A new Technique for proving Self-Stabilization under the Distributed Scheduler*

Sven Köhler and Volker Turau

Institute of Telematics
Hamburg University of Technology
Hamburg, Germany
{sven.koehler, turau}@tu-harburg.de

Abstract. Proving stabilization of a complex algorithm under the distributed scheduler is a non-trivial task. This paper introduces a new method which allows to extend proofs for the central scheduler to the distributed scheduler. The practicability of the method is shown by applying it to two existing algorithms, for which stabilization under the distributed scheduler was an open problem.

1 Introduction

The notion of self-stabilization was coined by E. W. Dijkstra [2]. Self-stabilizing distributed systems are guaranteed to converge to a desired state or behaviour in finite time, regardless of the initial state. Convergence is also guaranteed after the system is affected by transient faults, no matter their scale or nature. This makes self-stabilization an elegant and formal approach for non-masking fault-tolerance.

Self-stabilizing algorithms can be designed with different schedulers in mind. Possible schedulers include the central scheduler (only one node can make a move in each step), the distributed scheduler (any number of nodes make a move in each step), and the synchronous scheduler (all nodes make a move in each step). In an early manuscript [3] (written 1973), Dijkstra discusses his choice of the scheduler. Unsure, whether non-trivial algorithms for the distributed scheduler exist, he decides to design his self-stabilizing algorithms for a "rather powerful daemon, that may be awkward to implement": the central scheduler. He points out that his choice avoids difficulties like oscillation which may easily occur when using the distributed scheduler.

The model of the central scheduler provides rather strong assumptions. They make it easy to develop and prove correctness of self-stabilizing algorithms. On the other hand, the distributed scheduler is the more realistic model. Even though some algorithms that are designed for the central scheduler also stabilize under the distributed scheduler, the majority of algorithms does not have this property. In these cases new algorithms have to be devised or existing algorithms

* This research was funded by the German Research Foundation (DFG), contract number TU 221/3-1.

have to be extended. For the latter case, generic methods have been invented: transformers, for example Mutual Exclusion [1] and Conflict Managers [6]. It is considered worthwhile (for example in [1]) to design algorithms for the central scheduler and then transform them to the desired model. But these transformations come with a time overhead of at least $\mathcal{O}(\Delta)$.

In case no transformer is applied, the stabilization proofs are usually very problem-specific and do not allow for generalization. Generic proof techniques such as potential functions and convergence stairs, that work well for the central scheduler, are very hard to apply in case of the distributed scheduler. Section 4 provides a generic technique for proving stabilization under the distributed scheduler by extending existing proofs that assume the central scheduler. The contribution of this paper is completed by Section 5 in which the technique is applied to two complex algorithms for which stabilization under the distributed scheduler has not been proven and it has been an open question, whether this is feasible at all. Section 6 gives a case study of a protocol to which the proof technique cannot be applied.

2 Related Work

Common techniques for proving stabilization include variant functions (also called potential functions) and convergence stairs [4]. In principle, all these do apply to the distributed scheduler, but these techniques rely on properties of transitions from one system configuration to another. In case of the distributed scheduler, transitions are hard to analyse due to the large of number of possible selections of simultaneously executed moves.

Two transformers that allow algorithms written for the central scheduler to stabilize under the distributed scheduler are widely known. The first by Beauquier et al. [1] solves the mutual exclusion problem: (a) at least one process is privileged (b) only non-neighboring nodes are privileged (c) any node is privileged infinitely often. Mutual exclusion allows algorithms designed for a fair central scheduler to stabilize under an unfair distributed scheduler. The second by Gradinariu et al. [6] implements the concept of conflict managers: (a) at least one node is privileged (b) only non-conflicting nodes are privileged. Whether two nodes are conflicting is defined by the symmetric relation $\mathcal{R} \subseteq V \times V$ (the so-called conflict relation). If $\mathcal{R}$ is chosen such that neighboring nodes are conflicting, then this allows algorithms designed for the central scheduler to stabilize under the distributed scheduler.

Both transformers are partly based on the idea that the locally central scheduler (any number of non-neighboring nodes is privileged in each step) is virtually identical to the central scheduler, since the behaviour of a node only depends on the information stored in the node's neighborhood. Hence, moves of non-neighboring nodes can make their moves in an arbitrary order, or even simultaneously. The resulting configuration is always the same. It is rather restrictive to demand that any order of moves is equivalent to their simultaneous execution. For a given set of moves, it suffices to show that one particular sequence is

equivalent. Section 4 extends this idea to a technique, that allows to prove that such a sequence exists for any set of enabled moves in any configuration. The new proof-technique is directly applied to algorithms themselves, without any transformer.

Proving probabilistic self-stabilization under the distributed scheduler of protocols designed for the central scheduler is surprisingly easy. In the fashion of the scheduler-luck-game [4], one can show that steps in which only non-neighboring nodes simultaneously make a move exist with positive probability, if each move depends on the outcome of an independent random experiment. In conclusion, executions in which this is the case for every step exist with positive probability. This has facilitated the construction of the probabilistic conflict manager by Gradinariu et al. [6] and the transformations by Turau et al. [10] and Herman [7].

3 Model of Computation

A distributed system is represented as an undirected graph (V, E) where V is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges*. Let $n = |V|$ and Δ denote the maximal degree of the graph. If two nodes are connected by an edge, then they are called *neighbors*. The set of neighbors of node v is denoted by $N(v) \subseteq V$ and $N[v] = N(v) \cup \{v\}$. Each node stores a set of variables. The values of all variables constitute the *local state* of a node. The *configuration* of a system is the n-tuple of all local states in the system and Σ denotes the set of all configurations.

Nodes communicate via locally shared memory, that is every node can read the variables of all its neighbors. Nodes are only allowed to modify their own variables. Each node $v \in V$ executes a protocol consisting of a list of rules. Each rule consists of a *guard* and a *statement*. A guard is a Boolean expression over the variables of node v and its neighbors. A rule is called *enabled* if its guard evaluates to true. A node is called *enabled* if one of the rules is enabled.

