Topological Self-Stabilization with Name-Passing Process Calculi

Christina Rickmann[1], Christoph Wagner[2], Uwe Nestmann[3], and Stefan Schmid[4]

1 Technische Universität Berlin, Germany
c.rickmann@tu-berlin.de
2 Technische Universität Berlin, Germany
christoph.wagner@tu-berlin.de
3 Technische Universität Berlin, Germany
uwe.nestmann@tu-berlin.de
4 Aalborg University, Denmark
schmiste@cs.aau.dk

Abstract

Topological self-stabilization is the ability of a distributed system to have its nodes themselves establish a meaningful overlay network. Independent from the initial network topology, it converges to the desired topology via forwarding, inserting, and deleting links to neighboring nodes.

We adapt a linearization algorithm, originally designed for a shared memory model, to asynchronous message-passing. We use an extended localized π-calculus to model the algorithm and to formally prove its essential self-stabilization properties: closure and weak convergence for every arbitrary initial configuration, and strong convergence for restricted cases.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Distributed Algorithms, Fault Tolerance, Topological Self-Stabilization, Linearization, Process Calculi

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.19

1 Introduction and Technical Preliminaries

Distributed algorithms are designed to be executed on networked hardware consisting of several connected processes like computers, processors or threads [7]. With the importance of distributed algorithms and the increasing complexity, the need for robust, error-prone solutions rises. The field of self-stabilization [5] offers such fault tolerance. We adapt the algorithm of [4] to a more realistic setting, i.e., we are using a local memory model with asynchronous message-passing opposed to the shared memory model of [4]. Based on an adapted version of the localized π-calculus [8], we formalize the algorithm and use methods similar to [3, 12] to prove it correct.

The approach of self-stabilizing systems was first introduced by [2]. According to [3], the idea of a self-stabilizing system is simply as follows: when started in an arbitrary state it always converges to a desired state. This leads to the ability to tolerate any transient fault, including process crash with recovery, transmission errors like loss or corruption, and corrupted random-access memory. A transient fault is any event that may changes the state of the system, but not its behavior i.e., the program code. The state after the end of the last fault can be considered as a new initial state and the system must recover if no new faults occur for a sufficiently long period of time. Another characteristic of self-stabilizing algorithms is that they must not terminate [3] and processes must continuously communicate with neighboring nodes. As a consequence, the participating processes can not know whether the system is stabilized (i.e., is in a correct configuration). Dolev defines a self-stabilizing system in [3] as follows:

▶ **Definition 1** (Self-Stabilizing System). A self-stabilizing system can be started in any arbitrary configuration and will eventually exhibit a desired "legal" behavior. We define the desired legal behavior by a set of legal executions denoted LE. A set of legal executions is defined for a particular system and a particular task. Every system execution of a self-stabilizing system should have a suffix that appears in LE. A configuration c is *safe* with respect to a task LE and an algorithm if every fair execution of the algorithm that starts from c belongs to LE (*closure*). An algorithm is *self-stabilizing* for a task LE if every fair execution of the algorithm reaches a safe configuration with respect to LE (*[strong] convergence*).

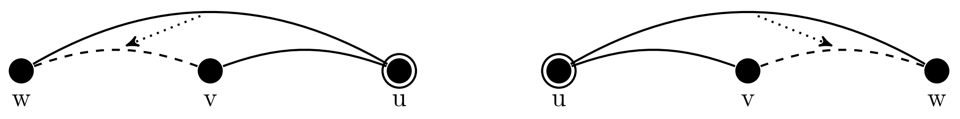
According to [3], so-called *potential functions* are a classic approach to prove convergence. The idea is to define a function over the configuration set and to prove that this function monotonically decreases (or increases) with every executed step. Additionally, it has to be shown that after the function reaches a certain threshold, the system is in a safe configuration. Since the closure property states that every step from a safe or correct configuration leads again to a correct configuration, closure is usually proven through invariants. An easier to achieve and easier to prove property is weak stabilization. According to [6], a system is weakly stabilizing if for every initial configuration there is an execution that reaches a safe or correct configuration. This property is called *weak convergence*.

Topological self-stabilization describes a particular class of self-stabilizing systems. The goal is that the nodes themselves establish a meaningful overlay network, independent from the initial network topology, via forwarding, inserting, and deleting links to neighboring nodes. One of such desired network topologies is a chain. Given a fixed set of nodes V with unique identifiers (ids) and a total order ($\leq$), the goal is to build an ordered list of the nodes according to their ids. Hence, the (undirected) *linear/chain graph* $G_L = (V, E_L)$ is defined as $\{u, v\} \in E_L$ iff $u = \mathit{succ}(v) \vee v = \mathit{succ}(u)$ (where $\mathit{succ}(v)$ defines the $\leq$-next id after v). Since the successor of any node is (if existent) uniquely defined, the linear graph is also unique for a given node set V. According to [4], a linearization algorithm is defined as follows:

▶ **Definition 2** (Linearization). A *linearization algorithm* is a distributed self-stabilizing algorithm where an *initial configuration* forms any (undirected) connected graph $G_0 = (V, E_0)$, the only *legal configuration* is the linear topology $G_L = (V, E_L)$ on the nodes V, and actions only update the neighborhoods of the nodes.

Gall et al. introduce in [4] two variants of a self-stabilizing algorithm for graph linearization, named $\mathit{LIN}_{\mathit{all}}$ and $\mathit{LIN}_{\mathit{max}}$. Both are based on the idea that whenever a node has two neighbors, both of which have a smaller (or both a greater) id, it establishes a link between them and deletes its link to the smaller (respectively greater) one. These steps are called left and right linearization and are depicted in Figure 1. The variants of the algorithm only differ in which linearization steps are enabled.

The algorithm only works in a setting where all nodes have write access to the whole memory as the shared variables are written not only by the nodes themselves but also by their neighbors. Such a shared memory model does not seem a good match for distributed systems like peer-to-peer systems, which usually rely on message-passing, and where communication links may be asymmetric. We redesigned and modelled the algorithm for an asynchronous



(a) Left Linearization. **(b)** Right Linearization.

Figure 1 Linearization steps.

message-passing system. Our algorithm corresponds to the LIN_{all} variant, but it would be equally possible—with a small adjustment—to implement LIN_{max}.

