttt{SepVal}(v) + (\texttt{MaxPh}^{\gamma}(v) - \texttt{ph}_v^{\gamma}$, which means $\texttt{SepDist}^{\gamma'}(v) < \texttt{SepDist}^{\gamma}(v)$.
  If $\texttt{ph}_v^{\gamma'} = 1$ then according to Lemma 5.40 $\neg\texttt{WillReset}^{\gamma'}(v)$. By definition of $\texttt{MaxPh}^{\gamma}(v)$, we have $\texttt{MaxPh}^{\gamma}(v) \geq \texttt{ph}_v$, so $\texttt{SepDist}^{\gamma}(v) \geq \texttt{SepVal}(v)$. On the other hand, whether $\texttt{Finished}^{\gamma'}(v)$ or $\texttt{WontReset}^{\gamma'}(v)$, we have $\texttt{SepDist}^{\gamma}(v) < \texttt{SepVal}(v)$. As a consequence, we obtain $\texttt{SepDist}^{\gamma'}(v) < \texttt{SepDist}^{\gamma}(v)$. ■

Lemma 5.45

Let $cs = \gamma \to \gamma'$ *be a computing step and let* v *be a node such that* $\texttt{tok}_v^{\gamma} = \bullet$ *and* $\texttt{tok}_v^{\gamma'} = \bullet$, *and suppose that* $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$.
Then $\texttt{ParSteps}^{\gamma'}(v) \leq \texttt{ParSteps}^{\gamma}(v)$, *and if* v *executes* $\mathbb{R}_{\texttt{maxPos}}$ *or* $\mathbb{R}_{\texttt{NewPh}}$ *during* cs, *then* $\texttt{ParSteps}^{\gamma'}(v) < \texttt{ParSteps}^{\gamma}(v)$.

Proof: Remark first that if v is not activated during cs, then according to Lemma 5.43 $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$. On the other hand, $\texttt{b}_v^{\gamma'} = \texttt{b}_v^{\gamma}$, so $\texttt{ParSteps}^{\gamma'}(v) \leq \texttt{ParSteps}^{\gamma}(v)$.

If v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs then according to Lemma 5.44,

$$\begin{array}{rrl} & \texttt{SepDist}^{\gamma'}(v) + 1 & \leq \texttt{SepDist}^{\gamma}(v) \\ \Rightarrow & 2 \times \texttt{SepDist}^{\gamma'}(v) + 2 & \leq 2 \times \texttt{SepDist}^{\gamma}(v) \\ \Rightarrow & 2 \times \texttt{SepDist}^{\gamma'}(v) + 1 & < 2 \times \texttt{SepDist}^{\gamma}(v). \end{array}$$

Furthermore, $\texttt{ParSteps}^{\gamma'}(v) \leq 2 \times \texttt{SepDist}^{\gamma'}(v) + 1$ and $\texttt{ParSteps}^{\gamma}(v) \geq 2 \times \texttt{SepDist}^{\gamma}(v)$. By transitivity, we deduce $\texttt{ParSteps}^{\gamma'}(v) < \texttt{ParSteps}^{\gamma}(v)$.

Finally, if v executes $\mathbb{R}_{\texttt{maxPos}}$ during cs, then according to Lemma 5.43 $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$. Furthermore, we have $\texttt{b}_v^{\gamma} = \bot$ and $\texttt{b}_v^{\gamma'} \neq \bot$, so $\texttt{ParSteps}^{\gamma'}(v) = 2 \times \texttt{SepDist}^{\gamma'}(v) \leq 2 \times \texttt{SepDist}^{\gamma}(v) < 2 \times \texttt{SepDist}^{\gamma}(v) + 1 = \texttt{ParSteps}^{\gamma}(v)$, which reduces to $\texttt{ParSteps}^{\gamma'}(v) < \texttt{ParSteps}^{\gamma}(v)$. ■

Lemma 5.46

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} = \bullet$ and $\texttt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$.

Let us consider $u \in \texttt{Players}(v)$. Then $\texttt{ChSteps}^{\gamma'}(v) \leq \texttt{ChSteps}^{\gamma}(v)$, and if u executes $\mathbb{R}_{\texttt{NewBit}}$ during cs, then $\texttt{ChSteps}^{\gamma'}(v) < \texttt{ChSteps}^{\gamma}(v)$.

Proof: Suppose first that u does not execute $\mathbb{R}_{\texttt{NewBit}}$.

If v is not activated, then $\texttt{ChSteps}^{\gamma'}(v) = \texttt{ChSteps}^{\gamma}(v)$.

If v executes $\mathbb{R}_{\texttt{NewPh}}$ during cs then according to Lemma 5.44, $\texttt{SepDist}^{\gamma'}(v) < \texttt{SepDist}^{\gamma}(v)$, and thus $\texttt{ChSteps}^{\gamma'}(u, v) \leq \texttt{ChSteps}^{\gamma}(u, v)$.

If v executes $\mathbb{R}_{\texttt{maxPos}}$ during cs then no children of v is activated during cs according to Lemma 5.20, and thus, $\texttt{Synch}^{\gamma}(v)$. Consequently, $\texttt{ph}_u^{\gamma'} = \texttt{ph}_u^{\gamma} = \texttt{ph}_v^{\gamma} = \texttt{ph}_v^{\gamma'}$. Furthermore, since $\texttt{b}_v^{\gamma'} \neq \bot$, we obtain $\neg\texttt{AnswerPar}^{\gamma'}(u, v)$. According to Lemma 5.43, $\texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v)$ so we deduce $\texttt{ChSteps}^{\gamma'}(u, v) \leq \texttt{ChSteps}^{\gamma}(u, v)$.

Now suppose that u executes $\mathbb{R}_{\texttt{NewBit}}$ during cs. According to Lemma 5.20, v is not activated during cs. Furthermore, we have, by definition, $\texttt{AnswerPar}^{\gamma}(u, v)$. The execution of $\texttt{Announce}(u)$ sets $\texttt{ph}_u^{\gamma'} = \texttt{ph}_u^{\gamma} = \texttt{ph}_u^{\gamma'}$ and $\texttt{b}_u^{\gamma'} = \texttt{Bit}_u(\texttt{ph}_v^{\gamma})$. If $\texttt{ph}_v^{\gamma} = 1$ then $\texttt{Bit}_u(\texttt{ph}_v^{\gamma}) \neq \bot$, so $\neg\texttt{AnswerPar}^{\gamma'}(v)$, and if $\texttt{ph}_v^{\gamma} \neq 1$ then $\texttt{ph}_v^{\gamma'} \neq 1$ and thus $\neg\texttt{AnswerPar}^{\gamma'}(v)$. Thus, we conclude $\texttt{ChSteps}^{\gamma'}(u, v) = \texttt{SepDist}^{\gamma'}(v) \leq \texttt{SepDist}^{\gamma}(v) < \texttt{SepDist}^{\gamma}(v) + 1 = \texttt{ChSteps}^{\gamma}(u, v)$. By transitivity, $\texttt{ChSteps}^{\gamma'}(u, v) < \texttt{ChSteps}^{\gamma}(u, v)$. ■

Lemma 5.47

Let $cs = \gamma \to \gamma'$ be a computing step and let v be a node such that $\texttt{tok}_v^{\gamma} = \bullet$ and $\texttt{tok}_v^{\gamma'} = \bullet$, and suppose that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma') = (\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}})(v, \gamma)$.

Then $\texttt{W}_{\texttt{Nego}}(v, \gamma') \leq \texttt{W}_{\texttt{Nego}}(v, \gamma)$, and if v executes $\mathbb{R}_{\texttt{NewPh}}$ or $\mathbb{R}_{\texttt{maxPos}}$ during cs, or if $\exists u \in \texttt{Players}(v)$ such that u executes $\mathbb{R}_{\texttt{NewBit}}$ during cs, then $\texttt{W}_{\texttt{Nego}}(v, \gamma') < \texttt{W}_{\texttt{Nego}}(v, \gamma)$.

Proof: This is a direct consequence of Lemmas 5.45 and 5.46. ■

Theorem 5.11 establishes that $(\texttt{W}_{\texttt{Ch}}, \texttt{W}_{\texttt{Er}}, \texttt{W}_{\texttt{Circ}}, \texttt{W}_{\texttt{Play}}, \texttt{W}_{\texttt{Nego}})$ is pre-admissible.

Theorem 5.11

Let $cs = \gamma \to \gamma'$ be a computing step and let a be an anchor at γ' such that a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs.

$(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma'}, \gamma') \preceq^5 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma}, \gamma)$.

Proof: According to Theorem 5.9,

$$(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma'}, \gamma') \preceq^4 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma}, \gamma).$$

If $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma'}, \gamma') \prec^4 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma}, \gamma)$, then the result is trivial. Let us now suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma'}, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma}, \gamma)$, and prove that $\mathtt{W_{Nego}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W_{Nego}}(D_a^{\gamma}, \gamma)$.

In the same way we did in the proof of Theorem 5.5, we can suppose that for any branch $\mathcal{B}$ of G_a, $D_a^{\gamma'}(\mathcal{B})$ is not longer than $D_a^{\gamma}(\mathcal{B})$. In such circumstances, we have $\mathtt{W_{Nego}}(D_a^{\gamma'}, \gamma') \succ \mathtt{W_{Nego}}(D_a^{\gamma}, \gamma)$ only if there exists a branch $\mathcal{B}$ of G_a such that $\exists v \in D_a^{\gamma'}(\mathcal{B})$ such that $\mathtt{W_{Nego}}(v, \gamma') > \mathtt{W_{Nego}}(v, \gamma)$.

Let us consider one such node $v \in D_a^{\gamma'}(\mathcal{B})$. According to Theorems 5.6 and 5.8, we have $\mathtt{tok}_v^{\gamma'} = \mathtt{tok}_v^{\gamma}$. If $\mathtt{tok}_v^{\gamma} \neq \bullet$, then $\mathtt{W_{Nego}}(v, \gamma') = 0 = \mathtt{W_{Nego}}(v, \gamma)$. If $\mathtt{tok}_v^{\gamma} = \bullet$ and $\mathtt{tok}_v^{\gamma'} = \bullet$, then we fall under the hypothesis of Lemma 5.47 and thus $\mathtt{W_{Nego}}(v, \gamma') \leq \mathtt{W_{Nego}}(v, \gamma)$.

Thus, for any branch $\mathcal{B}$ of G_a, for any node $v \in D_a^{\gamma'}(\mathcal{B})$, $\mathtt{W_{Nego}}(v, \gamma') \leq \mathtt{W_{Nego}}(v, \gamma)$, and thus $\mathtt{W_{Nego}}(D_a^{\gamma'}, \gamma') \preceq \mathtt{W_{Nego}}(D_a^{\gamma}, \gamma)$. ■

Theorem 5.12 establishes that under certain circumstances, we are certain that the weight of one $\mathtt{CD}^o$ decreases. It will we useful to prove the admissibility of $\mathtt{W}$.

Theorem 5.12

Let $cs = \gamma \to \gamma'$ be a computing step and let a be an anchor at γ' such that a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs.

If $\exists v \in D_a^{\gamma}$ such that $\mathtt{tok}_v^{\gamma} = \bullet$ and either v executes $\mathbb{R}_{\mathtt{NewPh}}$ or $\mathbb{R}_{\mathtt{maxPos}}$ during cs, either $\exists u \in \mathtt{Players}^{\gamma}(v)$ such that u executes $\mathbb{R}_{\mathtt{NewBit}}$ during cs, then

$(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma'}, \gamma') \prec^5 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma}, \gamma)$

Proof: According to Theorem 5.11,

$$(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma'}, \gamma') \preceq^5 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma}, \gamma).$$

If $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma'}, \gamma') \prec^4 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma}, \gamma)$, then the result is trivial. Let us now suppose that $(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma'}, \gamma') = (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}})(D_a^{\gamma}, \gamma)$, and prove that $\mathtt{W_{Nego}}(D_a^{\gamma'}, \gamma') \prec \mathtt{W_{Nego}}(D_a^{\gamma}, \gamma)$.