Execution of the statements is controlled by a scheduler which operates in *steps*. At the beginning of step i, the scheduler first non-deterministically selects a non-empty subset $S_i \subseteq V$ of enabled nodes. Each node in S_i then executes the statement of its enabled rule. It is said, that the nodes make a *move*. A step is finished, if all nodes have completed their moves. Changes made during the moves become visible to other nodes at the end of the step and not earlier (composite atomicity).

An *execution* $\langle c_0, c_1, c_2, \ldots \rangle$, $c_i \in \Sigma$ is a sequence of configurations c_i where c_0 is the *initial configuration* and c_i is the configuration of the system after the i-th step. In other words, if the current configuration is c_i and all nodes in S_{i+1} make a move, then this yields c_{i+1}.

Time is measured in *rounds*. Let x be an execution and $x_0 = x$. Then x is partitioned into rounds by induction on $i = 0, 1, 2, \ldots$: round r_i is defined to be the minimal prefix of x_i, such that each node $v \in V$ has either made a move or has been disabled at least once within r_i. Execution x_{i+1} is obtained by removing prefix r_i from x_i. The intuition is that within a round, each node

that is enabled at the beginning of the round, gets the chance to make a move if it has not become disabled by a move of its neighbors.

Let P be a protocol and let $Legit_P$ denote a Boolean predicate over Σ. If $Legit_P(c)$ is true, then configuration c is called *legitimate*. P is said to be *self-stabilizing* with respect to $Legit_P$, if both the following properties are satisfied. *Convergence*: for any execution of P, a legitimate configuration is reached within a finite number of steps. *Closure*: for any execution of P it holds that once a legitimate configuration is reached, all subsequent configurations are also legitimate. A self-stabilizing protocol is *silent* if all nodes are disabled after a finite number of steps.

3.1 Nonstandard Extensions

The standard model as defined above is extended to a *multi-protocol* model. An *algorithm* is denoted by a set $\mathcal{A}$ of protocols. Each node designates a separate set of variables to each $p \in \mathcal{A}$. An *instance* is a tuple (v, p), $v \in V$ and $p \in \mathcal{A}$. $\mathcal{M} = V \times \mathcal{A}$ is the set of instances. Instance (v, p) is called *enabled*, if p is enabled on node v. Node v is called *enabled*, if any $p \in \mathcal{A}$ is enabled on v. Two instances (v_1, p_1) and (v_2, p_2) are called *neighboring*, if $v_1 \in N[v_2]$ or vice versa.

During the i-th step, a scheduler selects a subset $S_i \subseteq \mathcal{M}$ of enabled instances. In case of the central scheduler it holds $|S_i| = 1$. The distributed scheduler may chose any non-empty subset S_i of enabled instances, even containing several instances of the same node but distinct protocols. If node v executes a rule by protocol $p \in \mathcal{A}$, then it is said that v has made the move (v, p). No assumptions on the fairness of the scheduler are made.

An instance (v, p) cannot modify any other variables than the ones designated to protocol p by node v. Read access is permitted to all variables of all $v \in N[v]$, no matter which protocol they belong to. For any pair of instances (v_1, p_1) and (v_2, p_2) that are being selected during a single step, changes made by (v_1, p_1) don't become visible to (v_2, p_2) until the end of the step (composite atomicity), even if $v_1 = v_2$. Due to these constraints, the result of a step does not depend on the order in which the individual moves are executed. This model defines a natural extension of the notion of rounds. A *round* is a prefix of an execution, such that every instance $m \in \mathcal{M}$ has been executed or has been disabled at least once. This model is identical to the standard model, if $|\mathcal{A}| = 1$. How algorithms designed for this model can be transformed to the standard single-protocol model is discussed in Section 5.3.

Without loss of generality, it is assumed that per instance only one guard can be enabled at a time and that all rules are deterministic. How to widen the techniques to a non-deterministic or randomized model is discussed in the concluding remarks. Using the assumption of deterministic protocols, the following notations are defined: $(c : m)$ denotes the configuration after the execution of $m \in \mathcal{M}$ in c. Similarly, $(c : S)$ denotes the configuration after the simultaneous execution of all instances $S \subseteq \mathcal{M}$ in a single step. The execution of a sequence of instances is denoted by $(c : m_1 : m_2 : \ldots : m_x) = ((c : m_1 : m_2 : \ldots : m_{x-1}) : m_x)$, $m_i \in \mathcal{M}$, $x > 1$. These notations are used to describe executions by the central scheduler

or the distributed scheduler. Note that $(c : m)$ and $(c : S)$ are undefined, if m is not enabled in c or if S contains instances that are not enabled in c respectively.

Furthermore, let $c|_m$ denote the part of configuration c which reflects the values of all variables dedicated to instance m. The expression $c \vdash e$ denotes the value of expression e in case that the current configuration equals c. Note, that e can be a variable, function or Boolean predicate.

4 Serialization

To proof stabilization under the distributed scheduler, we first define the notion of a serialization. A serialization of a set of enabled instances is a sequence of instances, that can be executed under the central scheduler and yields the same configuration as executing the set of instances during a single step of the distributed scheduler.

Definition 1. *Let c be a configuration and $S \subseteq \mathcal{M}$ be a set of instances enabled in c. A sequence $s = \langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{M}$ is called a* ***serialization*** *of S in c if it satisfies*

$$(c : S) = (c : m_1 : m_2 : \ldots : m_k) \tag{1}$$

S is called ***serializable*** *in c, if a serialization in c exists.*

With respect to S, the serialization contains each instance $m_i \in S$ at least once. The simple reason is, that the serialization is required to modify $c|_{m_i}$, which no instance other than m_i is capable of. Apart from that, instances may occur multiple times within the sequence. Even additional instances that are not in S may be included, but their effect has to be compensated such that Equation (1) holds again in the end.