In order to prevent a faulty design of an algorithm and to confirm the correctness of proofs the usage of formal methods is imperative. The π-calculus is a well-known and widely-used process calculus to model concurrent and distributed systems. According to [9] it is designed to naturally express processes with a changing link infrastructure, as the communication between processes carries information that leads to a change in the linkage of processes. We model our algorithm in an extension of the localized π-calculus [8], a distributable variant [1] of the π-calculus [11]. We extend the calculus with data (similar to [12]) in order to explicitly keep track of the neighbors of each node and thus of the systems topology. Each node can receive messages via a channel with the same name as its own id. To enable a neighboring node to communicate with another neighbor it is sufficient to send it the corresponding id. We prove the self-stabilization properties closure, weak convergence, and restricted cases of strong convergence utilizing state-based reasoning rather than an action-based style (cf. [12]).

Related Work. Self-stabilization in the context of distributed computing was first introduced by [2]. The fundamentals of topological self-stabilization and linearization for this work originated in [4], which is also the foundation for the algorithm and the main idea of the proofs. The basics in designing a self-stabilizing algorithm and main proof techniques, as well as a general introduction and overview can be found in [3]. We used an extended localized π-calculus to model our algorithm in an unambiguous way and as a formal basis for proofs. Self-stabilization does not offer masking fault tolerance, i.e., it does not ensure liveness and safety of the whole system, but it still ensures liveness, hence nonmasking fault tolerance. Even though masking fault tolerance is strictly stronger than nonmasking, it might not always be achievable or too costly [5], making nonmasking fault tolerance—thus self-stabilizing algorithms—a good alternative. The basic localized π-calculus is due to [8] and extended in a way similar to [12]. Furthermore, we introduce standard forms for reachable states—again based on ideas from [12], and inspired by [9]—which enables us to explicitly and structurally keep track of the global state and therefore the topology of the system.

Contributions and Overview. We adapt a self-stabilizing algorithm for graph linearization: whereas the original algorithm works only in a very restrictive shared memory model, our algorithm is applicable in a completely asynchronous message-passing system.

We formally prove the closure property, i.e., if the system reaches a correct configuration, it stays in a correct configuration if no fault occurs. Furthermore, we prove strong convergence for restricted cases. Assume an initial configuration that is connected while taking the messages in transit into account. Strong convergence holds whenever every process knows, first, at most its desired neighbors, and second, at least its desired neighbors. For the general case, i.e., an arbitrary connected initial configuration, we prove weak convergence. With strong convergence for the corner cases and weak convergence in general, we have all essentials for proving strong convergence in general. Approaches are discussed in [10].

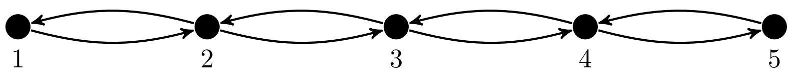


Figure 2 Desired network topology, whereby the nodes are ordered according to their ids.

First, we introduce our model (Section 2) and the redesigned algorithm for asynchronous message-passing (Section 3). Then, we present selected proven properties (Section 4); the complete proofs can be found in [10]. Finally, we summarize our approach and hint at future work (Section 5).

2 Model for Asynchronous Message-Passing

We assume n processes in the system and every process has a unique id. To be as general as possible, we only assume the existence of a total order on these ids.

▶ Assumption (Ids). Every process has a unique constant id.

▶ **Definition 3** (Process identifiers $\mathcal{P}$)**.** Let $\mathcal{P}$ be the (non-empty) finite *set of unique identifiers* of the processes in the system. Let $\leq$ be a total order on $\mathcal{P}$. Let $|\mathcal{P}| = n \in \mathbb{N}$ then there exists an index function (bijection) $i : \mathcal{P} \to \{1, \dots, n\}$ and $\forall p \in \mathcal{P}.i(p) = |\{q \in \mathcal{P} | q \leq p\}|$ i.e., $i(p)$ describes the position of p with respect to $\leq$.

We define the *predecessor* and the *successor* $pred, succ : \mathcal{P} \to (\mathcal{P} \cup \bot)$ of a process as respectively the next smaller and next greater process according to the total order (and $\bot$ if there is none). We call such a pair of processes *consecutive*. The overlay network that the processes (represented as nodes) shall establish is an ordered doubly-linked list according to the total order on the ids (example depicted in Figure 2). Every process has an unidirectional link (represented as edge) to its predecessor and its successor (with exception of the smallest resp. greatest process) and there are no other links. We call this topology the (directed) linear graph. The undirected linear graph is the undirected variant. Here, for every pair of consecutive processes at least one of them has a link to the other one.

▶ **Definition 4** (Desired Topology Graph)**.** The *(directed) linear graph* $G_{LIN} = (\mathcal{P}, E)$ is defined as $E = \{(p, q) | p, q \in \mathcal{P} \wedge (p = succ(q) \vee q = succ(p))\}$ and the *undirected* U_{LIN} is defined accordingly.

The *distance* $dist : \mathcal{P} \times \mathcal{P} \to \mathbb{N}$ is the number of nodes between the two processes according to the total order. A process has a distance of zero to itself and the distance between a pair of consecutive processes is one. The length of a (directed or undirected) edge is defined as the distance between the connected nodes.

Extended Localized Pi-Calculus. To model the algorithm, we introduce an extension of the name-passing localized π-calculus, the extended localized π-calculus $\mathrm{eL}\pi = \langle \mathcal{P}_{\mathrm{eL}}, \longmapsto \rangle$. The extension is based on ideas similar to [12], which allows us to define a kind of standard form for a configuration of our algorithm. The local state of all processes and the messages in transit, and therefore the global state of the system, is directly accessible via the parameters of the corresponding process definition. This in turn allows state-based proofs, which is more traditional for distributed algorithms [7], instead of the action-based style of process calculi.

▶ Notation (Multisets). Let $a, b, c \in S$ be arbitrary elements of an arbitrary set S. We denote with $M = \{\!|a, a, b, c|\!\}$ a multiset and use $\mathbb{N}^S$ as the type of such a set. Furthermore, the union $\cup$ of two multisets is the multiset where all appearances of elements in both are added and the difference $\setminus$ is the multiset where all appearances of elements in the first multiset are decreased by those in the second (but at least zero). Since sets are only special cases with multiplicity one for all elements, we also use combinations of sets and multisets.