Let us consider $v \in D_a^{\gamma}$ such that $\mathtt{tok}_v^{\gamma} = \bullet$ and either v executes $\mathbb{R}_{\mathtt{NewPh}}$ or $\mathbb{R}_{\mathtt{maxPos}}$ during cs, either $\exists u \in \mathtt{Players}^{\gamma}(v)$ such that u executes $\mathbb{R}_{\mathtt{NewBit}}$ during cs. According to Theorems 5.6 and 5.8, we have $\mathtt{tok}_v^{\gamma'} = \bullet$. Thus, we fall under the hypothesis of Lemma 5.47 and we have $\mathtt{W_{Nego}}(v, \gamma') \prec \mathtt{W_{Nego}}(v, \gamma)$. Thus, $\mathtt{W_{Nego}}(D_a^{\gamma'}, \gamma') \neq \mathtt{W_{Nego}}(D_a^{\gamma}, \gamma)$ so according to Theorem 5.11, we conclude that

$$(\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma'}, \gamma') \prec^5 (\mathtt{W_{Ch}}, \mathtt{W_{Er}}, \mathtt{W_{Circ}}, \mathtt{W_{Play}}, \mathtt{W_{Nego}})(D_a^{\gamma}, \gamma).$$

■

5.4.2.9 Conclusions

In this section we combine the results of Sections 5.4.2.4 to 5.4.2.8 to provide a suitable definition of *admissible potential function*. More precisely, we provide a definition of admissibility which is wide enough to include our weight function $\mathtt{W}$, and tight enough to be convenient in the following proofs.

The final theorem of this section states that our algorithm satisfies Condition 2 of Specification 5.1, but other intermediate theorems will be used in the following section.

From now on, to ease the reading, when we use comparison operations on 5-tuple induced by $\mathtt{W}$, we simply use the symbol $\prec$ rather than $\prec^5$ to denote the lexicographic order on 5-tuples of $\mathtt{WD}^o$'s.

Theorem 5.13 states that $\mathtt{W}$ is pre-admissible.

Theorem 5.13

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_m$ be an execution. If a is an anchor at γ_m, and if a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during ϵ, then $\mathtt{W}(D_a^{\gamma_m}) \preceq \mathtt{W}(D_a^{\gamma_0})$.

Proof: By definition, $\forall t \in [0, m-1]$, a does not execute $\mathbb{R}^r_{\mathtt{NewDFS}}$ during $\gamma_t \to \gamma_{t+1}$, so according to Theorem 5.11, $\forall t \in [0, m-1], \mathtt{W}(D_a^{\gamma_{t+1}}) \preceq \mathtt{W}(D_a^{\gamma_t})$. By transitivity, we conclude $\mathtt{W}(D_a^{\gamma_m}) \preceq \mathtt{W}(D_a^{\gamma_0})$. ■

As hinted above, to define admissibility we must first interest to what an activation of a $\mathtt{CD}^o$ is.

Definition 5.24 (*Activation of a* $\mathtt{CD}^o$)

Let $cs = \gamma \to \gamma'$ be a computing step, let a be an anchor at γ', and let $u \in V$ be a node. We say that u activates D_a^γ during cs if:

- *$u = a$ and u executes $\mathbb{R}^r_{\mathtt{NewDFS}}$ during cs, or*
- *$u \in \mathbb{B}(D_a^\gamma)$ and u executes $\mathbb{R}_{\mathtt{Win}}$ or $\mathbb{R}_{\mathtt{FakeWin}}$ during cs, or*
- *$u \in D_a^\gamma$ and u executes $\mathbb{E}_{\mathtt{TrustChild}}$ or $\mathbb{R}_{\mathtt{Give}}$ or $\mathbb{R}_{\mathtt{NewPlay}}$ or $\mathbb{R}_{\mathtt{Drop}}$ or $\mathbb{R}_{\mathtt{Nego}}$ or $\mathbb{R}_{\mathtt{OfferUp}}$ or $\mathbb{R}_{\mathtt{Receive}}$ or $\mathbb{R}_{\mathtt{Return}}$ or $\mathbb{R}_{\mathtt{ReNego}}$ or $\mathbb{R}_{\mathtt{NewPh}}$ or $\mathbb{R}_{\mathtt{maxPos}}$ during cs, or*
- *$\exists v \in D_a^\gamma, v \in \mathcal{P}(u), \mathtt{tok}_v^\gamma = \bullet$ and u executes $\mathbb{R}_{\mathtt{Lose}}$ or $\mathbb{R}_{\mathtt{NewBit}}$ during cs, or*
- *$\exists v \in D_a^\gamma, v \in \mathcal{P}_{\mathtt{neq}}(u), \mathtt{tok}_v^\gamma \in \{\circlearrowleft, \circ\}$ and u executes $\mathbb{R}_{\mathtt{ReplayD}}$ during cs, or*
- *$\exists v \in D_a^\gamma, v \in \mathcal{P}_{\mathtt{eq}}(u), \mathtt{tok}_v^\gamma = \uparrow$ and u executes $\mathbb{R}_{\mathtt{Fake}}$ or $\mathbb{R}_{\mathtt{ReplayUp}}$ during cs.*

We say that D_a^γ is activated during cs if there exists one node $u \in V$ which activates D_a^γ during cs.

We now prove that the definition of activation matches the evolution of $\mathtt{CD}^o$'s. Namely, if one $\mathtt{CD}^o$ is not activated during one computing step, either it is unchanged, either it disappears, its anchor becoming an intern node of one other $\mathtt{CD}^o$.

Lemma 5.48

Let $cs = \gamma \to \gamma'$ be a computing step, and let $a \in \mathbb{A}(\gamma)$ be an anchor at γ. If D_a^γ is not activated during cs then either $D_a^{\gamma'} = D_a^\gamma$, either $a \notin \mathbb{A}(\gamma')$.

Proof: Let us suppose that D_a^γ is not activated during cs and that $a \in \mathbb{A}(\gamma')$, and let us prove $D_a^{\gamma'} = D_a^\gamma$. To achieve this, let us prove that for all branch $\mathcal{B} = v_1 \cdots v_k$ of G_a, $D_a^{\gamma'}(\mathcal{B}) = D_a^\gamma(\mathcal{B})$.

Remark first that $\forall i \in [1, k]$, $\mathtt{tok}_{v_i}^{\gamma'} = \mathtt{tok}_{v_i}^\gamma$, which implies $D_a^\gamma(\mathcal{B}) \subset D_a^{\gamma'}(\mathcal{B})$. We have already seen in the proof of Lemma 5.18 that if $D_a^{\gamma'}(\mathcal{B})$ is longer than $D_a^\gamma(\mathcal{B})$ then there exists $v \in \mathbb{B}(D_a^\gamma)$ which executes $\mathbb{R}_{\mathtt{Win}}$ during cs, which would activate D_a^γ. Consequently, $D_a^{\gamma'}(\mathcal{B}) = D_a^\gamma(\mathcal{B})$.

This, being true for all branch $\mathcal{B}$ of G_a, proves $D_a^{\gamma'} = D_a^\gamma$.

■

We now prove that the definition of activation captures almost all computing steps of our algorithm. The only situation in which no $\texttt{CD}^o$ is activated is when all activated nodes execute $\mathbb{R}_{\texttt{ReWin}}$, rule which is not considered by $\texttt{W}$, but which cannot be responsible of livelock only by itself.

Lemma 5.49

Let $cs = \gamma \to \gamma'$ be a computing step. There exists an anchor $a \in \mathbb{A}(\gamma)$ such that D_a^γ is activated during cs, or all nodes activated during cs execute $\mathbb{R}_{\texttt{ReWin}}$.

Proof: Let us suppose that there exists at least one node $u \in V$ such that u executes one rule different from $\mathbb{R}_{\texttt{ReWin}}$ during cs.

- If u executes $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs then u is the root, and u activates D_r^γ.
- If u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs then $\exists u \in \mathcal{P}(v) : \texttt{tok}_p^\gamma \notin \{\bot, \uparrow, \downarrow\}$, so according to Lemma 5.8, $\exists a \in \mathbb{A}(\gamma)$ such that $v \in D_a^\gamma$ and thus $u \in \mathbb{B}(D_a^\gamma)$, and u activates D_a^γ.
- If u executes $\mathbb{E}_{\texttt{TrustChild}}$ or $\mathbb{R}_{\texttt{Give}}$ or $\mathbb{R}_{\texttt{NewPlay}}$ or $\mathbb{R}_{\texttt{Drop}}$ or $\mathbb{R}_{\texttt{Nego}}$ or $\mathbb{R}_{\texttt{OfferUp}}$ or $\mathbb{R}_{\texttt{Receive}}$ or $\mathbb{R}_{\texttt{Return}}$ or $\mathbb{R}_{\texttt{ReNego}}$ or $\mathbb{R}_{\texttt{NewPh}}$ or $\mathbb{R}_{\texttt{maxPos}}$ during cs then $\texttt{tok}_u^\gamma \neq \bot$ so according to Lemma 5.8, there exists $a \in \mathbb{A}(\gamma)$ such that $u \in D_a^\gamma$ and thus u activates D_a^γ.
- If u executes $\mathbb{R}_{\texttt{Lose}}$ or $\mathbb{R}_{\texttt{NewBit}}$ during cs then $\texttt{LoseNeg}^\gamma(u) \vee \texttt{AnswerNeg}^\gamma(u)$. In both cases, $\exists p \in \mathcal{P}(u) : \texttt{tok}_p^\gamma = \bullet$, so according to Lemma 5.8, there exists $a \in \mathbb{A}(\gamma)$ such that $v \in D_a^\gamma$ and thus u activates D_a^γ.
- If u executes $\mathbb{R}_{\texttt{ReplayD}}$ during cs then $\texttt{ReplayL}^\gamma(u)$ and thus $\exists p \in \mathcal{P}_{\texttt{neq}}(u) : \texttt{tok}_p^\gamma \in \{\circlearrowleft, \circ\}$. According to Lemma 5.8, there exists $a \in \mathbb{A}(\gamma)$ such that $v \in D_a^\gamma$ and thus u activates D_a^γ.
- If u executes $\mathbb{R}_{\texttt{Fake}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs then $\exists v \in \mathcal{P}_{\texttt{neq}}{}^\gamma(u) : \texttt{tok}_v^\gamma = \uparrow$. According to Lemma 5.8, there exists $a \in \mathbb{A}(\gamma)$ such that $v \in D_a^\gamma$ and thus u activates D_a^γ.

■

We can finally provide a definition of admissibility, which corresponds to our weight function $\texttt{W}$ according to Theorem 5.14.

Definition 5.25 (*Admissible weight function*)

Let $\texttt{W}_1, \ldots, \texttt{W}_k : V \times \Gamma \to M$ be k weight functions on nodes such that $(\texttt{W}_1, \ldots, \texttt{W}_k)$ is pre-admissible. We say that $(\texttt{W}_1, \ldots, \texttt{W}_k)$ is admissible if for all computing step $cs = \gamma \to \gamma'$, for all $\texttt{CD}^o$ $D_a^{\gamma'}$, if D_a^γ is activated during cs and if a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs, then

$$(\texttt{W}_1, \ldots, \texttt{W}_k)(D_a^{\gamma'}, \gamma') \prec^k (\texttt{W}_1, \ldots, \texttt{W}_k)(D_a^\gamma, \gamma)$$

Theorem 5.14

$\texttt{W}$ is admissible.

Proof: We established in Theorem 5.13 that $\texttt{W}$ is pre-admissible. Let us now consider one computing step $\gamma \to \gamma'$, one $\texttt{CD}^o$ $D_a^{\gamma'}$ which is activated during cs, and suppose that a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs.

Let us consider one node u which activates D_a^γ during cs.

- If $u \in \mathbb{B}(D_a^\gamma)$ and u executes $\mathbb{R}_{\texttt{Win}}$ or $\mathbb{R}_{\texttt{FakeWin}}$ during cs, then according to Theorem 5.4, $\texttt{W}(D_a^{\gamma'}, \gamma') \prec \texttt{W}(D_a^\gamma, \gamma)$.
- If $u \in D_a^\gamma$ and u executes $\mathbb{E}_{\texttt{TrustChild}}$ or $\mathbb{R}_{\texttt{Give}}$ or $\mathbb{R}_{\texttt{NewPlay}}$ or $\mathbb{R}_{\texttt{Drop}}$ or $\mathbb{R}_{\texttt{Nego}}$ or $\mathbb{R}_{\texttt{OfferUp}}$ or $\mathbb{R}_{\texttt{Receive}}$ or $\mathbb{R}_{\texttt{Return}}$ or $\mathbb{R}_{\texttt{ReNego}}$ or $\mathbb{R}_{\texttt{NewPh}}$ or $\mathbb{R}_{\texttt{maxPos}}$ during cs, then according to Theorems 5.6, 5.8, and 5.12, $\texttt{W}(D_a^{\gamma'}, \gamma') \prec \texttt{W}(D_a^\gamma, \gamma)$.