Observation 2. The sequence $\langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{M}$ is a serialization of $S \subseteq \mathcal{M}$ in c if and only if

$$(c : S)|_{m_i} = (c : m_1 : m_2 : \ldots : m_k)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

If $m_1, m_2, \ldots, m_k$ are distinct, then this is equivalent to

$$(c : S)|_{m_i} = (c : m_1 : \ldots : m_i)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

Furthermore, it is clear that $(c : S)|_{m_i} = (c : m_i)|_{m_i}$.

The rest of this section lays the groundwork for a technique that facilitates the construction of serializations. First, the notion of a ranking is defined. It assigns a natural number (the rank) to each enabled instance. The rank describes the behaviour of an instance (i.e. how it changes the variables) depending on the current configuration. The goal is to obtain a serialization by sorting instances by their rank.

Definition 3. *The mapping* $r : \mathcal{M} \to \mathbb{N} \cup \{\bot\}$ *is called a* ***ranking****, if the following conditions hold for any configuration:*

$$r(m) = \bot \text{ if instance } m \text{ is disabled}$$
$$r(m) \in \mathbb{N} \text{ otherwise}$$

For a given ranking it remains to show that sorting a set of instances by their rank actually yields a serialization. As a step towards this, an invariancy relation on instance/rank-tuples is defined. In general, this relation is not symmetric.

Definition 4. *Let* r *denote a ranking. A tuple* $(m_2, r_2) \in \mathcal{M} \times \mathbb{N}$ *is called* ***invariant*** *under the tuple* $(m_1, r_1) \in \mathcal{M} \times \mathbb{N}$*, if the following two conditions hold for all* $c \in \Sigma$ *that satisfy* $r_1 = c \vdash r(m_1)$ *and* $r_2 = c \vdash r(m_2)$*:*

$$(c : m_1) \vdash r(m_2) = c \vdash r(m_2)$$
$$(c : m_1 : m_2)|_{m_2} = (c : m_2)|_{m_2}$$

If the tuple (m_2, r_2) is invariant under (m_1, r_1), then the rank of m_2 as well as the result of the execution of m_2 remains the same, no matter whether m_1 has been executed prior to m_2, or not. Note, that this invariancy holds for all configurations, in which m_2 and m_1 have the given ranks r_2 and r_1 respectively. The following proposition illustrates, why this is useful.

Proposition 5. *Let* c *be a configuration,* r *a ranking, and* m_1*,* m_2*,* m_3 *three distinct enabled instances. If* $(m_2, c \vdash r(m_2))$ *is invariant under* $(m_1, c \vdash r(m_1))$ *and* $(m_3, c \vdash r(m_3))$ *is invariant under both* $(m_1, c \vdash r(m_1))$ *and* $(m_2, c \vdash r(m_2))$*, then* $\langle m_1, m_2, m_3 \rangle$ *is a serialization of* $\{m_1, m_2, m_3\}$ *in* c*.*

Proof. By Observation 2 it suffices to prove the following three equations:

$$(c : m_1)|_{m_1} = (c : m_1)|_{m_1} \tag{2}$$
$$(c : m_2)|_{m_2} = (c : m_1 : m_2)|_{m_2} \tag{3}$$
$$(c : m_3)|_{m_3} = (c : m_1 : m_2 : m_3)|_{m_3} \tag{4}$$

Equation (2) is clear. Equation (3) holds, because $(m_2, c \vdash r(m_2))$ is invariant under $(m_1, c \vdash r(m_1))$. In order to understand the validity of Equation (4) it is necessary to take a closer look at the sequential execution of m_1, m_2, and m_3. Consider the intermediate configuration $c' = (c : m_1)$. Because $(m_3, c \vdash r(m_3))$ is invariant under $(m_1, c \vdash r(m_1))$, it holds that $(c' : m_3)|_{m_3} = (c : m_3)|_{m_3}$. In order for Equation (4) to be satisfied, it must be the case that $(c' : m_2 : m_3)$ equals $(c' : m_3)$. This is true, if $(m_3, c' \vdash r(m_3))$ is invariant under $(m_2, c' \vdash r(m_2))$. This becomes clear, if one considers that $c' \vdash r(m_3) = c \vdash r(m_3)$ as well as $c' \vdash r(m_2) = c \vdash r(m_2)$ and that the invariancy of m_3 under m_2 holds no matter whether the current configuration is c or c'. □

Definition 6. *A ranking* r *is called an* ***invariancy-ranking****, if*

$$r_2 \geq r_1 \Rightarrow (m_2, r_2) \text{ is invariant under } (m_1, r_1)$$

holds with respect to r *for all* $m_2 \neq m_1$*,* $m_2, m_1 \in \mathcal{M}$*,* $r_2, r_1 \in \mathbb{N}$*.*

Theorem 7. *For an algorithm with an invariancy-ranking, every set of enabled instances is serializable in any configuration.*

Proof. Let r be an invariancy-ranking, c a configuration, $S \subseteq \mathcal{M}$ a set of instances enabled in c, and $s = \langle m_1, m_2, \ldots, m_k \rangle$ a sequence of all instances of S sorted in ascending order by their rank with respect to c. Denote by c_x the configuration $(c : m_1 : m_2 : \ldots : m_x)$ and $c_0 = c$. By Observation 2 it suffices to prove $c_j|_{m_j} = (c_0 : m_j)|_{m_j}$ for $j = 1, 2, \ldots, k$.