We assume the existence of a countably infinite set $\mathbf{A}$ containing all channel names, function names, and variables. $K(X)$ denotes a parameterized process constant, which is defined with respect to a finite set of process equations D of the form $\{K_j(X) = P_j\}_{j \in J}$. Since we use parameterized process constants, we exclude replication and use instead recursion via process definitions to model repetitive behavior.

▶ **Definition 5** (Syntax of the extended Localized π-Calculus: $\mathcal{P}_{\mathrm{eL}}$).

$$\begin{array}{lll}
\textsc{Data Values}\ \ \mathbf{V} & v ::= \bot \ \mid\ 0 \ \mid\ 1 \ \mid\ c \ \mid\ (v,v) \ \mid\ \{v,\dots,v\} \ \mid \\
& \qquad \{\!|v,\dots,v|\!\},\ \text{with } c \in \mathbf{A} \\
\textsc{Variable Pattern} & X ::= x \ \mid\ (X,X)\,,\ \text{with } x \in \mathbf{A} \\
\textsc{Expressions} & e\ ::= v \ \mid\ X \ \mid\ (e,e) \ \mid\ f(e)\,,\ \text{with } f \in \mathbf{A} \\
\textsc{Processes}\ \ \mathbf{P} & P ::= 0 \ \mid\ P \mid P \ \mid\ c(X)\,.P \ \mid\ \overline{c}\langle v\rangle \ \mid\ (\nu c)\,P \ \mid \\
& \qquad \textbf{if } e \textbf{ then } P \textbf{ else } P \ \mid\ \textbf{let } X = e \textbf{ in } P \ \mid\ K(e) \\
\textsc{Process Equations} & \ \ D = \{K_j(X) = P_j\}_{j\in J}\ \ \text{a finite set of process definitions}
\end{array}$$

where in $c(X)\,.P$ every variable x that appears in X may not occur free in P in input position.

Names received as an input and restricted names are *bound names*. The remaining names are *free names*. Accordingly, we assume three sets, the sets of names $\mathsf{n}(P)$ and its subsets of free names $\mathsf{fn}(P)$ and bound names $\mathsf{bn}(P)$, with each term P. To avoid name capture or clashes, i.e., to avoid confusion between free and bound names or different bound names, bound names can be mapped to fresh names by *α-conversion*. We write $P \equiv_\alpha Q$ if P and Q differ only by α-conversion. The substitution of value v for a variable pattern X in expression e or process P is written $\{{}^{v}/_{X}\}e$ and $\{{}^{v}/_{X}\}P$ respectively. Note that only data values can be substituted for names and that all variables of the pattern X must be free in P (while possibly applying α-conversion to avoid capture or name clashes). Let $[\![e]\!]$ denote the evaluation of expression e which allows results in a data value, defined in the standard way.

▶ **Definition 6** (Structural Congruence for the Localized π-Calculus). The structural congruence for the extended localized π-calculus is based on the structural congruence for the π-calculus:

$$\begin{array}{c}
P \equiv Q \text{ if } P \equiv_\alpha Q \qquad P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu n)\,0 \equiv 0 \\
P \mid (\nu n)\,Q \equiv (\nu n)\,(P \mid Q)\,, \text{if } n \notin \mathsf{fn}(P) \qquad\qquad (\nu n)\,(\nu m)\,P \equiv (\nu m)\,(\nu n)\,P \\
\textbf{if } e \textbf{ then } P \textbf{ else } Q \equiv P,\ \text{if } [\![e]\!] = 1 \qquad\qquad \textbf{if } e \textbf{ then } P \textbf{ else } Q \equiv Q \text{ if } [\![e]\!] = 0 \\
\textbf{let } X = e \textbf{ in } P \equiv \{{}^{[\![e]\!]}/_{X}\}P \qquad\qquad K(e) \equiv \{{}^{[\![e]\!]}/_{X}\}P \text{ if } (K(X) = P) \in D
\end{array}$$

We are only interested in the interaction between the processes and not with any further environment. Therefore, we only present a reduction semantics for our extended localized π-calculus, based on the reduction semantics of the π-calculus.

▶ **Definition 7** (Reduction Semantics of the extended Localized π-Calculus: $\longmapsto$). Defined as:

$$\texttt{comm:}\ \frac{}{c(X).P \mid \overline{c}\langle v\rangle \longmapsto \{{}^{v}/_{X}\}P} \qquad\qquad \texttt{res:}\ \frac{P \longmapsto P'}{(\nu c)\,P \longmapsto (\nu c)\,P'}$$

$$\texttt{struct:}\ \frac{P \equiv Q \qquad Q \longmapsto Q' \qquad Q' \equiv P'}{P \longmapsto P'} \qquad\qquad \texttt{par:}\ \frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q}$$

▶ **Definition 8** (Steps). We call a single application of this reduction semantics, $P \longmapsto P'$, a step and write $\Longmapsto$ for the reflexive and transitive closure of $\longmapsto$. We use *execution* to refer to a reduction starting from a particular term.

Every structural extension like function calls, if-then-else-statements, and let-in-statements are evaluated in the structural congruence. The evaluation of these constructs is not considered a step on its own. Hence, internal computations are executed as parts of other steps.

The fault tolerance of self-stabilization is based on the property that the initial state can be arbitrarily. It is sufficient to show that every arbitrary state can serve as an initial state to ensure this form of fault tolerance since the state after every fault can be seen as a new initial state. Thus, we assume for the proofs as usual that there are no faults. An infinite message delay can be seen as message loss. Therefore, every message that is sent is received after finite time. Since the message-passing model is asynchronous, there are no further assumptions regarding the delivery time of messages.

▶ Assumption (No Message Loss). Every message is received after a finite but arbitrary number of steps.

Furthermore, we need an assumption of fairness, as otherwise nodes could starve. A process starves if it never executes a step. Furthermore, a subprocess of a node could starve, e.g. if the process is only consuming messages, but never tries to find a linearization step itself. Without a fairness assumption, it is not possible to show any progress in the system.

▶ Assumption (Fairness). Every continuously enabled subprocess will eventually (after an arbitrary but finite number of steps) execute a step.