- $\exists v \in D_a^\gamma, v \in \mathcal{P}(u), \texttt{tok}_v^\gamma = \bullet$ and u executes $\mathbb{R}_{\texttt{Lose}}$ or $\mathbb{R}_{\texttt{NewBit}}$ during cs, then since $\texttt{OkNeg}^\gamma(u)$, we deduce $v \in \mathcal{P}_{\texttt{neq}}{}^\gamma(v)$, and thus according to Theorems 5.10 and 5.12 $\texttt{W}(D_a^{\gamma'}, \gamma') \prec \texttt{W}(D_a^\gamma, \gamma)$.
- $\exists v \in D_a^\gamma, v \in \mathcal{P}_{\texttt{neq}}(u), \texttt{tok}_v^\gamma \in \{\circlearrowleft, \circ\}$ and u executes $\mathbb{R}_{\texttt{ReplayD}}$ during cs, then according to Theorem 5.12 $\texttt{W}(D_a^{\gamma'}, \gamma') \prec \texttt{W}(D_a^\gamma, \gamma)$.
- $\exists v \in D_a^\gamma, v \in \mathcal{P}_{\texttt{eq}}(u), \texttt{tok}_v^\gamma = \uparrow$ and u executes $\mathbb{R}_{\texttt{Fake}}$ or $\mathbb{R}_{\texttt{ReplayUp}}$ during cs, then according to Theorem 5.12 $\texttt{W}(D_a^{\gamma'}, \gamma') \prec \texttt{W}(D_a^\gamma, \gamma)$. ■

We now prove that if the anchor of a $\texttt{CD}^o$ does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$, then the $\texttt{CD}^o$ can only be activated a finite number of time.

Theorem 5.15

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an infinite execution, and let $a \in V$ such that $\forall t \geq 0, a \in \mathbb{A}(\gamma_t)$, and such that a does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during ϵ.

There only exists a finite number of computing steps $cs_t = \gamma_t \to \gamma_{t+1}$ such that $D_a^{\gamma_t}$ is activated during cs_t.

Proof: According to Theorem 5.14, $\texttt{W}$ is admissible, so if $D_a^{\gamma_t}$ is activated during cs_t, we have $\texttt{W}(D_a^{\gamma_{t+1}}, \gamma_{t+1}) \prec \texttt{W}(D_a^{\gamma_t}, \gamma_t)$.

Let us now reason by contradiction and suppose that there exists an infinity of such computing steps, $cs_{t_0}, cs_{t_1}, \ldots$. We have $\forall i \geq 0, \texttt{W}(D_a^{\gamma_{t_i+1}}, \gamma_{t_i+1}) \prec \texttt{W}(D_a^{\gamma_{t_i}}, \gamma_{t_i})$. Furthermore, according to Theorem 5.13, we also have $\forall i \geq 0, \texttt{W}(D_a^{\gamma_{t_{i+1}}}, \gamma_{t_{i+1}}) \preceq \texttt{W}(D_a^{\gamma_{t_i+1}}, \gamma_{t_i+1})$. Thus, we deduce $\forall i \geq 0, \texttt{W}(D_a^{\gamma_{t_i+1}}, \gamma_{t_{i+1}}) \prec \texttt{W}(D_a^{\gamma_{t_i}}, \gamma_{t_i})$.

Since $\prec$ is a well-founded order on $\texttt{WD}^o$'s, such infinite decreasing sequence does not exist, and thus we raised a contradiction. Consequently, there only exists a finite number of computing steps $cs_t = \gamma_t \to \gamma_{t+1}$ such that $D_a^{\gamma_t}$ is activated during cs_t. ■

The following theorem is crucial, since it states that at some point, the $\texttt{CD}^o$ which are not anchored at the root cease being activated, and cease disappearing too. It is a key theorem to prove the last result of this section, but will also be determining in the following section to prove the fairness of our algorithm.

Theorem 5.16

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an infinite execution. There exists $t_0 \geq 0$ such that for all $t \geq t_0$,

$\mathbb{A}^*(\gamma_t) = \mathbb{A}^*(\gamma_{t_0})$ and $\forall a \in \mathbb{A}^*(\gamma_t), D_a^{\gamma_t}$ is not activated during $\gamma_t \to \gamma_{t+1}$.

Proof: Let us first define t_0, and let us consider $a \in \mathbb{A}^*(\gamma_0)$.

If $\exists t \geq 0 : a \notin \mathbb{A}^*(\gamma_t)$ then let us set $t_a = t$. According to Lemma 5.9, $\forall t \geq t_a, a \notin \mathbb{A}(\gamma_t)$.

Otherwise, we have $\forall t \geq 0, a \in \mathbb{A}^*(\gamma_t)$. According to Theorem 5.15, there exists only a finite number of computing steps $cs_t = \gamma_t \to \gamma_{t+1}$ such that $D_a^{\gamma_t}$ is activated during cs_t (since $a \neq r$, it cannot execute $\mathbb{R}^r_{\texttt{NewDFS}}$). Let us consider t' the highest value of t such that $D_a^{\gamma_t}$ is activated during cs_t, and let us set $t_a = t' + 1$.

Let us now set $t_0 = \max_{a \in \mathbb{A}^*(\gamma_0)} t_a$. Let us consider one anchor $a \in \mathbb{A}^*(\gamma_{t_0})$. We have $\forall t \geq t_0, a \in \mathbb{A}^*(\gamma_t)$, since if not, we would have $t_a > t_0$ which is contradictory with the definition of t_0. Therefore we have $\forall t \geq t_0, \mathbb{A}^*(\gamma_{t_0}) \subseteq \mathbb{A}^*(\gamma_t)$, and according to Lemma 5.9 we conclude $\forall t \geq t_0, \mathbb{A}^*(\gamma_{t_0}) = \mathbb{A}^*(\gamma_t)$.

Furthermore, since $\forall t \geq t_0, a \in \mathbb{A}^*(\gamma_t)$, t_a has been defined as one more than the highest value such that $D_a^{\gamma_t}$ is activated during cs_t. Therefore, $\forall t \geq t_0, D_a^{\gamma_t}$ is not activated during cs_t. ■

We now formally prove that our algorithm satisfies Condition 2 of Specification 5.1.

Theorem 5.17

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be a maximal execution, and let r be the root of G. ϵ is infinite, and can be divided into an infinite number of circulation rounds.

Proof: According to Theorem 5.1, we only have to prove that there exists an infinite number of computing steps $cs_i = \gamma_i \to \gamma_{i+1}$ such that r executes $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs_i.

According to Theorem 5.16, there exists $t_0 \geq 0$ such that $\forall a \in \mathbb{A}(\gamma_t) \setminus \{r\}, \forall t' \geq t_0, a \in \mathbb{A}(\gamma_{t'})$ and $D_a^{\gamma_{t'}}$ is not activated during $\gamma_{t'} \to \gamma_{t'+1}$.

Let us reason by contradiction and suppose that there only exists a finite number of computing steps $cs_i = \gamma_i \to \gamma_{i+1}$ such that r executes $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs_i. Then, there exists $t_1 \geq 0$ such that $\forall t \geq t_1, r$ does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs_t. Thus, according to Theorem 5.15, there exists $t_2 \geq t_1$ such that $\forall t \geq t_2, D_r^{\gamma_t}$ is not activated during cs_t.

Let us consider $t_3 = \max(t_0, t_2)$, and the infinite execution $\epsilon_3 = \gamma_{t_3} \to \gamma_{t_3+1} \to \cdots$. By definition, no $\texttt{CD}^o$ is activated during ϵ_3. Thus, during ϵ_3, nodes only execute $\mathbb{R}_{\texttt{ReWin}}$. But once a node v executes $\mathbb{R}_{\texttt{ReWin}}$, it cannot execute it again before the execution of one other rule sets $\texttt{play}_v = \texttt{F}$. Thus, there does not exist such execution, so we reached a contradiction.

Consequently, there exists an infinite number of computing steps $cs_i = \gamma_i \to \gamma_{i+1}$ such that r executes $\mathbb{R}^r_{\texttt{NewDFS}}$ during cs_i. ■

5.4.3 Convergence

In this section, we establish that our algorithm behaves correctly, by largely using results established in Section 5.4.2, and especially in Section 5.4.2.9. We use in particular Theorems 5.16 and 5.17, that allow us to start the reasoning at a moment where the algorithm has already partly converged. Namely, we consider executions in which the set of all the anchors remain constant, and such that all the $\texttt{CD}^o$'s which are not anchored at the root of the $\texttt{D}^o$ are not activated.

Once this is considered, the proof is organized according to the following scheme. We first establish some additional stability properties on the $\texttt{CD}^o$ anchored at the root of the $\texttt{D}^o$. Then, we give a formal definition of the notion of having the token, for one node. The formal notion we use to describe the part of an execution where one node is involved in the token circulation is *circulation round*, introduced in Theorem 5.1 and Definition 5.27. After that, we prove that if one node executes a circulation round, then all of its children which might take the token (depending on their variables `play` and `c`) actually do it too, and execute a circulation round as well. We can thus reason by induction an prove that, starting at the root, the token browses a part of the $\texttt{D}^o$. Finally we prove that the alternation of the color of the root, which occurs each time it executes $\mathbb{R}^r_{\texttt{NewDFS}}$, associated to properties of the algorithm, especially its effect on the variables `play` of the children of a node, guarantee that the part of the $\texttt{D}^o$ which is being browsed by the token grows each time the root executes $\mathbb{R}^r_{\texttt{NewDFS}}$, and eventually become the entire $\texttt{D}^o$.

As we said before, we consider executions which start after the value of t_0 guaranteed by Theorem 5.16. In particular, the set of anchors does not evolve during the executions we consider, and according to Lemma 5.48, no $\texttt{CD}^o$ other than the one anchored at r will be updated. Consequently, to enhance readability, we allow ourselves to simply write $\mathbb{A}$ (resp. $\mathbb{A}^*$) without referring to the configuration in which this set is evaluated, and to simply write D_a when $a \in \mathbb{A}^*$ without referring to the configuration in which we consider the $\texttt{CD}^o$ anchored at a.

Lemma 5.50

Let ϵ be a maximal execution. We have $\forall t \geq 0, \forall a \in \mathbb{A}^*, D_r^{\gamma_t} \cap D_a = \emptyset$.

Proof: Let us reason by contradiction and suppose that $\exists v \in D_r^{\gamma_t} \cap D_a$. Remark first that since $v \in D_a$ and $a \neq r$, we have $v \neq r$.

Since $v \in D_r^{\gamma_t}$, there exists $\mathcal{B} = v_1 \cdots v_k$ a branch of $D_r^{\gamma_t}$ such that $v_1 = r$ and $v_k = v$. According to Theorem 5.17, there exists $t' \geq t$ such that r executes $\mathbb{R}^r_{\texttt{NewDFS}}$ during $\gamma_{t'} \to \gamma_{t'+1}$. By definition, at $\gamma_{t'}$, only r is a branch of $D_r^{\gamma_{t'}}$, and in particular, $\mathcal{B}$ is not. Let us consider t_0 the smallest value greater than t such that $\mathcal{B}$ is a branch of $D_r^{\gamma_{t_0}}$ and $\mathcal{B}$ is not a branch of $D_r^{\gamma_{t_0+1}}$. According to Lemma 5.10, we deduce that v executes $\mathbb{R}_{\texttt{Return}}$ during $\gamma_{t_0} \to \gamma_{t_0+1}$.