In the following, it is shown by induction on i that $(c_{i-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ and $c_{i-1} \vdash r(m_j) = c_0 \vdash r(m_j)$ hold for all $i = 1, 2, \ldots, k$ and $j = i, i+1, \ldots, k$. This is obviously true for $i = 1$. Assume the following for $i < k$:

$$\begin{aligned} c_{i-1} \vdash r(m_j) &= c_0 \vdash r(m_j) && \forall j = i, i+1, \ldots, k \\ (c_{i-1} : m_j)|_{m_j} &= (c_o : m_j)|_{m_j} && \forall j = i, i+1, \ldots, k \end{aligned}$$

By assumption, $(m_j, c_{i-1} \vdash r(m_j))$ is invariant under $(m_i, c_{i-1} \vdash r(m_i))$ for all $j = i+1, i+2, \ldots, k$. Hence, the following is satisfied in c_i:

$$\begin{aligned} c_i \vdash r(m_j) = c_{i-1} \vdash r(m_j) &= c_0 \vdash r(m_j) && \forall j = i+1, i+2, \ldots, k \\ (c_i : m_j)|_{m_j} = (c_{i-1} : m_j)|_{m_j} &= (c_0 : m_j)|_{m_j} && \forall j = i+1, i+2, \ldots, k \end{aligned}$$

In particular, it follows that $c_j|_{m_j} = (c_{j-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ for all $j = 1, 2, \ldots, k$. □

Corollary 8. *For any execution e under the distributed scheduler of an algorithm with an invariancy-ranking, there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

Theorem 9. *For an algorithm with an invariancy-ranking, move- and round-complexity under the central scheduler are upper bounds for the move- and round-complexity under the distributed scheduler.*

Proof. See Appendix. □

5 Practicability

In the following, the new proof technique is applied to two algorithms, one from [5] and the other from [8]. Both algorithms are transformers that add fault-containing properties to any silent self-stabilizing protocol P. Prior to this paper, it has been an open problem whether these algorithms work under the distributed scheduler.

The overall procedure to first design a ranking of which it is shown that it is an invariancy-ranking. This is done by inspecting all pairs (r_2, r_1) of ranks that satisfy $r_2 \geq r_1$. For each pair, it is shown that all (m_2, r_2) are invariant under any (m_1, r_1). More m_2 is called *invariant* under m_1, if (m_2, r_2) is invariant under (m_1, r_1) for all ranks $r_2, r_1 \in \mathbb{N}$. The following observation justifies, that the proofs only consider the case that m_2 and m_1 are neighboring.

Observation 10. Let r be a ranking, and let m_2 and m_1 be two non-neighboring moves. If $r(m_2)$ solely depends on variables in the neighborhood of m_2, then m_2 is invariant under m_1.

5.1 Algorithm $\mathcal{A}_1$

First, algorithm $\mathcal{A}_1 = \{Q\}$ by Ghosh et al. [5] is analysed. The algorithm is actually a transformer, that is protocol Q internally calls a given silent self-stabilizing protocol P and extends P by fault-containment properties. The goal of fault-containment is to minimize the time that a protocol needs to recover from small scale faults. This property is combined with self-stabilization which guarantees recovery from large scale faults. Protocol Q basically is a silent phase clock, with a limited clock range of $[0, M]$. During stabilization, the timestamps are decremented towards 0 in an evenly fashion until every node has reached 0. Along with this decrementation, three protocols are executed: C, P, and B. Protocol C is executed for upper range timestamps and repairs corrupted P-variables using backups stored on each neighbor. It consists of three phases that are executed by Q in a synchronous fashion. For mid-range timestamps, P is executed by Q and is given enough time to stabilize before B, which is executed for lower range timestamps, creates the backups. Timestamps are globally reset to M if an inconsistency or a fault is detected. For details refer to [5].

Protocol Q is implemented in form of two rules. Each node $v \in V$ stores its timestamp in the variable $v.t$. Rule S_1 resets $v.t$ to M, if $\mathit{PorC\_inconsistent}(v)$ or $\mathit{raise}(v)$ is satisfied.

$$
\begin{aligned}
&\mathit{raise}(v) \equiv \mathit{raise}_1(v) \vee \mathit{raise}_2(v)\\
&\mathit{raise}_1(v) \equiv v.t \neq M \wedge \exists u \in N(v) : v.t - u.t > 1 \wedge u.t < M - n\\
&\mathit{raise}_2(v) \equiv v.t < M - n \wedge \exists u \in N(v) : u.t = M\\
&\mathit{PorC\_inconsistent}(v) \equiv v.t = 0 \wedge (\forall u \in N(v) : u.t = 0)\\
&\qquad\qquad\qquad\qquad \wedge (G_P(v) \vee \neg \mathit{Legit}_C(v))
\end{aligned}
$$

The predicate $G_P(v)$ is true if and only if P is enabled on v. The predicate $\mathit{Legit}_C(v)$ is true if and only if any backup differs from the current values of P-variables. If $\mathit{decrement}(v)$ is true, rule S_2 first executes a move of C if $v.t \in [M-2, M]$, a move of P if $v.t \in [3, M - \max\{n, 3\}]$, or a move of B if $v.t = 2$ and then decrements $v.t$ by one. The local state of protocol P for node v is held in the variable $v.x$ which is called *primary state*.

$$
\begin{aligned}
&\mathit{decrement}(v) \equiv \mathit{decrement}_1(v) \vee \mathit{decrement}_2(v)\\
&\mathit{decrement}_1(v) \equiv v.t > 0 \wedge \forall u \in N(v) : 0 \leq v.t - u.t \leq 1\\
&\mathit{decrement}_2(v) \equiv \forall u \in N(v) : v.t \geq u.t \wedge u.t \geq M - n
\end{aligned}
$$

To obtain serializations, the following ranking is used:

$$
r(v,p) := \begin{cases} 0 & \text{if } \mathit{decrement}(v)\\ M - v.t & \text{if } \mathit{raise}(v) \vee \mathit{PorC\_inconsistent}(v)\\ \bot & \text{otherwise} \end{cases}
$$

Note, that $raise(v) \vee PorC\_inconsistent(v)$ implies $v.t < M$ and thereby $r(v, Q) > 0$. So all instances of rank 0 decrement $v.t$ and all others reset it to M.