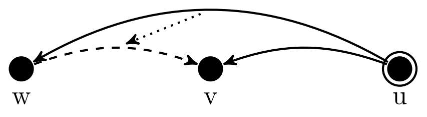
3 Linearization Algorithm for Asynchronous Message-Passing

The utilization of a calculus enables us to model the algorithm unambiguously and allows us to formally prove properties of the algorithm. Although in the calculus itself all channels are bidirectional, we only use them in an unidirectional manner. Together with the output-capability restriction of the localized π-calculus this helps us to ensure that every process can be implemented in an asynchronous setting on a different location [1].

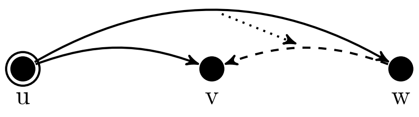
Each node can receive messages from other nodes via a channel with the same name as its id. One could think of the ids as serving as the IP address of the corresponding process. To enable a neighbor to communicate with another process, one sends it the corresponding id. Thus, $\mathbf{A}$ contains the ids of all processes in the system i.e., $\mathcal{P} \subseteq \mathbf{A}$.

To model local variables, we use restricted channels for every process. Each variable is represented through a message in transit that only can be sent and received by the corresponding process. The value of such a local variable is modeled by the value of the matching message in transit. Thus, receiving the message corresponds to reading the variable and sending corresponds to writing. In our algorithm, every process p has one local variable nb_p, describing the neighborhood of the process i.e., it contains the ids of all processes that p knows and can send messages to.

Each process can be in one of two states. In the state $Alg\,(p, nb)$ the process p is able to receive a message from another process in the system. In the state $Alg'\,(p, nb, x)$ the process p can add a previously received process id x to its current neighborhood nb. In doing so, a process blocks the reception of messages until the the previously received id is added to the neighborhood. In both states, the process can additionally try to find a linearization step in its current neighborhood nb based on internal computations. If it finds no possible linearization step, it sends *keep-alive*-messages to its current neighbors.



(a) Directed Left Linearization.



(b) Directed Right Linearization.

Figure 3 Directed linearization steps.

In the algorithm, $\mathit{LeftN} : \mathcal{P} \times 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ calculates the *left neighborhood* of a process i.e., all neighbors with a smaller id and corresponding $\mathit{RightN} : \mathcal{P} \times 2^{\mathcal{P}} \to 2^{\mathcal{P}}$ the *right neighborhood* of a process i.e., all neighbors with a greater id. $\mathit{findLin} : \mathcal{P} \times 2^{\mathcal{P}\times\mathcal{P}} \to 2^{\mathcal{P}\times\mathcal{P}}$ calculates all possible linearization steps in the neighborhood of a process, based on the input set: $\mathit{findLin}(p, y) = \{(q, r) | q, r \in y \wedge q < r \wedge (q, r \in \mathit{LeftN}(p, y) \vee \mathit{RightN}(p, y))\}$.
The function $\mathit{select} : 2^{\mathcal{P}\times\mathcal{P}} \to (\mathcal{P} \times \mathcal{P})$ returns one arbitrary of these linearization steps.

▶ **Definition 9** (Subprocesses). For every process $p \in \mathcal{P}$ we denote $\mathit{Alg}_{\mathit{match}}(p)$, $\mathit{Alg}_{\mathit{rec}}(p)$, and $\mathit{Alg}_{\mathit{add}}(p, \cdot)$ as *subprocesses* of p. Note that all subprocess are input guarded.

$$\mathit{Alg}(p, \mathit{initNb}) = (\nu nb_p)\left(\overline{nb_p}\langle \mathit{initNb}\rangle \mid \mathit{Alg}_{\mathit{rec}}(p) \mid \mathit{Alg}_{\mathit{match}}(p) \right)$$
$$\mathit{Alg}'(p, \mathit{initNb}, x) = (\nu nb_p)\left(\overline{nb_p}\langle \mathit{initNb}\rangle \mid \mathit{Alg}_{\mathit{add}}(p, x) \mid \mathit{Alg}_{\mathit{match}}(p) \right)$$
$$\mathit{Alg}_{\mathit{rec}}(p) = p(x).\mathit{Alg}_{\mathit{add}}(p, x) \qquad \mathit{Alg}_{\mathit{add}}(p, x) = nb_p(y).\left(\overline{nb_p}\langle y \cup \{x\}\rangle \mid \mathit{Alg}_{\mathit{rec}}(p)\right)$$

```
$\mathit{Alg}_{\mathit{match}}(p) = nb_p(y).\big($ let $x = \mathit{select}(\mathit{findLin}(p, y))$ in
        if $x = \bot$ then $\prod_{j \in y} \overline{j}\langle p\rangle \mid \overline{nb_p}\langle y\rangle$
        else if $x = (j, k)$ then
            if $j < k \wedge k < p$ then $\overline{j}\langle k\rangle \mid \overline{nb_p}\langle y \setminus \{j\}\rangle$
            else if $j < k \wedge p < j$ then $\overline{k}\langle j\rangle \mid \overline{nb_p}\langle y \setminus \{k\}\rangle$
                else $\overline{nb_p}\langle y\rangle$
        else $\overline{nb_p}\langle y\rangle$
    $\mid \mathit{Alg}_{\mathit{match}}(p)\big)$
```

The subprocess $\mathit{Alg}_{\mathit{rec}}(p)$ models the ability of a process to receive a message. When it receives a message with content x, it continues as subprocess $\mathit{Alg}_{\mathit{add}}(p, x)$, thus the process changes its state. $\mathit{Alg}_{\mathit{add}}(p, x)$ reads the current value of the neighborhood of p and adds the previous received process id x. Afterwards, the process is again able to receive any message.

The subprocess $\mathit{Alg}_{\mathit{match}}(p)$ defines the behavior, based on the internal computations of $\mathit{findLin}(p, nb)$, in case process p tries to find a linearization step in its neighborhood nb. If there is no possible linearization step, *select* returns $\bot$ and the process sends *keep-alive*-messages to its current neighbors. In case *select* returns a tuple, it defines a left or right linearization step respectively and p sends the further away process the id of the other process and deletes the receiver from its neighborhood (as depicted in Figure 3). The other two else cases are only implemented to obtain a complete case distinction. It is ensured by the definitions that these branches are never explored. The sending of the message $\overline{nb_p}\langle y\rangle$ ensures that, if there would be a possibility to explore these branches, nothing changes. The same value that was read in the previous step (i.e., received by the $nb_p(y)$ message) is written (i.e., sent) again and therefore the neighborhood remains unchanged. In all cases, the process is directly able to try to find another linearization step.