By definition, we deduce that v activates D_a during $\gamma_{t_0} \to \gamma_{t_0+1}$, which is contradictory. ■

Definition 5.26 (*branch*-CD^o)

Let D_a^γ be a CD^o*. We say that D_a^γ is a branch-*CD^o *if the vertices of D_a^γ are $\{v_1, v_2, \ldots, v_k\}$ with $v_1 = a$ and $\forall i \in [2,k], v_i \in \mathcal{C}_{D_a^\gamma}(v_{i-1})$ and $\forall i \in [2,k], \texttt{c}_{v_i} = \texttt{c}_a$.*

*We say that a branch-*CD^o *is clean if $\forall i \in [1,k], \neg\texttt{Er}^\gamma(v_i)$, and if $\texttt{tok}^\gamma_{v_i} = \downdownarrows$, then $|\texttt{Players}^\gamma(v_i)| + |\{u \in \mathcal{C}^\gamma(v_i) : \texttt{tok}^\gamma_u = \bigstar\}| \leq 1$.*

Lemma 5.51

*Let D_a^γ be a clean branch-*CD^o*, and let us label its vertices $v_1, v_2 \ldots v_k$ such that $\forall i \in [2,k], v_i \in \mathcal{C}_{D_a^\gamma}(v_{i-1})$.*

$\forall i \in [1, k-2], \texttt{tok}_{v_i} = \downarrow$, and if $\texttt{tok}_{v_{k-1}} \neq \downarrow$ then $(\texttt{tok}_{v_{k-1}}, \texttt{tok}_{v_k}) \in \{(\downdownarrows, \bigstar), (\circlearrowleft, \bigstar), (\circ, \uparrow)\}$.

Proof: This is a direct consequence of the definition of predicate Er, since nodes are not in error and form a branch in G. ■

Lemma 5.52

*Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an execution such that r execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during $\gamma_0 \to \gamma_1$. Then $\forall t$, $D_r^{\gamma_t}$ is a clean branch-*CD^o*.*

Proof: We reason by induction. The base case is true since r executes $\mathbb{R}^r_{\texttt{NewDFS}}$ in $\gamma_0 \to \gamma_1$, thus $\texttt{tok}_r^{\gamma_0} = \uparrow$ and thus $D_r^{\gamma_0} = \{r\}$, which is a clean branch-CD^o. Let us also prove that $D_r^{\gamma_1}$ is a clean branch-CD^o. We have $\texttt{tok}_r^{\gamma_1} = \bigstar$. Let us prove by contradiction that $D_r^{\gamma_1} = \{r\}$, and suppose r has one child u with $\texttt{tok}_u^{\gamma_1} \neq \bot$. Since $u \notin D_r^{\gamma_0}$, u is not activated during $\gamma_0 \to \gamma_1$, and thus $\texttt{tok}_u^{\gamma_0} \neq \bot$. According to Lemma 5.8, $\exists a \in \mathbb{A}^* : u \in D_a$. As a consequence, $u \in D_a \cap D_r^{\gamma_1}$ which cannot be according to Lemma 5.50, and therefore $D_r^{\gamma_1}$ is a clean branch-CD^o.

Let us now suppose that the property holds at γ_t, and prove it at γ_{t+1}. Remark first that if no node u joins D_r by executing $\mathbb{R}_{\texttt{Win}}$, then the guards of the rules that update tok guarantee that no error can be generated.

Furthermore, following the same idea as what was presented for $D_r^{\gamma_1}$, a node u can join D_r only if it has a parent v in D_r such that $\texttt{tok}_v = \downdownarrows$, otherwise u would be in the intersection of D_r and one other CD^o.

Therefore, we only consider the different cases which imply $\downdownarrows$, the others being trivial. Let us consider v_k the last node of the clean branch-CD^o $D_r^{\gamma_t}$.

- If $\texttt{tok}^\gamma_{v_k} = \downdownarrows$ then $\forall i \in [1, k-1], \texttt{tok}_{v_i} = \downarrow$ and are not enabled. If the player of v, u, executes $\mathbb{R}_{\texttt{Win}}$, then $\texttt{c}_u^{\gamma_{t+1}} = \texttt{c}_v^{\gamma_t} = \texttt{c}_r^{\gamma_{t+1}}$ and therefore $D_r^{\gamma_{t+1}}$ is a branch-CD^o.

  Furthermore, no node reaches $\texttt{Players}^\gamma(v_k)$ since $\mathbb{R}_{\texttt{ReplayUp}}$ implies an activation of another CD^o, and $\mathbb{R}_{\texttt{ReplayD}}$ is not enabled due to $\neg$ReplayL. Therefore,

  $\texttt{Players}^{\gamma_{t+1}} \cup \{u \in \mathcal{C}(v_k) : \texttt{tok}_u^{\gamma_{t+1}} = \bigstar\} \subseteq \texttt{Players}^{\gamma_t} \cup \{u \in \mathcal{C}(v_k) : \texttt{tok}_u^{\gamma_t} = \bigstar\}$, and thus the branch-CD^o is clean.

- If $\texttt{tok}_{v_k}^{\gamma} = \bigstar \wedge \texttt{tok}_{v_{k-1}}^{\gamma} = \downdownarrows$ then $\forall i \in [1, k-2], \texttt{tok}_{v_i} = \downarrow$, and are not enabled, neither is v_k. Furthermore, $\texttt{Players}^{\gamma}(v_{k-1}) = \emptyset$ and no node reaches $\texttt{Players}^{\gamma}(v_{k-1})$ for the same reason as previously. Therefore, $D_r^{\gamma_{t+1}}$ is a clean branch-$\texttt{CD}^o$.
- If $\texttt{tok}_v^{\gamma} = \bullet$ then $\forall i \in [1, k-1], \texttt{tok}_{v_i} = \downarrow$ and are not enabled. Children u of v_k can only execute rules that update $\texttt{play}_u$ from $\texttt{P}$ to $\{\texttt{L}, \texttt{W}, \texttt{F}\}$ and thus $\texttt{Players}^{\gamma'}(v) \subseteq \texttt{Players}^{\gamma}(v)$. If v executes $\mathbb{R}_{\texttt{OfferUp}}$ then nothing has to be proven. If v executes $\mathbb{R}_{\texttt{Give}}$ then $|\texttt{Players}^{\gamma_t}(v)| + |\{u \in \mathcal{C}^{\gamma}(v) : \texttt{tok}_u^{\gamma} = \bigstar\}| \leq 1$, and thus $D_r^{\gamma_{t+1}}$ is a clean branch-$\texttt{CD}^o$. ■

Corollary 5.2

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots$ be an execution such that r execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during $\gamma_0 \to \gamma_1$. Then no node executes $\mathbb{R}_{\texttt{FakeWin}}$ during ϵ.

Proof: Let us consider $cs = \gamma \to \gamma'$ one computing step of ϵ, and let us consider one node v such that $\texttt{tok}_v^{\gamma} = \perp$ and $\texttt{play}_v^{\gamma} = \texttt{P}$.

Suppose first that $\forall p \in \mathcal{P}(v)$ such that $\texttt{tok}_p^{\gamma} \notin \{\perp, \uparrow\}$ we have $p \in D_r^{\gamma}$. According to Lemmas 5.52 and 5.51, there exists at most 2 parents of v such that $\texttt{tok}_p^{\gamma} \neq \downarrow$, and if there are 2 then we have $\texttt{tok}_{p_1} = \circlearrowleft$ and $\texttt{tok}_{p_2} = \bigstar$. In any case, we have $\texttt{OkNeg}^{\gamma}(v)$ and thus v does not execute $\mathbb{R}_{\texttt{FakeWin}}$ during cs.

Let us now suppose that $\exists p \in \mathcal{P}(v)$ such that $\texttt{tok}_p^{\gamma} \notin \{\perp, \uparrow\}$ and $p \notin D_r^{\gamma}$. Since $\texttt{tok}_p \neq \perp$, according to Lemma 5.8 there exists $a \in \mathbb{A}^*$ such that $p \in D_a$. But then, since $v \in \mathbb{B}(D_a)$, v does not execute $\mathbb{R}_{\texttt{FakeWin}}$ since it would activate D_a. ■

From now on, we suppose that the executions start after at least one $\mathbb{R}^r_{\texttt{NewDFS}}$ and thus we suppose that no node executes $\mathbb{R}_{\texttt{FakeWin}}$ during the executions.

We presented in Definition 5.13 and in Theorem 5.1 the notion of circulation rounds for the root. Hence, for the sake of the proof, we also introduce the notion of circulation round on nodes that are not the root. This will allow us to consider circulation round on any node $v \in V$, root or not. Namely, this corresponds to the part of a circulation round in which one particular node holds the token, or in which the token is in one of its descendants.

Definition 5.27 (*Circulation Round for $v \neq r$*)

Let $\epsilon = \gamma_0 \to \cdots \to \gamma_k$ be a finite execution.

We say that $v \neq r$ executes a circulation round during ϵ if v executes $\mathbb{R}_{\texttt{Win}}$ and $\mathbb{R}_{\texttt{Return}}$, exactly once during ϵ, in that order.

We call circulation round of v during ϵ, and denote $\epsilon_{rt}(v, \epsilon)$ the subexecution of $\epsilon : \gamma_{i+1} \to \cdots \to \gamma_j$ where $\gamma_i \to \gamma_{i+1}$ (resp. $\gamma_j \to \gamma_{j+1}$) is the computing step of ϵ where r executes $\mathbb{R}_{\texttt{Win}}$ (resp. $\mathbb{R}_{\texttt{Return}}$).

Lemma 5.53

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_k$ be a circulation round of v. Then

1. $\forall i \in [0, k], \texttt{c}_v^{\gamma_i} = \texttt{c}_v^{\gamma_0}$
2. $\forall u \in \mathcal{C}(v), \texttt{tok}_u^{\gamma_0} = \perp$
3. $\texttt{tok}_v^{\gamma_k} = \uparrow$ *and* $\texttt{MayDropUp}^{\gamma_k}(v)$.

Proof: Let us prove the three items separately.

1. If $v = r$ then v updates $\texttt{c}_v$ only with the action of $\mathbb{R}^r_{\texttt{NewDFS}}$, and it does not execute $\mathbb{R}^r_{\texttt{NewDFS}}$ during ϵ by definition.

If $v \neq r$ then v updates $\mathtt{c}_v$ only with the action of $\mathbb{R}_{\texttt{FakeWin}}$, which is not executed according to Corollary 5.2, or with the action of $\mathbb{R}_{\texttt{Win}}$, which is not executed by v during ϵ by definition.

2. Since v executes a circulation round, it is activated and thus belongs to D_r. According to Lemma 5.52, $\neg\texttt{Er}^{\gamma_0}(v)$. Furthermore, by definition of circulation round, we have $\texttt{tok}_v^{\gamma_0} \in \{\bullet, \bigstar\}$. Thus, $\forall u \in \mathcal{C}(v), \texttt{tok}_u^{\gamma_0} = \perp$.
3. By definition of a circulation round, v executes $\mathbb{R}^r_{\texttt{NewDFS}}$ or $\mathbb{R}_{\texttt{Return}}$ during $\gamma_k \to \gamma_{k+1}$ and thus $\texttt{tok}_v^{\gamma_k} = \uparrow$ and $\texttt{MayDropUp}^{\gamma_k}(v)$. ■

In the following lemma, we establish that if one node v executes a circulation round, then its playing children also execute a circulation round. This will allow us in the latter to reason by induction.

Lemma 5.54

Let us consider one node $v \in V$, and let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_k$ be a circulation round of v. Then $\forall u \in \texttt{Candidates}^{\gamma_0}(v), u$ executes a circulation round during ϵ.

Proof: Let us consider $u \in \texttt{Candidates}^{\gamma_0}(v)$. According to Lemma 5.53, $\forall i \in [0,k], \mathtt{c}_v^{\gamma_i} = \mathtt{c}_v^{\gamma_0}$ Since $\mathtt{c}_u^{\gamma_0} \neq \mathtt{c}_v^{\gamma_0}$, we deduce that $\forall i \in [0,k], \mathtt{c}_v^{\gamma_i} \neq \mathtt{c}_u^{\gamma_0}$. Lemma 5.53 also assures that $\texttt{tok}_u^{\gamma_0} = \perp$ which implies that, during ϵ, u cannot execute $\mathbb{R}_{\texttt{Return}}$ before it executes $\mathbb{R}_{\texttt{Win}}$. Let us first prove that u executes $\mathbb{R}_{\texttt{Win}}$ during ϵ.