The ranking r is designed in such a way that decrementations occur first within a serialization. Their order is not significant since a decrement move does not disable any neighboring instances. This is due to the pseudo-consistence criteria as defined in [5]. Next, all raise moves occur within the serialization, sorted by their timestamp in descending order. The following example illustrates why this is necessary: Consider a node v which is surrounded by nodes with a timestamp equal to 0 while $v.t = M - n$. In this configuration, node v is enabled by $raise_1(v)$. It is possible, that all neighbors of v are enabled by $raise_2(v)$. If the neighbors of v make a raise move before v does, then $v.t$ becomes pseudo-consistent, and hence v becomes disabled.

Observation 11. $decrement_1(v)$ as well as $decrement_2(v)$ imply $v.t \geq u.t$ for all $u \in N(v)$. So if $decrement(v)$ and $decrement(u)$ hold for two neighboring nodes v and u, then $v.t = u.t$ is true.

Lemma 12. *Assume that there exists an invariancy-ranking for each of the protocols P, C and B. Spreading rank 0 of r according to these invariancy-rankings (and shifting the higher ranks accordingly) yields an invariancy-ranking for algorithm $\mathcal{A}_1$.*

Proof. In the following, m_2 and m_1 denote moves by nodes v_2 and v_1 respectively. c denotes a configuration such that $r_2 = c \vdash r(m_2)$ and $r_1 = c \vdash r(m_1)$. Furthermore, c' denotes the configuration $(c : m_1)$. Instances m_2 and m_1 are assumed to be neighboring. This is justified by Observation 10.

Case a) $r_2 = r_1 = 0$: The assumption yields $c \vdash decrement(v_2)$ and $c \vdash decrement(v_1)$. From that and Observation 11 it follows that $c \vdash v_1.t = v_2.t$ and $c' \vdash v_1.t = v_2.t - 1$. If $c \vdash decrement_1(v_2)$, then $c' \vdash decrement_1(v_2)$. Otherwise $c \vdash (decrement_2(v_2) \wedge \neg decrement_1(v_2))$ from which $c \vdash v_2.t > M - n$ follows and thereby $c' \vdash v_1.t \geq M - n$ and $c' \vdash decrement_2(v_2)$.

Case b) $1 \leq r_2 \leq M \wedge r_1 = 0$: Because of $r_1 = 0$, $c \vdash decrement(v_1)$ and thus $c \vdash v_1.t > 0$ must hold. $c \vdash PorC\_inconsistent(v_2)$ cannot be satisfied, since it requires $c \vdash v_1.t = 0$. If $c \vdash raise_1(v_2)$, then there exists some node $w \in N(v_2)$ with $c \vdash (v_2.t - w.t > 1 \wedge w.t < M - n)$. $c \vdash w.t < N - n$ implies $c \vdash \neg decrement_2(w)$ and $c \vdash w.t - v_2.t < -1$ implies $c \vdash \neg decrement_1(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_1(v_2)$. If $c \vdash raise_2(v_2)$, then $c \vdash v_2.t < M - n$ and there exists some node $w \in N(v_2)$ with $c \vdash w.t = M$. $c \vdash w.t - v_2.t > 1$ implies $c \vdash \neg decrement_1(w)$ and $v_2.t < M - n$ implies $v \vdash \neg decrement_2(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_2(v_2)$.

Case c) $1 \leq r_2 \leq M \wedge 1 \leq r_1 \leq r_2$: If $c \vdash v_2.t < M - n$, then $c' \vdash raise_2(v_2)$ since $c' \vdash v_1.t = M$. Otherwise $c \vdash v_2.t \geq M - n$ which implies $c \vdash \neg raise_2(v_2)$ and $c \vdash \neg PorC\_inconsistent(v_2)$. and thus $c \vdash raise_1(v_2)$. Hence there exists some node $w \in N(v_2)$ with $c \vdash (w.t < v_2.t \wedge w.t < M - n)$. From $r_1 \leq r_2$ it follows that $c \vdash v_1.t \geq v_2.t$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_1(v_2)$.

In all cases $c' \vdash r(m_2) = c \vdash r(m_2)$. Furthermore, it is assumed that the moves of rank 0 are sorted in such a way that the order of m_2 and m_1 matches

a serialization of either protocol P, C or B, depending on $v_1.t$ and $v_2.t$ which are equal by Observation 11. Hence $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$ in all cases. □

Theorem 13. *If there exists an invariancy-ranking for each of the protocols P, C and B, then for any execution e of $\mathcal{A}_1$ under the distributed scheduler there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

5.2 Algorithm $\mathcal{A}_2$

In this section, algorithm $\mathcal{A}_2 = \{Q, R_1, R_2, \ldots, R_\Delta\}$ by Köhler et al. [8] is discussed. It implements a different approach to add fault-containment to any given silent self-stabilizing protocol P. It offers several improvements over algorithm $\mathcal{A}_1$: a constant fault-gap (that is the minimal time between two containable faults), strictly local fault repair without any global effects, and a stabilization time similar to the original protocol P (besides a constant slow-down factor).

Every node $v \in V$ designates one of the Δ protocols R_i to each of its neighbors. For notational convenience, the algorithm is defined to be a mapping $\mathcal{A}_2 : v \mapsto \{Q\} \cup \{R_u \mid u \in N(v)\}$ that assigns a set of protocols to each $v \in V$. In spite of this notation, algorithm $\mathcal{A}_2$ is still uniform. The behaviour of the algorithm is best described based on the notion of a *cell*. For $v \in V$, cell v consists of the instance (v, Q) and the instances (u, R_v), $u \in N(v)$. Cells execute cycles of a simple finite state-machine. During a cycle, cells first repair corrupted variables. The cycle is always guaranteed to start with the repair, even after a fault. Cells then check, whether there are any corruptions in their neighboring cells and if so, they wait for them to become repaired. Only after that, a single move of P is executed and as a final step backups of the variables of protocol P are created. To achieve this behaviour, there is a constant *dialog* between (v, Q) and all instances of R_v. For details refer to [8].