The system is composed of n such processes. The global states that serve as starting points for the executions of our algorithm are called initial configurations. Later we show that every global state can serve as such a starting point as required for self-stabilization.

▶ **Definition 10** (Initial Configuration). Let

- $\mathcal{P}$ be the *set of unique identifiers* and $P, P' \subseteq \mathcal{P}$ with $P \cup P' = \mathcal{P}$ and $P \cap P' = \emptyset$,
- $init : \mathcal{P} \to 2^{\mathcal{P}}$ a function that defines for every process $p \in \mathcal{P}$ the neighborhood i.e., which process ids are known by p,
- $Msgs \in \mathbb{N}^{\mathcal{P} \times \mathcal{P}}$ a multiset that describes the messages in transit and
- $add : \mathcal{P} \rightharpoonup \mathcal{P}$ a partial function with $\forall p \in P'.\exists q \in \mathcal{P}.(p, q) \in add$ and $\forall p \in P.\forall q \in \mathcal{P}.(p, q) \notin add$ that describes the adding in progress i.e., where $add(p) = q$ describes that p wants to add q to its neighborhood.

Then, an *initial configuration* of the algorithm is defined as the process term:

$$Alg_{all}(P, P', init, Msgs, add) = \prod_{j \in P} Alg(j, init(j)) \mid \prod_{j \in P'} Alg'(j, init(j), add(j)) \mid \prod_{(j,k) \in Msgs} \overline{j}\langle k \rangle$$

In an initial configuration there is for every process $p \in \mathcal{P}$ exactly one $\overline{nb_p}\langle\cdot\rangle$-message. This message can not be lost or duplicated through a previous fault as it models a variable. A transient fault can lead to an arbitrary value of a variable but not to its disappearance or duplication. This does not restrict the fault tolerance of the algorithm. Self-stabilization tolerates transient faults but no permanent ones. The disappearance or duplication of a variable could only be caused by corruption of the program code itself which would be a permanent fault. The value of this message can be an arbitrary set of $\mathcal{P}$ (without p itself), reflecting the arbitrary initial neighborhood of p. These messages of all processes describe the initial network topology. A configuration describes the global state of the system, consisting of the states of all processes and messages in transit.

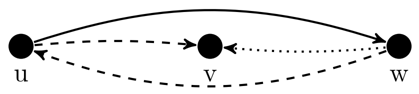
▶ **Definition 11** (Configuration). Let $\mathcal{I}$ be the set of all initial configurations. We call every process term C that can be reached from any arbitrary initial configuration, i.e., $\exists I \in \mathcal{I}.I \Longmapsto C$, *configuration*. We denote the set of all such configurations with $\mathcal{T}$.

▶ **Definition 12** (Reachability). We call a configuration C' *reachable* from a configuration C iff $C \Longmapsto C'$. Further, we say a configuration with a predicate P is *reached* from configuration C iff in every execution there is a configuration C' with $C \Longmapsto C'$ and P holds for C'.

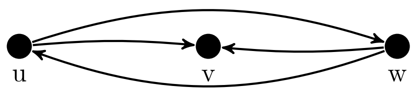
For every configuration C there are parameters with the same properties as in Definition 10 so that C is structurally equivalent to $Alg_{all}(\cdot, \cdot, \cdot, \cdot, \cdot)$. We call $Alg_{all}(\cdot, \cdot, \cdot, \cdot, \cdot)$ the standard form of a configuration and use it as representative of all structurally equivalent configurations. Therefore, every configuration is structurally equivalent to an initial configuration.

▶ **Lemma 13** (Standard form). *Starting from an arbitrary initial configuration every reachable configuration is structurally equivalent to a term $Alg_{all}(P, P', nb, Msgs, add)$ whereby $P, P' \subseteq \mathcal{P}$ with $P \cup P' = \mathcal{P}$ and $P \cap P' = \emptyset$, $nb : \mathcal{P} \to 2^{\mathcal{P}}$ is a function that defines for every process $p \in \mathcal{P}$ the neighborhood, $Msgs \in \mathbb{N}^{\mathcal{P} \times \mathcal{P}}$ is the multiset of messages in transit, $add : \mathcal{P} \rightharpoonup \mathcal{P}$ is a partial function with $\forall p \in P'.\exists q \in \mathcal{P}.(p, q) \in add$ and $\forall p \in P.\forall q \in \mathcal{P}.(p, q) \notin add$.*

▶ Notation (Configuration Components). Let A be an arbitrary configuration then there are parameters P, P', nb, $Msgs$, add with $A \equiv Alg_{all}(P, P', nb, Msgs, add)$ and we denote in the following : $P_A = P$, $P'_A = P'$, $nb_A = nb$, $Msgs_A = Msgs$ and $add_A = add$.



(a) Topology without Messages.



(b) Topology with Messages.

Figure 4 Topology with and without messages whereby solid lines represent the edges.

▶ **Corollary 14** (Steps). *For every configuration A there are always exactly the following steps (up to structural congruence) possible:*

- $\forall p \in \mathcal{P}.select\,(findLin\,(p, nb_A(p))) = \bot \implies$
  $A \longmapsto Alg_{all}\,(P_A, P'_A, nb_A, Msgs_A \cup \{(j,p) | j \in nb_A(p)\}, add_A)$
- $\forall p \in \mathcal{P}.select\,(findLin\,(p, nb_A(p))) = (j,k) \wedge (j < k \wedge k < p) \implies A \longmapsto A'$ *with*
  $A' \equiv Alg_{all}\,(P_A, P'_A, nb, Msgs_A \cup \{(j,k)\}, add_A)$ *and* $nb(x) = \begin{cases} nb_A(x), & \text{if } x \neq p \\ nb_A(p) \setminus \{j\}, & \text{if } x = p \end{cases}$
- $\forall p \in \mathcal{P}.select\,(findLin\,(p, nb_A(p))) = (j,k) \wedge (j < k \wedge p < j) \implies A \longmapsto A'$ *with*
  $A' \equiv Alg_{all}\,(P_A, P'_A, nb, Msgs_A \cup \{(k,j)\}, add_A)$ *and* $nb(x) = \begin{cases} nb_A(x), & \text{if } x \neq p \\ nb_A(p) \setminus \{k\}, & \text{if } x = p \end{cases}$
- $\forall p \in \mathcal{P}.$*if select* $(findLin\,(p, nb_A(p)))$ *does not match any previous case* $\implies A \longmapsto A$
- $\forall p \in P_A.\exists q \in \mathcal{P}.(p,q) \in Msgs_A \implies$
  $A \longmapsto Alg_{all}\,(P_A \setminus \{p\}, P'_A \cup \{p\}, nb_A, Msgs_A \setminus \{(p,q)\}, add_A \cup \{(p,q)\})$
- $\forall p \in P'_A.\exists q \in \mathcal{P}.(p,q) \in add_A \wedge\ A \longmapsto A'$ *with* $nb(x) = \begin{cases} nb_A(x), & \text{if } x \neq p \\ nb_A(p) \cup \{q\}, & \text{if } x = p \end{cases}$ *and*
  $A' \equiv Alg_{all}\,(P_A \cup \{p\}, P'_A \setminus \{p\}, nb, Msgs_A, add_A \setminus \{(p,q)\})$