Whether $v = r$ or $v \neq r$ has no incidence on the fact that, necessarily, v executes $\mathbb{R}_{\texttt{OfferUp}}$ during ϵ. Otherwise, we would have $\texttt{tok}_v^{\gamma_k} \notin \{\perp, \uparrow\}$ which is inconsistent. Let us name $cs_s = \gamma_s \to \gamma_{s+1}$ the computing step during which v executes $\mathbb{R}_{\texttt{OfferUp}}$. We deduce that $\mathtt{c}_u^{\gamma_s} = \mathtt{c}_v^{\gamma_s} (\neq \mathtt{c}_u^{\gamma_0}) \vee \texttt{play}_u^{\gamma_s} \in \{\texttt{W}, \texttt{F}\}$. Since we consider executions where nodes do not execute $\mathbb{R}_{\texttt{FakeWin}}$, $\mathbb{R}_{\texttt{Win}}$ is the only rule whereby u can update $\mathtt{c}_u$, and also the only rule whereby u can update $\texttt{play}_u$ from $\{\texttt{P}, \texttt{L}\}$ to $\{\texttt{W}, \texttt{F}\}$. We conclude that u executes $\mathbb{R}_{\texttt{Win}}$ during $\gamma_0 \to \cdots \to \gamma_s$, let us say during $cs_t = \gamma_t \to \gamma_{t+1}$.

Let us now prove that u executes $\mathbb{R}_{\texttt{Win}}$ only once during ϵ. Let us reason by contradiction and suppose there exists a second computing step $cs = \gamma \to \gamma'$ where u executes $\mathbb{R}_{\texttt{Win}}$. Since D_r^γ is a branch-$\texttt{CD}^o$, then $\forall w \in D_r^\gamma, \mathtt{c}_w^\gamma = \mathtt{c}_v^\gamma = \mathtt{c}_u^\gamma$. Indeed, u does not change its color between the first and the second execution of $\mathbb{R}_{\texttt{Win}}$ by definition. But because u executes $\mathbb{R}_{\texttt{Win}}$, $\exists p \in \mathcal{P}_{\texttt{neq}}{}^\gamma(u) : \texttt{tok}_p^\gamma = \downarrow\downarrow$. This parent is necessarily in one other $\texttt{CD}^o$, which is activated by u during cs, which is contradictory.

Since v executes $\mathbb{R}_{\texttt{OfferUp}}$ during cs_s, we have $\texttt{tok}_v^{\gamma_s} = \bullet \wedge \neg\texttt{Er}^{\gamma_s}(v)$ which implies that $\texttt{tok}_u^{\gamma_s} = \perp$, and because u executes $\mathbb{R}_{\texttt{Win}}$ during $\gamma_0 \to \cdots \to \gamma_a$, it means that u executes $\mathbb{R}_{\texttt{Return}}$ after having executed $\mathbb{R}_{\texttt{Win}}$ and before γ_s.

Consequently, u executes one circulation round during ϵ. ■

We now introduce the notion of Free DODAG. This notion is defined only on reset points, *i.e.* configuration immediately subsequent to an execution of $\mathbb{R}^r_{\texttt{NewDFS}}$ (see Definition 5.13). A Free DODAG corresponds to all the nodes which can reach the root by a path in G, such that no node of this path has any parent in one other $\texttt{CD}^o$. It corresponds to nodes which have the possibility to receive the token sent from the root by executing $\mathbb{R}_{\texttt{Win}}$ (we do not take into account the color of the nodes on this path in this definition).

Definition 5.28 (*Free DODAG*)

Let γ be a reset point of an execution ϵ. We call free DODAG in γ, and denote $\texttt{FD}^o(\gamma)$, the smallest set that contains r and such that $\forall w \neq r$:

$$w \in \texttt{FD}^o(\gamma) \iff \texttt{tok}_w = \perp \wedge \exists p \in \mathcal{P}(w) : p \in \texttt{FD}^o(\gamma) \wedge \forall p \in \mathcal{P}(w), (p = r \vee \texttt{tok}_p = \perp)$$

Remark 5.9

$\texttt{FD}^o(\gamma)$ *is a sub-*$\texttt{D}^o$ *of* G *anchored at* r*:* $\forall u, v \in \texttt{FD}^o(\gamma), u \in \mathcal{C}_{\texttt{FD}^o(\gamma)}(v) \iff u \in \mathcal{C}_G(v)$

Lemma 5.55

Let γ^1 *and* γ^2 *be two reset points of an execution. We have* $\texttt{FD}^o(\gamma^1) = \texttt{FD}^o(\gamma^2)$.

Proof: The set of nodes v such that $\texttt{tok}_v^{\gamma^1} \neq \bot$ is the union of $D_r^{\gamma^1} = \{r\}$ and of all $\texttt{CD}^o$'s $D_a^{\gamma^1}, a \in \mathbb{A}(\gamma^1)$. By hypothesis, the $\texttt{CD}^o$'s anchored to a node of $\mathbb{A}$ are not activated during the execution, and thus the set of nodes v such that $\texttt{tok}_v \neq \bot$ is the same in γ^1 as in γ^2. As a consequence, $\texttt{FD}^o(\gamma^1) = \texttt{FD}^o(\gamma^2)$. ■

$\texttt{FD}^o$ corresponds to all the nodes that may receive the token from the root, without considering variables `c` and `play`. In order to gain in precision, we define the set of all nodes which will actually receive the token during the starting circulation round.

Definition 5.29 (*Reachable area from r*)

Let γ *be a reset point. We define the reachable area from* r *at* γ*, and denote* $\texttt{Reach}_r(\gamma)$*:*

$$\texttt{Reach}_r(\gamma) = \{v_k \mid \exists v_1 \cdots v_k \in \texttt{FD}^o(\gamma) : \left\{ \begin{array}{l} v_1 = r \wedge \forall i \geq 2, v_i \in \mathcal{C}(v_{i-1}) \\ \forall i \geq 2, (\texttt{play}_{v_i}^{\gamma} \in \{\texttt{P}, \texttt{L}\} \wedge \texttt{c}_{v_i}^{\gamma} \neq \texttt{c}_r^{\gamma}) \end{array} \right. \}$$

We prove in the following Lemma that nodes in the Reachable area actually receive the token, and execute exactly one circulation round during the circulation round of the root.

Lemma 5.56

Let $\epsilon = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_k$ *be a circulation round of* r*. Then* $\forall u \in \texttt{Reach}_r(\gamma)$*,* u *executes exactly one circulation round during* ϵ*.*

Proof: Let us prove that lemma by induction on the depth of node u in $\texttt{Reach}_r(\gamma)$. The base case comes immediately, since r is the only node with depth 0 and by definition, r executes a circulation round during ϵ.

Let us now establish the induction step and suppose that for any node u with depth less that d of $\texttt{Reach}_r(\gamma)$, u executes a circulation round during ϵ. Let us consider one node $v \in \texttt{Reach}_r(\gamma)$ with depth $d+1$, and one of its ancestors $u \in \texttt{Reach}_r(\gamma)$ with depth d. By hypothesis, u executes a circulation round during ϵ. Let us consider $\epsilon_{rt}(u, \epsilon) = \gamma_i \to \cdots \to \gamma_j$ the circulation round of u during the circulation round of r. If $v \in \texttt{Candidates}^{\gamma_i}(u)$ then we can apply Lemma 5.54 and thus v executes a circulation round during $\epsilon_{rt}(u, \epsilon)$. If $v \notin \texttt{Candidates}^{\gamma_i}(u)$ then $\texttt{play}_v^{\gamma_i} \notin \{\texttt{P}, \texttt{L}\} \vee \texttt{c}_v^{\gamma_i} \neq \texttt{c}_u^{\gamma_i}$. This is possible only if, before γ_i v executes $\mathbb{R}_{\texttt{Win}}$. According to Lemma 5.53, $\texttt{tok}_v^{\gamma_i} = \bot$. Thus v executes at least one circulation round during ϵ.

Furthermore, v cannot execute several circulation round during ϵ, since v cannot activate any other $\texttt{CD}^o$ than D_r^{γ}, and after its first circulation round it has the same color as the root, and thus the same color as any node of the branch-$\texttt{CD}^o$. ■

The token does not only circulate through the reachable area of r indefinitely, it also makes its reachable area increase at each circulation round. Indeed, suppose one node v has the token, and has one child u with the same color as v. Then u is already seen as visited, which means that it won't receive the token, as being out of the reachable area. Yet, when sending the token upwards to its parent, v forces u to reset $\texttt{play}_u$ to `P`, which means that in the next circulation round, when the color of the token has switched, u will be able to receive the token. Note that this only works if $u \in \texttt{FD}^o(\gamma)$, otherwise u has other parents,

and therefore updates $\texttt{play}_u$ to F. Also note that if one child u does not receive the token for its variable play is already set to W or F, but had the appropriate color to receive it, then u will have to wait two circulation rounds before it actually receives the token. To embrace this behavior, we define the expansion areas of r, which constitutes all nodes in $\texttt{FD}^o(\gamma)$ which have a parent in $\texttt{Reach}_r(\gamma)$ without being in $\texttt{Reach}_r(\gamma)$ themselves.

Definition 5.30 (*Expansion areas of r*)

Let γ be a reset point. We define the expansion area of order 1 and of order 2 of r at γ, and denote $\texttt{Exp}_r^1(\gamma)$ and $\texttt{Exp}_r^2(\gamma)$:

$$\texttt{Exp}_r^1(\gamma) = \{u \in \texttt{FD}^o(\gamma) \mid \exists p \in \mathcal{P}(u) : p \in \texttt{Reach}_r(\gamma) \wedge \texttt{c}_u^\gamma = \texttt{c}_r^\gamma\}$$

$$\texttt{Exp}_r^2(\gamma) = \{u \in \texttt{FD}^o(\gamma) \mid \exists p \in \mathcal{P}(u) : p \in \texttt{Reach}_r(\gamma) \wedge \left\{ \begin{array}{l} \texttt{play}_u^\gamma \in \{\texttt{W},\texttt{F}\} \\ \texttt{c}_u^\gamma \neq \texttt{c}_r^\gamma \end{array} \right. \}$$

The following Lemma proves that the three sets defined above are jointly increasing, as intended. This will allow us to prove that at some point, the reachable area contains the entire $\texttt{FD}^o$.

Lemma 5.57

Let γ^1 and γ^2 be two consecutive reset points of an execution, and let $\epsilon = \gamma_0 \to \cdots \to \gamma_k$ be the circulation round of r between γ^1 and γ^2.

1. $\texttt{Reach}_r(\gamma^1) \subset \texttt{Reach}_r(\gamma^2)$
2. $\texttt{Exp}_r^1(\gamma^1) \subset \texttt{Reach}_r(\gamma^2)$
3. $\texttt{Exp}_r^2(\gamma^1) \subset \texttt{Exp}_r^1(\gamma^2)$

Proof: 1. Remark that $r \in \texttt{Reach}_r(\gamma^2)$ by definition. Let us consider one node $u \in \texttt{Reach}_r(\gamma^1), u \neq r$, and prove that $\texttt{play}_u^{\gamma^2} \in \{\texttt{P},\texttt{L}\} \wedge \texttt{c}_u^{\gamma^2} \neq \texttt{c}_r^{\gamma^2}$.

According to Lemma 5.56, u executes one circulation round during ϵ. After ϵ, u cannot execute $\mathbb{R}_{\texttt{Win}}$ after, at least, $\texttt{tok}_r = \downdownarrows$ which occurs after γ^2. Thus, in γ^2, u still has the color it took when reaching D_r during ϵ Thus, $\texttt{c}_u^{\gamma^2} = \texttt{c}_r^{\gamma^1} \neq \texttt{c}_r^{\gamma^2}$

By definition of $\texttt{Reach}_r$, $\exists v \in \mathcal{P}(u)$ such that $v \in \texttt{Reach}_r(\gamma^1)$. According to Lemma 5.56, v executes a circulation round during ϵ, and thus v executes $\mathbb{R}_{\texttt{OfferUp}}$ during ϵ. Let us consider $cs_i = \gamma_i \to \gamma_{i+1}$ the last computing step of ϵ such that one parent v of u executes $\mathbb{R}_{\texttt{OfferUp}}$. At γ_i, u terminated its circulation round. Indeed, if at γ_i, u has not begun its circulation round, then $\texttt{c}_u^{\gamma_i} \neq \texttt{c}_v^{\gamma_i}$ and $\texttt{play}_u^{\gamma_i} \in \{\texttt{P},\texttt{L}\}$ which would prevent v to execute $\mathbb{R}_{\texttt{OfferUp}}$. Furthermore, since $\neg\texttt{Er}^\gamma(v)$, we have $\texttt{tok}_u^{\gamma_i} = \bot$ so u terminated its circulation round at γ_i.