The instance (v, Q) maintains three variables: $v.s, v.q \in \mathbb{Z}_4$ and $v.p$, which is also called *primary state* and stores the local state of protocol P for node v. Each of the variables $v.s$ and $v.q$ can assume one of the four states $0 =$ Paused, $1 =$ Repaired, $2 =$ Executed, and $3 =$ Copied. If $v.s \neq v.q$, then $(v.s, v.q)$ is called a *query* for a transition from state $v.s$ to $v.q$. An instance (u, R_v) maintains the following variables: $u.r_v \in \mathbb{Z}_4$, $u.d_v \in \mathbb{Z}_3$, and $u.c_v$. One of the four state values is assigned $u.r_v$. The so-called *decision-variable* $u.d_v$ assumes one of the values Keep, Update, and Single. The *copy-variable* $u.c_v$ is used for storing backups of $v.p$.

Protocol Q consists of three rules. If $\neg$*dialogConsistent*(v), then instance (v, Q) resets $v.s$ and $v.q$ to Paused if they do not already have that value (Rule 1). If *dialogPaused*$(v) \wedge$ *startCond*$_Q(v)$, then (v, Q) sets $v.q$ to Repaired (Rule 2). If *dialogAcknowledged*(v), then (v, Q) calls procedure *action*$_Q(v)$, sets $v.s$ to $v.q$, and if $v.q \neq$ Paused, then $v.q$ is incremented (Rule 3). Note, that only one guard of (v, Q) can be true at a time. Protocol R_v consists two rules. If $v.s$ and $v.q$ equal Paused, then instance (u, R_v) sets $u.r_v$ to Paused if it

doesn't have that value already (Rule 1). If $validQuery(v)$ is true and predicate $waitCond_{R_v}(u)$ is false, then (u, R_v) sets $u.r_v$ to $v.q$ and calls procedure $action_{R_v}(u)$ (Rule 2). Again, only one guard of (u, R_v) can be true at a time.

$$
\begin{aligned}
&validQuery(v) \equiv v.q = (v.s + 1) \bmod 4\\
&dialogConsistent(v) \equiv (v.q = v.s = \textsc{Paused} \vee validQuery(v))\\
&\qquad\qquad \wedge \forall u \in N(v) : u.r_v \in \{v.s, v.q\}\\
&dialogAcknowledged(v) \equiv validQuery(v) \wedge \forall u \in N(v) : u.r_v = v.q\\
&dialogPaused(v) \equiv v.q = v.s = \textsc{Paused} \wedge \forall u \in N(v) : u.r_v = \textsc{Paused}\\
&copyConsistent(v) \equiv \forall u \in N(v) : u.c_v = v.p\\
&repaired(v) \equiv copyConsistent(v) \vee (dialogConsistent(v)\\
&\qquad\qquad \wedge (v.s = \textsc{Repaired} \vee v.s = \textsc{Executed})\\
&\qquad\qquad \wedge \forall u \in N(v) : (u.r_v = \textsc{Copied} \Rightarrow u.c_v = v.p))\\
&startCond_Q(v) \equiv \neg copyConsistent(v) \vee G_P(v)\\
&waitCond_{R_v}(u) \equiv v.q = \textsc{Executed} \wedge \neg repaired(u)
\end{aligned}
$$

Procedure $action_Q(v)$ performs the following actions: If $v.q = \textsc{Executed}$, then a move of protocol P is executed. If $v.q = \textsc{Repaired}$, then $v.p$ is checked for corruptions by using the backups provided by protocol R_v. If all backups have the same value and $v.p$ differs, then $v.p$ is updated. If there is only one neighbor $u \in N(v)$ and hence only one backup, then $v.p$ is only updated if $u.d_v = \textsc{Update}$. Procedure $action_{R_v}(u)$ performs the following: If $v.q = \textsc{Copied}$, then $u.c_v$ is updated with the value of $v.p$. If $v.q = \textsc{Repaired}$, then $u.d_v$ is set to either Keep or Update depending on whether $v.p := u.c_v$ would disable P on both u and v.

The case that the network contains a single edge only is not covered by the above description of $action_Q$ and $action_{R_v}$. In case of the distributed scheduler, it needs special treatment like symmetry breaking which is not included in the original version as given in [8]. The following describes a possible solution: Let u and v be neighbors and the only nodes of the system. The special value Single, which is assigned to $v.d_u$ and $u.d_v$ in this case, allows the detection of this case. The version of protocol $action_Q$ as given [8] would execute both of the assignments $v.p := u.p$ and $u.p := v.p$ if both v and u are selected in the same step by the distributed scheduler. Either one of the two assignments leads to a legitimate configuration, but in most cases the execution of both does not. Let v be the node with the lower Id. $action_Q$ can be altered in such a way that $v.p := u.p$ is not executed if $u.p := v.p$ leads to a successful repair.