Topology of Configuration. The system is in a legal state i.e., correct configuration, if the network topology of the system is the desired one, i.e., the linear graph. Thus, we need to define the network topology of a configuration. The network topology graph of a configuration describes to whom each process can send messages directly and therefore corresponds to the neighborhood sets of all processes. To define the several cases in which strong convergence holds and lower the preconditions as much as possible, we introduce variants of the network topology of a configuration. They differ in whether we regard the direction of edges, and if we take messages in transit into account or not. The network topology graphs with messages describe the neighborhoods if all current messages in transit were received and processed (depicted in Figure 4). This variant of the topology is, for example, also needed to define a correct configuration. It is not enough that the current topology corresponds to the linear graph, it also has to be ensured that no undesired connection will be established through a message in transit.

▶ Notation (Topology (without messages)). In figures of the topology graph without messages a solid line represents a process in the neighborhood, a dotted line a message in transit, and a dashed line an adding in progress, i.e., an id that is already received but not yet added to the neighborhood. In the topology graph with messages these are all defined as edges and represented through a solid line.

▶ **Definition 15** (Network Topology Graph (without Messages)). Let A be an arbitrary configuration. The *(directed) topology graph (without messages)* $T(A) = (\mathcal{P}, E)$ is defined as: $E = \{(p,q) | p,q \in \mathcal{P} \wedge q \in nb_A(p)\}$

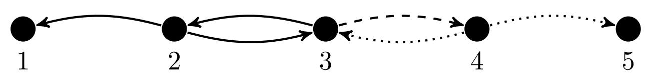


Figure 5 Topology of an undirected correct configuration.

▶ **Definition 16** (Network Topology Graph with Messages)**.** Let A be an arbitrary configuration. The *(directed) topology graph with messages* $T^M(A) = (\mathcal{P}, E)$ is defined as: $E = \{(p,q) | p,q \in \mathcal{P} \wedge (q \in nb_A(p) \vee (p,q) \in Msgs_A \vee add_A(p) = q)\}$.

The undirected variants, i.e., the *undirected topology graph* $U(A)$ and the *undirected topology graph with messages* $U^M(A)$, are defined correspondingly.

▶ Notation (Topology Graph Components)**.** Let A be an arbitrary configuration, we introduce $E_{T(A)}, E_{T^M(A)}, E_{U(A)}$ and $E_{U^M(A)}$ to denote the edges of the different topology graphs $T(A) = (\mathcal{P}, E)$, $T^M(A) = (\mathcal{P}, E')$, $U(A) = (\mathcal{P}, E'')$ and $U^M(A) = (\mathcal{P}, E''')$, i.e., we denote in the following: $E_{T(A)} = E$, $E_{T^M(A)} = E'$, $E_{U(A)} = E''$ and $E_{U^M(A)} = E'''$.

If the topology graph with messages of the initial configuration is weakly connected, the topology graph with messages of all reachable configurations is weakly connected. The only steps that remove edges are linearization steps. If a process executes a linearization step, the removed edge can be simulated by the introduced edge and the edge to the other neighbor of the executing process. Thus, linearization can not result in partitioning the network.

▶ **Lemma 17** (Connectivity)**.** *Let A_0 be an initial configuration and A an arbitrary reachable configuration. Then, it holds that if $U^M(A_0)$ is connected, also $U^M(A)$ is connected.*

Correct Configuration. A configuration is correct if every process knows only its consecutive processes. To ensure that no other connections will be established, it must also hold that every message in transit contains the id of a consecutive process of the receiver. Thus, the network topology with and without messages must be the desired linear graph. With exception of the number of these messages and the state of the processes, the correct configuration is unique.

▶ **Definition 18** (Correct Configuration)**.** Let A be an arbitrary configuration. A is a *correct configuration* iff $T^M(A) = G_{LIN} \;\wedge\; T(A) = G_{LIN}$

▶ **Lemma 19** (Uniqueness, Directed Case)**.** *The correct configuration is unique up to structural congruence, the number of messages in the system and the state of the processes.*

A weaker property is described by an undirected correct configuration. Here, we only demand that the undirected topology with message must be the undirected linear graph (as depicted in Figure 5). Hence, the neighborhood of each process is a subset of its consecutive processes and the messages in transit must satisfy the same requirement as before. To ensure connectivity, between each pair of consecutive processes there must be at least one connection while taking the messages in transit into account. Similarly to a correct configuration, an undirected configuration is uniquely defined with exception of the number of messages, the state of a process, and the type of the connection between a consecutive pair of processes.

▶ **Definition 20** (Undirected Correct Configuration)**.** Let A be an arbitrary configuration. A is an *undirected correct configuration* iff $U^M(A) = U_{LIN}$

▶ **Lemma 21** (Uniqueness, Undirected Case)**.** *The undirected correct configuration is unique up to structural congruence, the number of messages in the system, the state of the processes, and the type of connections.*

4 Results

We want to prove the algorithm correct. In order to show that the algorithm is self-stabilizing, we have to prove *convergence* and *closure* (Definition 1). To be a linearization algorithm according to Definition 2, the system is in a legal state if and only if the topology is the linear graph. The complete proofs together with all omitted results can be found in [10].