Consequently, we have $\texttt{c}_u^{\gamma_i} = \texttt{c}_r^{\gamma_i}$ and since $\neg\texttt{WaitSib}^{\gamma_i}(v)$, we also have $\texttt{play}_u^{\gamma_i} \neq \texttt{F}$. Furthermore, according to Lemma 5.50, we know that u has no parent such that $\texttt{tok}_p^{\gamma_i} \notin \{\bot,\uparrow\}$ apart from $D_r^{\gamma_i}$, which implies that u has only one parent such that $\texttt{tok}_p^{\gamma_i} \notin \{\bot,\uparrow\}$, which is v. Consequently, since v is the last parent of u that executes a circulation round, then from γ_{i+1} to the end of ϵ, all the parents of u are such that $\texttt{tok}_p^{\gamma} \in \{\bot,\uparrow\}$. Thus, during that part of the execution, we have constantly $\neg\texttt{FakeReplay}(u)$, and thus for that entire part of the execution, $\texttt{play}_u \neq \texttt{F}$.

Let us now consider γ_j the last configuration that is part of the circulation round of v. According to Lemma 5.53 we have $\texttt{MayDropUp}^{\gamma_j}(v)$, so $\texttt{play}_u^{\gamma_j} \neq \texttt{W}$ and thus $\texttt{play}_u^{\gamma_j} \in \{\texttt{P},\texttt{L}\}$. Since v does not execute $\mathbb{R}_{\texttt{Win}}$ nor $\mathbb{R}_{\texttt{FakeWin}}$ after its first circulation round of ϵ, we deduce $\texttt{play}_u^{\gamma^2} \in \{\texttt{P},\texttt{L}\}$

Since this is true for all the nodes of $\texttt{Reach}_r(\gamma^1)$, we can conclude $\texttt{Reach}_r(\gamma^1) \subseteq \texttt{Reach}_r(\gamma^2)$

2. Let us consider one node $u \in \texttt{Exp}_r^1(\gamma^1)$. By definition of $\texttt{Exp}_r^1$, there exists $p \in \mathcal{P}(v)$: $p \in \texttt{Reach}_r(\gamma^1)$. According to what we established above, $p \in \texttt{Reach}_r(\gamma^2)$, so we only have to prove that $\texttt{c}_u^{\gamma^2} \neq \texttt{c}_r^{\gamma^2} \wedge \texttt{play}_u^{\gamma^2} \in \{\texttt{P}, \texttt{L}\}$ to establish that $u \in \texttt{Reach}_r(\gamma^2)$.
   By definition, we have $\texttt{c}_u^{\gamma_0} = \texttt{c}_r^{\gamma_0}$. Since r does not update c_r during ϵ, and since D_r is a branch-$\texttt{CD}^o$ along ϵ, then until u executes $\mathbb{R}_{\texttt{Win}}$, it has the same color as all the nodes of D_r^γ during ϵ. Since, on the other hand, u cannot activate any other $\texttt{CD}^o$, we deduce that u actually does not execute $\mathbb{R}_{\texttt{Win}}$ during ϵ, neither during $\gamma_k \to \gamma^2$ for the same reason. Consequently, we have $\texttt{c}_u^{\gamma^2} \neq \texttt{c}_r^{\gamma^2}$.
   Now, the proof we made for the previous case totally applies: we can select the last computing step such that one parent p of u executes $\mathbb{R}_{\texttt{OfferUp}}$, at that moment we have $\texttt{c}_u = \texttt{c}_p$, thus we can infer from $\texttt{WaitSib}(v)$ that $\texttt{play}_u \neq \texttt{F}$ and from the non-activation of other $\texttt{CD}^o$'s that this remains true until at least γ^2. FInally, we can reason about $\texttt{MayDropUp}$ as well to establish that $\texttt{play}_u^{\gamma^2} \in \{\texttt{P}, \texttt{L}\}$ which terminates the proof.
3. Let us consider one node $u \in \texttt{Exp}_r^2(\gamma^1)$. By definition of $\texttt{Exp}_r^2$, there exists $p \in \mathcal{P}(v)$: $p \in \texttt{Reach}_r(\gamma^1)$. According to what we established above, $p \in \texttt{Reach}_r(\gamma^2)$, so we only have to prove that $\texttt{c}_u^{\gamma^2} = \texttt{c}_r^{\gamma^2}$ to establish that $u \in \texttt{Reach}_r(\gamma^2)$.
   By definition, we have $\texttt{c}_u^{\gamma_0} \neq \texttt{c}_r^{\gamma_0} \wedge \texttt{play}_u^{\gamma_0} \in \{\texttt{W}, \texttt{F}\}$. Thus, u cannot execute $\mathbb{R}_{\texttt{Win}}$ before it executes $\mathbb{R}_{\texttt{ReplayUp}}$. Since u does not activate any other $\texttt{CD}^o$ than D_r, u executes $\mathbb{R}_{\texttt{ReplayUp}}$ only if it has one parent $p \in D_r$ such that $\texttt{tok}_p = \uparrow$ and $\texttt{c}_p = \texttt{c}_v$. But this cannot happen in ϵ before u executes $\mathbb{R}_{\texttt{Win}}$, since according to Lemma 5.52, all the nodes of D_r have the same color $\texttt{c}_r^{\gamma_0}$, which is the opposite that $\texttt{c}_u$ until it executes $\mathbb{R}_{\texttt{Win}}$. We just found a loop of dependencies.
   We prove that u cannot update $\texttt{c}_u$ before at least γ^2, and thus $u \in \texttt{Exp}^1(\gamma^2)$. ■

The two following lemmas prove that the sets of reachable and expansion areas of r allow a permanent growth of $\texttt{Reach}$, which stops only when $\texttt{Reach}$ includes all the nodes of $\texttt{FD}^o$.

Lemma 5.58

Let γ be a reset point of an execution.
$\big(\texttt{Reach}_r(\gamma) = \texttt{FD}^o(\gamma)\big) \vee \big(\texttt{Exp}_r^1(\gamma) \cup \texttt{Exp}_r^2(\gamma) \neq \emptyset\big)$

Proof: By contradiction, suppose $\texttt{Reach}_r(\gamma) \neq \texttt{FD}^o(\gamma)$, and let us consider one node $u \in \texttt{FD}^o(\gamma) \setminus \texttt{Reach}_r(\gamma)$ with minimal depth.
   This node has a parent in $\texttt{FD}^o(\gamma) \cap \texttt{Reach}_r(\gamma)$ (otherwise it would not be minimal in depth). Thus, u belongs to $\texttt{Exp}_r^1(\gamma) \cup \texttt{Exp}_r^2(\gamma)$. ■

Lemma 5.59

There exists a reset point γ_f such that $\texttt{Reach}_r(\gamma_f) = \texttt{FD}^o(\gamma_f)$.

Proof: Let us consider $\gamma_1, \gamma_2, \cdots$ an infinite sequence of reset points. According to Lemma 5.58, we know that if $\texttt{Reach}_r(\gamma_i) \neq \texttt{FD}^o(\gamma_i)$ then $\texttt{Exp}_r^1(\gamma_i) \cup \texttt{Exp}_r^2(\gamma_i) \neq \emptyset$. But then, according to Lemma 5.57, we deduce that $\texttt{Reach}_r(\gamma_i) \subsetneq \texttt{Reach}_r(\gamma_{i+1})$ or $\texttt{Exp}_r^1(\gamma_i) \subsetneq \texttt{Exp}_r^1(\gamma_{i+1})$.
   If we now suppose that $\forall i, \texttt{Reach}_r(\gamma_i) \neq \texttt{FD}^o(\gamma_i)$, then we build an infinite sequence of growing sets of nodes, which cannot happen. By contradiction, we conclude. ■

Since we can now apply the properties of $\texttt{Reach}$, and especially Lemma 5.56 to $\texttt{FD}^o$, we can conclude that at some point, $\texttt{FD}^o = G$.

Theorem 5.18

$\texttt{FD}^o(\gamma_f) = G$

Proof: Let us reason by contradiction and suppose there exists one node $u \notin \texttt{FD}^o(\gamma_f)$. Let us consider one such node u with maximal height, *i.e.* such that $\forall p \in \mathcal{P}(u), p \in \texttt{FD}^o(\gamma_f)$. By definition this implies that $\forall p \in \mathcal{P}(u), \texttt{tok}_p^\gamma = \bot \vee p = r$. Thus, $u \notin \texttt{FD}^o(\gamma_f)$ is possible only by $\texttt{tok}_v^{\gamma_f} \neq \bot$. Since γ_f is a reset point, it means that $\exists a \in \mathbb{A}^* : v \in D_a$, and thus v is not activated at all.

Let us now consider one parent p of v. According to Lemma 5.59, we have $p \in \texttt{Reach}_r^{\gamma_f}$, and according to Lemma 5.56, p executes a circulation round after γ_f. But thus, according to Lemma 5.53 when p starts its circulation round we have $\texttt{tok}_v = \bot$ which is contradictory. ■

Corollary 5.3

$\texttt{Reach}_r(\gamma_f) = G.$

Theorem 5.19

Our algorithm is a fair self-stabilizing token circulation algorithm.

Proof: We already proved in Theorem 5.17 that our algorithm satisfies Condition 2 of Specification 5.1.

Let us consider one execution $\epsilon = \gamma_0 \to \cdots$, and more precisely the subexecution $\epsilon_f = \gamma_f \to \cdots$

Remark that having $\texttt{FD}^o(\gamma_f) = G$ implies that all nodes v but the root are such that $\texttt{tok}_v^{\gamma_f} = \bot$. Therefore, $\mathbb{A}^*(\gamma_f) = \emptyset$, and by Lemma 5.9, this is true in all configurations of ϵ_f. In particular, in any configuration of ϵ_f, the set of nodes with $\texttt{tok}_v \neq \bot$ corresponds to the clean branch-$\texttt{CD}^o$ D_r^γ, and by construction, we have $U_{\mathcal{TC}}(\gamma)$. This proves that our algorithm satisfies Condition 1 of Specification 5.1.

Furthermore, let us consider one circulation round of ϵ_f which starts at γ^1, and let us denote γ^2 the reset point consecutive to γ^1. According to Lemma 5.56, all nodes of $\texttt{Reach}_r(\gamma^1) = G$ execute exactly one circulation round during that circulation round. Since $\forall v \in G \setminus \{r\}, \texttt{tok}_v^{\gamma^1} = \texttt{tok}_v^{\gamma^2} = \bot$, this means that there is exactly one computing step in which v executes $\mathbb{R}_{\texttt{Win}}$, and thus there exists a finite, continuous, subexecution of this round in which $R(v)$. Therefore, this circulation round is fair. ■

5.4.4 Space Optimality of our Algorithm

In this section, we prove that our algorithm is asymptotically optimal in terms of memory for the token circulation problem. Our algorithm has a space complexity $\mathbb{S}(\mathcal{A}) \sim \log\log n$ bits per node.

Thus, we are going to prove that there does not exist any deterministic algorithm solving $\mathcal{TC}$ in $\mathcal{S}_{\mathsf{PU}}$ using $o(\log\log n)$ bits per node. We prove this lower bound under less challenging hypotheses than those under which our algorithm works. Namely, our algorithm is self-stabilizing, it works under the unfair distributed scheduler and in any DODAG. Our lower bound is established for general distributed algorithms (without any stabilization requirements), under simpler schedulers (the synchronous scheduler and the central strongly fair scheduler), and in a very simple class of $\texttt{D}^o$: the *bi-branch* $\texttt{D}^o$.

We basically use Theorem 3.2 established in Chapter 3 which introduces the equivalence between executions of small-memory algorithms in identified networks, and in anonymous networks. Specification 5.1 of $\mathcal{TC}$ relies on two local predicates, T and R. Similarly to how

we proved the lower bound for leader election (Theorem 3.1), we can consider the function $(\mathcal{A}, T, R)$ which has asymptotically the same space complexity as $\mathcal{A}$.