The following ranking r is used to obtain serializations. For convenience, the predicate $R(x)$ is used in the definition of $r(v, p)$. It is satisfied if and only if

rule x of instance (v, p) is enabled.

$$r(v,p) := \begin{cases} 1 & \text{if } \exists u \in N(v) : p = R_u \wedge R(2) \wedge u.q = \textsc{Repaired} \\ 2 & \text{if } \exists u \in N(v) : p = R_u \wedge ((R(2) \wedge u.q \neq \textsc{Repaired}) \vee R(1)) \\ 3 & \text{if } p = Q \wedge R(2) \\ 4 & \text{if } p = Q \wedge R(3) \wedge v.q = \textsc{Executed} \\ 5 & \text{if } p = Q \wedge ((R(3) \wedge v.q \neq \textsc{Executed}) \vee R(1)) \\ \bot & \text{otherwise} \end{cases}$$

With this ranking, any serialization will first execute all moves, that set decision variables. Their value is determined by the evaluation of guards of protocol P which references the primary states of various cells. Hence it is important, that the primary states have not been changed yet (which is only done by instances of rank 4, or 5). Next, all all other moves by instances of protocol R_v occur within the serialization. These do not reference any of the variables that have been changed previously. All instances of Q follow. They are categorized into three different ranks. Instances that are enabled due to rule 2 of Q are executed first to avoid the danger that this rule becomes disabled which is due to $startCond_Q(v)$ that checks whether protocol P is enabled for node v. Instances of rank 4 follow. They execute a single move of P and hence may change primary states. The fact that all moves of P fall into the same rank has the advantage, that invariancy-rankings for protocol P can be used to extent r to an invariancy ranking for algorithm $\mathcal{A}_2$. Last, all instances of rank 5 occur within the serialization. These moves do not read any primary states and their behaviour is hence not influenced by any of the changes done by previous instances of Q.

Lemma 14. *If cell v is not dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Lemma 15. *If cell v is dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Lemma 16. *Let v be a cell. (v, Q) is invariant under any (u, R_w) with $u \in N[v]$ and $w \neq v$.*

Lemma 17. *Assume, that an invariancy-ranking for protocol P exists. Spreading rank 4 of r according to this invariancy-ranking (and shifting the higher ranks accordingly) yields an invariancy-ranking for algorithm $\mathcal{A}_2$.*

Proof. The proof of case $r_2 \in \{3, 4, 5\} \wedge r_1 \in \{1, 2\}$ is based on Lemmas 14, 15, and 16. A detailed proof of those lemmas and a proof covering all other cases of r_2 and r_1 is given in the Appendix. □

Theorem 18. *If there exists an invariancy-ranking for protocols P, then for any execution e of $\mathcal{A}_2$ under the distributed scheduler there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

5.3 Proof refinement

The results of Theorems 13 and 18 do not only guarantee stabilization under the distributed scheduler. They guarantee, that the algorithms behave exactly as under the central scheduler, in all aspects – for example with respect to time-complexity and fault-containment properties. Yet, Theorems 13 and 18 require that an invariancy-rankings for protocols P, C, and B exist. This requirements can be relaxed.

Protocol C must show correct behaviour only if the initial configuration is 1-faulty. Such configurations are derived from a legitimate configuration by perturbing variables of a single node. Indeed, an invariancy-ranking can be found under these assumptions. For any other initial configuration, C may show an arbitrary behaviour. The invariancy ranking for B is simply $r_B(v, p) := 0$. Protocol B never reads the variables that it writes.

With respect to P, both $\mathcal{A}_1$ and $\mathcal{A}_2$ solely rely on the property that P terminates after a finite number of steps of the given scheduler. There is no other requirement concerning the behaviour of P. For each single step $S_i \subseteq \mathcal{M}$ of the distributed scheduler, the sequence obtained by sorting S_i by the rankings given above yields a configuration that differs from the execution of S_i only in the primary states of those nodes that make a P-move during the execution of S_i. In addition, it can be shown that P successfully progresses towards its termination during the execution of S_i.

Furthermore, it is easy to transform a given algorithm $\mathcal{A} = \{p_1, p_2, \ldots, p_k\}$ for the multi-protocol model into an algorithm $\mathcal{A}_s = \{q\}$ for the ordinary single-protocol model. In [8], a composition is used for the case of the central scheduler. The idea is to sequentially execute the individual p_i within a single move of q. Due to the nature of the central scheduler as defined in the multi-protocol model it is not a problem that the changes made by p_i become visible to p_{i+1} immediately. In case of the distributed scheduler, a different transformation is needed. Again, all of the p_i are executed sequentially during a single move of q. But to emulate composite atomicity, the changes made by any p_i are hidden from any p_j, $j \neq i$ until the end of the move of q. This may cost some memory overhead which can be avoided by finding a special invariancy-ranking only for serializing instances on a single node. In that style, the protocols of algorithm $\mathcal{A}_2$ can be executed in the order $\langle R_1, R_2, \ldots, R_\Delta, Q\rangle$. We would like to emphasize, that one round of $\mathcal{A}_s$ is equivalent to one round of $\mathcal{A}$.

6 Impossibility

Unfortunately, serializations do not always exist. This is the case for the MIS-protocol proposed in [9] which has been especially designed for the distributed scheduler. The proof of stabilization for the distributed scheduler is not straight forward [9]. In the following, the basic obstacles that prevent the application of the new proof technique are explained.

The protocol assigns one of the three states OUT, WAIT, and IN to each node. If a node is in state OUT and does not have a neighbor in state IN, then

its state is changed to IN (Rule 1). A node in state WAIT changes its state to OUT, if it has a neighbor in state IN (Rule 2). A node in state WAIT changes its state to IN, if it does not have a neighbor in state IN and all neighbors in state WAIT have a higher Id (Rule 3). A node in state IN switches to state OUT, if it has a neighbor in state IN (Rule 4).

As a first example, consider two neighboring nodes v and u, both in state IN, while all their neighbors are in state OUT. If the distributed scheduler selects both u and v during a single step, then both simultaneously switch to state to OUT. Note, that the state of u is the only reason that v is enabled and vice versa. Under the central scheduler, one of the two nodes becomes disabled after the first move and remains in state IN. Hence no serialization exists.

Now imagine that v is in state WAIT and u is in state OUT, while all their neighbors are in state OUT again. Furthermore, assume that the Id of node u is higher then the Id of v. If the distributed scheduler selects both v and u during a single step, then v sets its state to IN by Rule 3 and u switches to state WAIT by Rule 1. Again, there is no serialization because a move by v disables u and vice versa.