Closure. Several closure properties are based on two facts: every process may only remove processes from its own neighborhood and a process never removes the id of a desired neighbor i.e., its predecessor and successor. Further, if every process knows only (a subset of) desired neighbors, there are no more possible linearization steps in the system. All processes only send and receive *keep-alive*-messages to and from desired neighbors respectively.

The only steps that remove edges i.e., ids from the neighborhood, are linearization steps. Whenever a process does not execute any linearization steps, its neighborhood can be expanded by reception of messages but not shrink.

If a process executes a linearization step it never removes a correct neighbor as it always removes the process that is further away. Thus, if a process knows a correct neighbor, it remains in the neighborhood for every reachable configuration. It follows directly that in the directed and undirected topology without messages edges between consecutive neighbors are preserved.

▶ **Corollary 22** (Preservation of Correct Edges). *Let A be an arbitrary configuration. For every reachable configuration R i.e., $A \Longmapsto R$, it holds: $\forall p \in \mathcal{P}.\forall p' \in \{succ(p), pred(p)\}.(p,p') \in E_{T(A)} \implies (p,p') \in E_{T(R)}$ and $\forall p \in \mathcal{P}.\{p, succ(p)\} \in E_{U(A)} \implies \{p, succ(p)\} \in E_{U(R)}$*

This preservation holds further for correct edges in the topologies with messages. Every id carried by a messages cannot get lost and is received and processed eventually. Therefore, also edges between desired neighbors that represent adding or messages are preserved.

▶ **Lemma 23** (Preservation of Correct Edges with Messages). *Let A be an arbitrary configuration. For every reachable configuration R holds: $\forall p \in \mathcal{P}.\forall p' \in \{succ(p), pred(p)\}.(p,p') \in E_{T^M(A)} \implies (p,p') \in E_{T^M(R)}$ and $\forall p \in \mathcal{P}.\{p, succ(p)\} \in E_{U^M(A)} \implies \{p, succ(p)\} \in E_{U^M(R)}$.*

The topology of a configuration contains the desired topology i.e., the linear graph is a subgraph, if every correct edge is already established but possibly undesired edges are still existent in addition. Since every correct edge is always preserved, this property is invariant.

▶ **Corollary 24** (Closure for $T^M \subseteq G_{LIN} \wedge T \subseteq G_{LIN}$). *If A is a configuration with $G_{LIN} \subseteq T^M(A) \wedge G_{LIN} \subseteq T(A)$ it holds for every reachable configuration R i.e., $A \Longmapsto R$, that $G_{LIN} \subseteq T^M(R) \ \wedge \ G_{LIN} \subseteq T(R)$.*

In the topology of an undirected correct configuration every edge is correct but there may be correct edges missing. Through preservation of correct edges this property is invariant. Each configuration that is reachable from an undirected correct configuration is itself an undirected correct configuration. Further, none already established correct edge has been removed.

▶ **Lemma 25** (Closure for Undirected Correct Configuration). *Let A be an arbitrary undirected correct configuration. It holds for every reachable configuration C i.e., $A \Longmapsto C$, that C is also an undirected correct configuration.*

The topology is the linear graph if and only if the configuration is a correct configuration as in Definition 18. *Closure* states once a correct configuration is reached, provided no fault occurs, the system stays in a correct configuration.

▶ **Theorem 26** (Closure for Correct Configurations). *Let A be a correct configuration then it holds for every reachable configuration C i.e., $A \Longmapsto C$, that C is also a correct configuration.*

Convergence We prove strong convergence for restricted cases and weak convergence in general. Strong convergence is proven if either there are possibly correct edges missing but no non-correct edges existent in the topology with messages i.e., the topology is a subgraph of the linear graph, or there are possibly still non-correct edges but at least all correct edges are contained in the topology with messages i.e., the linear graph is a subgraph of the topology.

If there are only correct edges missing, there is at least one connection between every pair of consecutive processes. No more possible linearization steps exist and every process sends *keep-alive*-messages to the current known subset of desired neighbors. All these messages are received and processed eventually and all missing correct edges are eventually established.

▶ **Lemma 27** (Convergence for $U^M = U_{LIN}$). *Let A be an arbitrary configuration. If the undirected network topology with messages is the desired undirected topology i.e., $U^M(A) = U_{LIN}$, then a correct configuration C is reached after a finite number of steps i.e., $A \Longmapsto C \;\wedge\; T^M(C) = G_{LIN} \;\wedge\; T(C) = G_{LIN}$.*

If there are possibly still undesired edges in the topology but at least all correct edges are established, the only processes that can send *keep-alive*-messages are processes that only know their desired neighbors. Every process knows at least its desired neighbors and whenever a process has additional neighbors, there is a linearization step. With every linearization step the topology gets closer to the desired topology which is shown via a potential function.

▶ **Lemma 28** (Convergence for $G_{LIN} \subseteq T^M \wedge G_{LIN} \subseteq T$). *Let A be an arbitrary configuration. If $G_{LIN} \subseteq T^M(A) \wedge G_{LIN} \subseteq T(A)$, then a correct configuration C is reached after a finite number of steps i.e., $A \Longmapsto C \;\wedge\; T^M(C) = G_{LIN} \;\wedge\; T(C) = G_{LIN}$.*

Convergence also holds if the desired topology is a subgraph of the topology with messages. Since every message that is in transit will eventually be received and processed, we always reach a configuration with the linear graph as a subgraph of the topology without messages.

▶ **Lemma 29** (Convergence for $G_{LIN} \subseteq T^M$). *Let A be an arbitrary configuration. If $G_{LIN} \subseteq T^M(A)$, then a correct configuration C is reached after a finite number of steps i.e., $A \Longmapsto C \;\wedge\; T^M(C) = G_{LIN} \;\wedge\; T(C) = G_{LIN}$.*

The proofs use the fact that *keep-alive*-messages are only exchanged between desired neighbors. Proving strong convergence in general is much more difficult as the sending of *keep-alive*-messages to undesired neighbors can cause the reestablishing of undesired connections. Therefore, we show weak convergence i.e., for every initial configuration there are executions that converge to a correct configuration. For this, we define a perfect oracle. It cannot be implemented and should only be seen as a restriction on the set of executions.

▶ **Definition 30** (Perfect Oracle O). A *perfect oracle* is a global omniscient instance that whenever the system is not in an (undirected) correct configuration only let the processes send *keep-alive*-messages to resolve deadlocks and otherwise suppresses all *keep-alive*-messages.