Let us first define the topology used to prove our lower bound. A bi-branch $\mathtt{D}^o$ is a $\mathtt{D}^o$ in which the root has two children, and all the other nodes have one child, and one parent. Although we call it $\mathtt{D}^o$, it is actually a tree, with bounded degree. For simplicity, we provide the set of edges of the bi-branch as a set of oriented edges, which matches the relation $\rightarrow$.

Definition 5.31 (*bi-branch* $\mathtt{D}^o$ B_k)

For any $k \geq 1$, we define B_k the k-th bi-branch $\mathtt{D}^o$ by $B_k = (V_k, E_k)$ where:

- $V_k = \{r, u_1, u_2, \cdots, u_k, v_1, v_2, \cdots, v_k\}$,
- $E_k = \{(u_1, r), (u_2, u_1), \cdots, (u_k, u_{k-1}), (v_1, r), (v_2, v_1), \cdots, (v_k, v_{k-1})\}$.

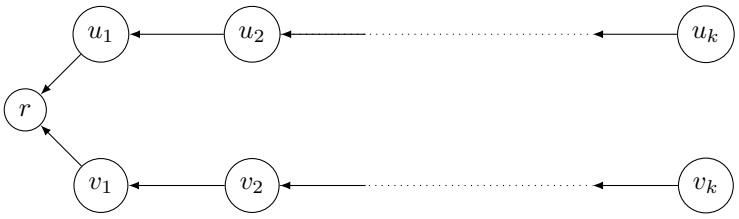

Figure 5.29: Bi-branch $\mathtt{D}^o$ B_k

Similarly to how we considered symmetric configurations on rings in Section 3.6, we define the symmetric configurations on bi-branch $\mathtt{D}^o$.

Definition 5.32 (*Symmetric configurations on B_k*)

Let us consider $k \geq 1$. A configuration of B_k, γ, is said symmetric if
$\forall i \in [1, k], \mathtt{state}^{\gamma}_{u_i} = \mathtt{state}^{\gamma}_{v_i}$.

Theorem 5.20 (*Lower Bound for $\mathcal{TC}$ in $\mathcal{S}_{\mathsf{PU}}$*)

Let $c > 1$. Every deterministic algorithm solving $\mathcal{TC}$ in $\mathcal{S}_{\mathsf{PU}}$ under a central strongly fair scheduler or under the synchronous scheduler requires registers of size $\Omega(\log \log n)$ bits per node in bi-branch $\mathtt{D}^o$'s with unique identifiers in $[1, n^c]$.

Proof: Similarly to the proof of Theorem 3.1, we first prove the theorem under the hypothesis of the central strongly fair scheduler, and will show later how to treat the case of the synchronous scheduler.

Let us suppose that there exists an algorithm $\mathcal{A}$ solving $\mathcal{TC}$ in $\mathcal{S}_{\mathsf{PU}}$ under a central strongly fair scheduler, using $o(\log \log n)$ bits per nodes in bi-branch $\mathtt{D}^o$'s with unique identifiers in $[1, n^c]$. By definition, there exist two local predicates T and R which specifies how $\mathcal{A}$ solves $\mathcal{TC}$.

The class $\mathcal{G} = (B_i)_{i \geq 1}$ contains graphs of arbitrary large size, and is Δ-limited by $\log n$ since all the graphs it contains have degree 2. Therefore, we can apply Theorem 3.2. Rather than simply applying it to $\mathcal{A}$, we consider the tuple $(\mathcal{A}, T, R)$. Theorem 3.2 guarantees that, for any n large-enough, there exist n distinct identifiers $\mathtt{ID}_1, \ldots, \mathtt{ID}_n$ in $[1, n^c]$ which the tuple $(\mathcal{A}, T, R)$ does not distinguish.

Let us now consider one such large-enough network B_n, such that the identifiers of the nodes are $\mathtt{ID}_1, \ldots, \mathtt{ID}_n$. We consider the framework of distributed algorithms, where the initial configuration is not corrupted. Therefore, we suppose a configuration γ_0 such that

the root holds the token (*i.e.* $T^{\gamma_0}(r) = \texttt{true}$), and that all the other nodes are in one unique state. By definition, γ_0 is symmetric. Let us build step by step an infinite execution such that $\forall i \in \mathbb{N}, T^{\gamma_i}(r)$. This is clearly contradictory with the fairness of the token circulation.

We reason by induction, building step by step an execution in such that each one or two configuration is symmetric, and such that the root permanently holds the token. The base case is immediate: γ_0 respects both properties.

Let us suppose that γ_k is symmetric, and $T^{\gamma_k}(r) = \texttt{true}$. By symmetry, we have $\forall i \in [1, k]$, for all rules R of $R_{\mathcal{A}}$, R is enabled on u_i if and only if it is enabled on v_i. In particular, u_i is enabled if and only if v_i is enabled. Since by hypothesis $\mathcal{A}$ solves $\mathcal{TC}$, there is at least one enabled node.

If the central strongly fair scheduler activates r in γ_k, then the resulting configuration γ_{k+1} is symmetric. Predicate T only relies on the states of the nodes, since it cannot distinguish the identifiers by construction, and therefore $T^{\gamma_{k+1}}(u_i) \iff T^{\gamma_{k+1}}(v_i)$ by symmetry. By unicity of the token, we have $T^{\gamma_{k+1}}(r)$, and the induction step is established.

Suppose now that the central strongly fair scheduler activates u_i in γ_k. In this case, the scheduler can activate v_i in γ_{k+1}. Since there is no edge between u_i and v_i, the action of u_i does not have any incidence on v_i, which means that v_i takes the exact same step as u_i. As a result, γ_{k+2} is symmetric, and consequently r holds the token in γ_{k+2}. Finally, we prove that r also holds the token in γ_{k+1}. Since predicate T is local, and since only u_i is activated in γ_k, only neighbors of u_i or u_i itself may have their value on T updated between γ_k and γ_{k+1}. Let w be the node such that $T^{\gamma_{k+1}}(w) = \texttt{true}$, and suppose that $w \neq r$, *i.e.* $w = u_j$ with $j \in \{i-1, i, i+1\}$. Therefore, by symmetry and since v_i takes the exact same step as u_i, and is the only activated node in γ_{k+1}, $T^{\gamma_{k+2}}(v_j) = T^{\gamma_{k+2}}(u_j) = \texttt{true}$, which breaks the unicity of the token. As a consequence, r holds the token in γ_{k+1}, and the induction step is established.

Let us now consider the synchronous scheduler. All the enabled nodes are activated at each step, which preserves symmetry at each step, and thus the root permanently holds the token too. ■

5.5 Conclusion

In this chapter, we introduce a memory-optimal self-stabilizing algorithm for the fair token circulation in arbitrary DODAGs. Our algorithm works under the port-unknown state model $\mathcal{S}_{\mathsf{PU}}$, and requires only $\Theta(\log \log n)$ bits per node. To our knowledge, this is the first self-stabilizing algorithm which breaks symmetry in this model, in graphs of arbitrary degree with this complexity.

One consequence of the combination of the communication model $\mathcal{S}_{\mathsf{PU}}$ and of this low space complexity is that nodes cannot store a permanent pointer to any of their neighbors. This induces many complications compared to other circulation algorithms in the literature. We overcome these complications by the use of many precautions in the design of the rules of the algorithm, and notably for the passing of the token from one node to another. The resulting algorithm is relatively complex, and so is the proof of its validity.

Due to the complexity if the algorithm, we had to make a rather formal demonstration of its validity. We use the well-known concept of potential functions to prove the finiteness of circulation rounds, but ours is particularly elaborated, which make the proof especially trustful.

Chapter 6

Conclusion

So long
Farewell
Aufwiedersehn
Goodbye

Von Trapp Family

This thesis focuses on low-memory self-stabilizing algorithms for networks in which point-to-point communication is not built-in. With the recent development of networks composed of small autonomous devices, such algorithms are more and more desirable. We are interested in both lower and upper bounds. By lower bounds, we mean impossibility theorems, which prove some minimal requirement in terms of memory to solve a problem. By upper bound, we mean efficient algorithms, with better complexity than was previously achieved.

We obtained both types of results, and in various, and general settings.

Our first contribution is an impossibility result. We prove that, whatever the actual settings of the networks, one cannot use the algorithmic power procured by unique identifiers with less memory than $O(\log \log n)$ bits per node. We proved that under that, execution of algorithms was the same on identified networks, and on homonymous or anonymous networks. We obtain this result by a counting argument. We prove that, although the behavior of an algorithm can depend on the value of the identifier, if the memory is too small, then there are much less way to behave differently than there are identifiers. In other words, there always exists a collection of several identifiers that the algorithm cannot distinguish through the choices it makes.

We extend this lower bound from executions to problems, by proving as an example a $\Omega(\log \log n)$ lower bound for the leader election problem. The main idea is to consider jointly the behavior of the algorithm, and the predicate that decides whether the node is leader or not. The boolean predicate does not significantly increase the size of the output, and therefore the reasoning made previously remains valid. This technique can be applied to any problem whose specification boils down to a constant number of local boolean predicates.

This lower bound of $\Omega(\log \log n)$ for the leader election problem is tight on graphs with degree less than $\log n$, since an algorithm was presented in [BT20] for the leader election problem, with only $\Theta(\log \log n + \log \Delta)$ bits per node required.

Our second contribution is twofold. The main part is a silent snap-stabilizing, for the detection of termination in anonymous networks, which requires $\Theta(\log D)$ bits per node.

The problem of termination detection was widely studied in the past years, but there are very few snap-stabilizing algorithms in the literature, and this is the first such algorithm designed for anonymous networks, and has, furthermore, a low memory requirement.

To design this algorithm, we basically use two counters, that are both synchronized with a self-stabilizing unison algorithm. One counter is local to each node, and the other one is synchronized between neighbors.

Any self-stabilizing unison algorithm can be chosen for the synchronization of the two counters of our algorithm. This genericity was achieved after a theoretical analysis of unison algorithms, derived only from the specification of the Unison. We established strong synchronization properties that are necessarily satisfied by any algorithm solving unison.

Our third contribution is a token circulation algorithm. It is the first sublogarithmic token circulation algorithm for this problem in the port-unknown state-model, and we also prove that it is optimal in memory. The combination of the communication model and the memory requirements makes it impossible to statically designate one unique neighbors, which highly complicates the passing of the token. Our algorithm works under the weakest scheduling assumptions, on any Destination-Oriented Directed Acyclic Graphs, a class which notably contains trees.

General Perspectives

In this section, we propose general perspective as potential extensions of the presented results.

Our termination detection algorithm only returns positive answers: no answer is given while the observed algorithm does not effectively terminate according to its specification. Adapting our solution to provide snap-stabilizing negative answers, without any false negative is not so easy to achieve. It seems to be dependent on stabilizing time of the unison algorithm. Therefore, we should be able to achieve this goal by developing an *ad-hoc* solution, but likely at the expense of genericity. Also, two other issues remain open in anonymous settings, namely (i) the question of necessity for nodes to know (an upper bound on) the diameter of the network (no matter the solution being self- or non-self-stabilizing), and (ii) the optimality in both spaces and time to solve snap-stabilizing TD in anonymous networks. More generally, is $O(\log D)$ bits per node necessary to solve global observation problems in anonymous networks ?

Our lower $\Omega(\log\log n)$ bits per node lower bound applies to various problems. One remaining question is the tightness of this bound. Indeed, the best complexity achieved so far for the leader election problem, and the spanning tree construction is $O(\log\log n+\log\Delta)$ bits per node. The question whether it is possible to solve leader election or other spanning structure construction with exactly $\Theta(\log\log n)$ bits per node, or whether an additional cost is necessary, remains open.

Our self-stabilizing token circulation algorithm is providing hope that it might be possible to design self-stabilizing token circulation algorithms for arbitrary networks G, beyond the case of $\mathtt{D}^o$, with space-complexity $O(\log\log n)$ bits. The design of such algorithms requires to overcome at least two problems: the presence of more than one root, and the symmetry caused by the presence of cycles. The presence of multiple roots is an issue which may not be too dramatic, as it may be possible to let several tokens circulate, one per root, and to remove the tokens one by one until a single token remains. The symmetries caused by the presence of cycles appear to cause severe difficulties, and our current knowledge is insufficient to guarantee that a space-complexity $O(\log\log n)$ bits can be achieved under such symmetries.