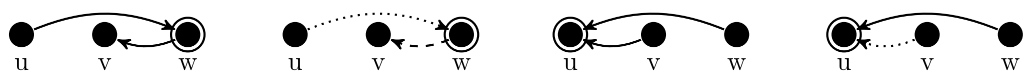


Figure 6 Linearization steps in the undirected topology with messages but not in the directed.

▶ Remark. We show in Theorem 31 that starting from an arbitrary initial configuration a correct configuration is reached after a finite number of steps. A perfect oracle O does not suppress the sending of *keep-alive*-messages in a correct configuration. Once the system is in a correct configuration, it stays in a correct configuration according to the closure property as shown in Theorem 26. Hence, a perfect oracle is not contradictory to the fairness assumption.

We show that every execution that is admissible under the restriction of a perfect oracle converges to a correct configuration. Since for every configuration this set of executions is non-empty, we prove weak convergence in general. We introduced three potential functions and showed that whenever at least one of them is minimal, the system reaches a correct configuration in a finite number of steps. For every configuration, exactly one of the three cases true: there is no more linearization step in the undirected topology with messages, there is a linearization step in the directed topology with messages, or there is a linearization step in the undirected but not in the directed topology with messages. We show that the system neither can stay infinite long in the second case nor can infinitely often alternate between the second and the third case without reaching a configuration in which at least one of the three potential functions is minimal and thus convergence ensured.

▶ **Theorem 31** (Convergence with Perfect Oracle). *Let I be an arbitrary connected, i.e., $U^M(I)$ is connected, initial configuration and A an arbitrary reachable configuration i.e., $I \Longmapsto A$. Assume there is a perfect oracle O. Then a correct configuration C is reached after a finite number of steps i.e., $A \Longmapsto C \;\wedge\; T^M(C) = G_{LIN} \;\wedge\; T(C) = G_{LIN}$.*

5 Conclusion

We adapted the algorithm for shared memory of [4] such that it works in an asynchronous message-passing system. The algorithm of [4] requires a system where all processes must have access to the whole memory, which is very restrictive. In the redesigned algorithm, processes communicate via message-passing and we do not make any assumptions about the time a process needs to execute a step or for message delivery. This makes it applicable in a completely asynchronous message-passing system which is a significantly weaker requirement and corresponds more to real life system conditions.

An algorithm is self-stabilizing, if it satisfies the properties of closure and convergence. We formally proved the closure property, i.e., if the system reaches a correct configuration, it stays in a correct configuration if no fault occurs. There are two forms of convergence. Starting with an arbitrary initial configuration, strong convergence requires that in every execution a correct configuration is reached, whereas weak convergence only claims the existence of such an execution. We proved strong convergence for restricted cases. First, whenever the topology with messages of a connected initial configuration only lacks desired edges but no undesired ones exist, i.e., every process knows at most its desired neighbors, strong convergence holds. Second, strong convergence also holds, whenever in the topology with messages of an initial configuration there are just too many edges, but no desired ones are missing. For the general case, i.e., an arbitrary connected initial configuration, we proved weak convergence. For this proof, we introduced a global omniscient entity, called a perfect oracle. We showed that every execution that is admissible under the assumption of a perfect oracle ensures strong convergence. Since for every initial configuration this is a non-empty set of executions, weak convergence holds in general.

We extended the localized π-calculus, which provides us through its clearly defined syntax and semantics with the possibility to model the algorithm in a precise and unambiguous manner. It is also the basis for formally proving properties about the algorithm. The usage of standard forms [12] of configurations helps us to simplify the proof by identifying every possible reachable process term with a structurally equivalent representative and significantly reduces the number of cases to be analyzed. Further, this enables us to explicitly and conveniently keep track of the global state of the system. This allows us to execute our proofs in a state-based fashion, which is more traditional for distributed algorithms [7], rather than in an action-based style, which would be more typical when using process calculi [11].

Future Work. As we strongly conjecture strong convergence to hold in general, the primary goal is a convergence proof for the general case that works without any oracle at all. The problem is that *keep-alive*-messages can reestablish edges that were already removed through linearization steps. Nevertheless, if a process executes a linearization step, it prevents the further away process eventually from ever sending *keep-alive*-messages to it again. Neither *keep-alive*-messages nor linearization steps establishing edges that are longer as the current longest edge in the topology with messages. The edge that is established through a linearization step is even strictly shorter than the at least temporarily removed one.

The main difficulty is: a potential function that decreases strictly with every linearization step cannot be monotonically decreasing with every step. Thus, potential functions alone are not sufficient to prove strong convergence. A promising approach lies in finding good properties for livelock freedom. With such properties, a general proof can likely be achieved in various ways, as discussed in [10]. For example, livelock freedom properties could be used to show that linearization steps involving the current longest edges are eventually enabled and executed. If it can be shown that the current longest edges are eventually removed permanently, strong convergence would hold.

In preparation for a general proof, it could be interesting to lower the restrictions on the set of executions by an oracle and consider a weaker oracle to acquire further insight in properties that could be helpful for a proof without any oracle.

References

1 P.-D. Brodmann. Distributability of Asynchronous Process Calculi. Master's thesis, Technische Universität Berlin, Germany, October 2014.

2 E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, November 1974.

3 S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

4 D. Gall et al. A Note on the Parallel Runtime of Self-Stabilizing Graph Linearization. *Theory of Computing Systems*, 55(1):110–135, 2014.

5 F. C. Gärtner. Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments. *ACM Comput. Surv.*, 31(1):1–26, March 1999.

6 M. G. Gouda. The Triumph and Tribulation of System Stabilization. In *Proc. of the 9th International WDAG*, WDAG '95, pages 1–18, 1995.

7 N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

8 M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proc. of ICALP*, volume 1443 of *LNCS*, pages 856–867. Springer, 1998.

9 R. Milner. *communicating and mobile systems: the pi-calculus.* Cambridge UP, 1999.

10 C. Rickmann. Topological Self-Stabilization with Name-Passing Process Calculi. Master's thesis, Technische Universität Berlin, Germany, October 2015. arxiv.org/abs/1604.04197.

11 D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge UP, 2001.

12 C. Wagner and U. Nestmann. States in Process Calculi. In *Proc. of EXPRESS/SOS*, volume 160 of *EPTCS*, pages 48–62, 2014.