Such an algorithm with complexity $\Theta(\log\log n)$ bits per node, solving token circulation on arbitrary graphs would be a valuable toolbox which could be used to solve other problems, such as the leader election, the spanning tree construction, *etc.* Combined with our lower bound, this would be a significant advance in the knowledge of the complexity of a large variety of problems.

Bibliography

[ADD+20] Karine Altisen, Ajoy K. Datta, Stéphane Devismes, Anaïs Durand, and Lawrence L. Larmore. Election in unidirectional rings with homonyms. *J. Parallel Distributed Comput.*, 146:79–95, 2020.

[ADDP19] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit, editors. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing. Morgan & Claypool Publishers, 2019.

[Ang80] Dana Angluin. local and global properties in networks of processors. In *proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)*, pages 82–93, 1980.

[ANT12] Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012*, pages 126–132, 2012.

[AO94] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). In *13th Annual ACM Symposium on Principles of Distributed Computing, PODC 1994*, pages 254–263, 1994.

[BBD18] Lélia Blin, Fadwa Boubekeur, and Swan Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. *J. Parallel Distributed Comput.*, 117:50–62, 2018.

[BDF20] Lélia Blin, Swan Dubois, and Laurent Feuilloley. Silent mst approximation for tiny memory. In *Stabilization, Safety, and Security of Distributed Systems - 22nd International Symposium, SSS 2020*, pages 118–132, 2020.

[BDPV07] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[BF15] Lélia Blin and Pierre Fraigniaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015*, pages 589–598, 2015.

[BFB19] Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Brief announcement: Memory lower bounds for self-stabilization. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 37:1–37:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BFB21] Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Optimal space lower bound for deterministic self-stabilizing leader election algorithms. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021*, volume 217 of *LIPIcs*, pages 24:1–24:12, 2021.

[BFB22] Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Borne inférieure optimale pour la complexité spatiale des algorithmes déterministes auto-stabilisants d'élection. In *13e rencontres francophones sur les aspects algorithmiques des télécommunications, Algotel 2022, May 30 - June 3, Université Paris-Saclay, France*, 2022.

[BGJ99] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *18th Annual ACM Symposium on Principles of Distributed Computing, PODC 1999*, pages 199–207, 1999.

[BGW89] GM Brown, MG Gouda, and CL Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.

[BJBP22a] Lélia Blin, Colette Johnen, Gabriel Le Bouder, and Franck Petit. Détection de terminaison auto-stabilisante silencieuse, anonyme et instantanée. In *13e rencontres francophones sur les aspects algorithmiques des télécommunications, Algotel 2022, May 30 - June 3, Université Paris-Saclay, France*, 2022.

[BJBP22b] Lélia Blin, Colette Johnen, Gabriel Le Bouder, and Franck Petit. Silent anonymous snap-stabilizing termination detection. In *41th IEEE Symposium on Reliable Distributed Systems (SRDS 2022), Vienna, Austria, October 19-22, 2022*. IEEE Computer Society, 2022.

[BKN17] Lucas Boczkowski, Amos Korman, and Emanuele Natale. Minimizing message size in stochastic communication patterns: Fast self-stabilizing protocols with 3 bits. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 2540–2559, 2017.

[BLP08] Christian Boulinier, Mathieu Levert, and Franck Petit. Snap-stabilizing waves in anonymous networks. In *Distributed Computing and Networking, 9th International Conference, ICDCN 2008*, volume 4904 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2008.

[BP89] JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[BPBRT09] Lélia Blin, Maria Potop-Butucaru, Stéphane Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *23rd International Symposium on Distributed Computing (DISC 2009)*, pages 407–422, 2009.

[BPR05] Joffroy Beauquier, Laurence Pilard, and Brigitte Rozoy. Observing locally self-stabilization. *J. High Speed Networks*, 14(1):3–19, 2005.

[BPR06] Joffroy Beauquier, Laurence Pilard, and Brigitte Rozoy. Observing locally self-stabilization in a probabilistic way. *J. Aerosp. Comput. Inf. Commun.*, 3(10):516–537, 2006.

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004*, pages 150–159. ACM, 2004.

[BPV08] Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. *Algorithmica*, 51(1):61–80, 2008.

[BT18] Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, 31(2):139–166, 2018.

[BT20] Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. *J. Parallel Distributed Comput.*, 144:278–294, 2020.

[CDD+16] Alain Cournier, Ajoy Kumar Datta, Stéphane Devismes, Franck Petit, and Vincent Villain. The expressive power of snap-stabilization. *Theor. Comput. Sci.*, 626:40–66, 2016.

[CDPR20] Arnaud Casteigts, Swan Dubois, Franck Petit, and John Michael Robson. Robustness: A new form of heredity motivated by dynamic networks. *Theor. Comput. Sci.*, 806:429–445, 2020.

[CDPV06] Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain. Snap-stabilizing depth-first search on arbitrary networks. *The Computer Journal*, 49(3):268–280, 2006.

[CDV09] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):6:1–6:27, 2009.

[CFG92] JM Couvreur, N Francez, and M Gouda. Asynchronous unison. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems (ICDCS'92)*, pages 486–493, 1992.

[DGFTT14] Carole Delporte-Gallet, Hugues Fauconnier, and Hung Tran-The. Leader election in rings with homonyms. In *NETYS*, pages 9–24, 2014.

[DGS99] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[Dij82] Edsger W. Dijkstra. *Self-Stabilization in Spite of Distributed Control*, pages 41–46. Springer New York, New York, NY, 1982.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[DJ19] Stéphane Devismes and Colette Johnen. Self-stabilizing distributed cooperative reset. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019*, pages 379–389. IEEE, 2019.

[DJPV00] Ajoy Kumar Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000.

[DLV11a] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *Journal of Parallel and Distributed Computing*, 71(11):1532–1544, 2011.

[DLV11b] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011. Stabilization, Safety and Security.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DT11] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. arXiv: 1110.0334, 2011.

[EK21] Yuval Emek and Eyal Keren. A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks. In *PODC '21: ACM Symposium on Principles of Distributed Computing*, pages 93–102. ACM, 2021.

[Feu19] Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019.

[FF16] Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016. url: bulletin.eatcs.org link, arXiv: 1606.04434.

[FKK+04] Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, and Nicola Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004.

[FMS03] Paola Flocchini, Bernard Mans, and Nicola Santoro. Sense of direction in distributed computing. *Theor. Comput. Sci.*, 291(1):29–53, 2003.

[Fra80] Nissim Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42–55, 1980.

[GH96] Mohamed G. Gouda and F. Furman Haddix. The stabilizing token ring in three bits. *J. Parallel Distrib. Comput.*, 35(1):43–48, 1996.

[GH99] M Gouda and F Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 48–53. IEEE Computer Society Press, 1999.

[Gho93] S Ghosh. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems*, 15:735–742, 1993.

[GLM06] Vijay Gupta, Cedric Langbort, and Richard M. Murray. On the robustness of distributed algorithms. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 3473–3478, 2006.

[God02] Emmanuel Godard. A self-stabilizing enumeration algorithm. *Inf. Process. Lett.*, 82(6):299–305, 2002.

[God19] Emmanuel Godard. Snap-stabilizing tasks in anonymous networks. *Theory Comput. Syst.*, 63(2):326–343, 2019.

[GP16] Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 19–28, 2016.

[HC93] ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[HP00] Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.

[HPP+15] James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 91–100, 2015.

[IL94] Gene Itkis and Leonid A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Annual Symposium on Foundations of Computer Science FOCS 1994*, pages 226–239, 1994.

[ILS95] Gene Itkis, Chengdian Lin, and Janos Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *Distributed Algorithms, 9th International Workshop, WDAG '95*, pages 288–302, 1995.

[JB95] C Johnen and J Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4.1–4.15, 1995.

[JN18] Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2620–2632, 2018.

[Joh97] Colette Johnen. Memory efficient, self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 97*, page 288, 1997.

[KK07] Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.

[KKP10] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[LPPP05] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005.

[LS92] Chengdian Lin and Janos Simon. Observing self-stabilization. In Norman C. Hutchinson, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1992*, pages 113–123. ACM, 1992.

[Mar65] J. J. Martin. Distribution of the time through a directed, acyclic network. *Operations Research.*, 13(1):46–66, 1965.

[Maz97] Antoni W. Mazurkiewicz. Distributed enumeration. *Inf. Process. Lett.*, 61(5):233–239, 1997.

[NS95] Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995.

[PV07] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *Journal of Parallel Distributed Computing*, 67(1):1–12, 2007.

[SSP85] Boleslaw K. Szymanski, Yuan Shi, and Noah S. Prywes. Terminating iterative solution of simultaneous equations in distributed message passing systems. In Michael A. Malcolm and H. Raymond Strong, editors, *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pages 287–292. ACM, 1985.

[Tel88] Gerard Tel. Total algorithms. In Vogt FH (ed) Springer, editor, *Concurrency 88*, volume LNCS 335, pages 277–291. Springer-Verlag, 1988.

[TR21] P. Thubert and M. Richardson. Routing for rpl (routing protocol for low-power and lossy networks) leaves. RFC 9010, RFC Editor, April 2021.

[WTB+12] Tim Winter, Pascal Thubert, Anders Brandt, Jonathan W. Hui, Richard Kelsey, Philip Alexander Levis, Kris Pister, René Struik, Jean-Philippe Vasseur, and Roger K. Alexander. RPL: ipv6 routing protocol for low-power and lossy networks. *RFC*, 6550:1–157, 2012.

Optimisation de la Mémoire pour les Algorithmes Distribués Auto-Stabilisants

Résumé : L'auto-stabilisation est un paradigme adapté aux systèmes distribués, particulièrement susceptibles de subir des fautes transitoires. Des erreurs de corruption de mémoire, de messages, la rupture d'un lien de communication peuvent plonger le système dans un état incohérent. Un protocole est auto-stabilisant si, quel que soit l'état initial du système, il garantit un retour à un fonctionnement normal en temps fini.

Plusieurs contraintes s'appliquent aux algorithmes conçus pour les systèmes distribués. L'asynchronie en est un exemple emblématique. Avec le développement de réseaux d'objets connectés, censés être autonomes, il devient également central de concevoir des algorithmes ayant un faible coût en termes de consommation énergétique et peu exigeants en termes de ressources.

Une des manières d'appréhender ces problèmes est de chercher à réduire la taille des messages échangés entre les différents nœuds du réseau. Cette thèse se concentre sur l'optimisation de la mémoire nécessaire à la communication pour les algorithmes distribués auto-stabilisants.

Nous établissons dans cette thèse plusieurs résultats négatifs, démontrant l'impossibilité de résoudre certains problèmes sans une certaine taille minimale pour les messages échangés, en établissant une impossibilité d'utiliser jusqu'au bout l'existence d'identifiants uniques dans le réseau en dessous de cette taille minimale. Ces résultats sont génériques et peuvent s'appliquer à de nombreux problèmes distribués. Dans un second temps, nous proposons des algorithmes particulièrement efficaces en mémoire pour la résolution de deux problèmes fondamentaux des systèmes distribués: la détection de terminaison, et la circulation perpétuelle de jeton.

Mots clés : Algorithmes distribues, tolerance aux pannes, auto-stabilisation, anonymat, optimisation mémoire

Memory-Optimization for Self-Stabilizing Distributed Algorithms

Abstract: Self-stabilization is a suitable paradigm for distributed systems, particularly prone to transient faults. Errors such as memory or messages corruption, break of a communication link, can put the system in an inconsistent state. A protocol is self-stabilizing if, whatever the initial state of the system, it guarantees that it will return a normal behavior in finite time.

Several constraints concern algorithms designed for distributed systems. Asynchrony is one emblematic example. With the development of networks of connected, autonomous devices, it also becomes crucial to design algorithms with a low energy consumption, and not requiring much in terms of resources.

One way to address these problems is to aim at reducing the size of the messages exchanged between the nodes of the network. This thesis focuses on the memory optimization of the communication for self-stabilizing distributed algorithms.

We establish in this thesis several negative results, which prove the impossibility to solve some problems under a certain limit on the size of the exchanged messages, by showing an impossibility to fully use the presence of unique identifiers in the network below that minimal size. Those results are generic, and may apply to numerous distributed problems. Secondly, we propose particularly efficient algorithms in terms of memory for two fundamental problems in distributed systems: the termination detection, and the token circulation.

Keywords: Distributed algorithms, fault-tolerance, self-stabilization, anonymity, memory optimization