For the sake of optimization, [9] proposes a modified version of Rule 4. Rule 4′ only sets the state to OUT, if there is an IN-neighbor with a lower Id. This allows serializations of the first example by sorting the moves in descending order by the Ids. The node with the lowest Id serves as a "final cause" of the moves by the other nodes. Furthermore, it is possible to modify Rule 1 in such a way that nodes only switch from OUT to WAIT, if there is no WAIT-neighbor with a lower Id. Then the second example becomes serializable as well. Obviously, in order for serializations to exist, situations in which moves disable moves of neighboring nodes must be avoided or at least, it must be possible to resolve these conflicts by sorting.

7 Concluding Remarks

This paper has described a new technique for proving self-stabilization under the distributed scheduler. The task of proving self-stabilization is reduced to the task of finding an invariancy-ranking. The proof that a given ranking is indeed an invariancy-ranking is solely based on properties of sequential executions of pairs of moves under the central scheduler. The new technique has been successfully applied to two algorithms. Even more, Corollary 8 guarantees, that all properties of the algorithms are preserved. In particular, by Theorems 13 and 18, the two algorithms are the first transformers for adding fault-containment to self-stabilizing protocols that are known to work under the distributed scheduler. It has also been discussed that algorithms exist which stabilize under the distributed scheduler, but for which it is impossible to find serializations. Furthermore, rankings may exist that yield serializations but are not invariancy-rankings. We're not aware of any examples for the latter.

It remains to be investigated, how serializations can be found other than simply by sorting moves. Instances may occur multiple times or additional instances

may be included in the serialization. Randomized protocols or non-deterministic choices by the scheduler are not a problem. Both issues can be solved by extending the instance tuple with information about the scheduler's choice (the rule number) and the outcome of the random experiment during the move. This way, the information becomes part of the ranking. Another possibility is to generalize the notation $(c : m)$ to $\{c : m\}$ which denotes the set of configurations reachable from $c \in \Sigma$ by the execution of an instance $m \in \mathcal{M}$.

References

1. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. In: Proceedings of the 14th International Conference on Distributed Computing. Lecture Notes in Computer Science, vol. 1914, pp. 223–237. Springer, Berlin, Germany (2000) 2
2. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974) 1
3. Dijkstra, E.W.: EWD 391, self-stabilization in spite of distributed control. In: Selected writings on computing: a personal perspective, pp. 41–46. Springer, Berlin, Germany (1982), originally written in 1973 1
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge, MA, USA (2000) 2, 3
5. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing 20(1), 53–73 (2007) 7, 8, 9
6. Gradinariu, M., Tixeuil, S.: Conflict managers for self-stabilization without fairness assumption. In: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems. p. 46. IEEE Computer Society, Los Alamitos, CA, USA (2007) 2, 3
7. Herman, T.: Models of self-stabilization and sensor networks. In: Proceedings of the 5th International Workshop on Distributed Computing. Lecture Notes in Computer Science, vol. 2918, pp. 205–214. Springer, Berlin, Germany (2003) 3
8. Köhler, S., Turau, V.: Fault-containing self-stabilization in asynchronous systems with constant fault-gap. In: Proceedings of the 30th IEEE International Conference on Distributed Computing Systems. pp. 418–427. IEEE Computer Society, Los Alamitos, CA, USA (2010) 7, 10, 11, 13, 16
9. Turau, V.: Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. Information Processing Letters 103(3), 88–93 (2007) 13, 14
10. Turau, V., Weyer, C.: Fault tolerance in wireless sensor networks through self-stabilization. International Journal of Communication Networks and Distributed Systems 2(1), 78–98 (2009) 3

Appendix

Let c denote a configuration. The *changeset* of $(c : m)$ is the set of variables that have changed between c and $(c : m)$.

Proof (Lemma 14). Let c be a configuration, in which both $m_2 = (v, Q)$ and $m_1 = (u, R_v)$ are enabled. Let c' denote the configuration $(c : m_1)$. Rules 2 and

3 of m_2 are disabled in c since they require *dialogConsistent*(v). Only rule 1 of m_2 is enabled in c and hence $c \vdash (v.s \neq \textsc{Paused} \vee v.q \neq \textsc{Paused})$. In conclusion, rule 1 of m_1 is disabled and only rule 2 is enabled in c. Thus $c \vdash (\mathit{validQuery}(v) \wedge u.r_v = v.s)$. From this and $c \vdash \neg \mathit{dialogConsistent}(v)$ it follows, that a node $w \in N(v)$ exists for which $c \vdash w.r_v \notin \{v.s, v.q\}$. Hence $w \neq u$ and $c'|_w = c|_w$ which implies $c' \vdash \neg \mathit{dialogConsistent}(v)$ and $c' \vdash r(m_2) = c \vdash r(m_2)$. Rule 1 of m_2 sets both $v.s$ and $v.q$ to Paused. Hence $(c' : m_2)|_{m_2} = (c, m_2)|_{m_2}$. □

Proof (Lemma 15). It is shown, that no configuration exists, in which both (v, Q) and (u, R_v) are enabled. If *dialogConsistent*(v), then rule 1 of (v, Q) and rule 1 of (u, R_v) are disabled. If rule 2 of (u, R_v) is enabled, then *validQuery*(v) and $u.r_v = v.s$ holds. Hence, neither *dialogPaused*(v) nor *dialogAcknowledged*(v) and rules 2 and 3 of (v, Q) are disabled. □

Proof (Lemma 16). Let c' denote $(c : (u, R_w))$. The changeset of c' is a subset of $\{u.r_w, u.d_w, u.c_w\}$. Rank, guards and statements of (v, Q) only reference variables from cell v and primary states of neighboring cells. These are not in the changeset of c'. □

Proof (Lemma 17). In the following, m_2 and m_1 denote two distinct instances of nodes v_2 and v_1 and cells u_2 and u_1 respectively. For the given ranks r_2, r_1, the variable c denotes a configuration which satisfies $r_2 = c \vdash r(m_2) = r_2$ and $r_1 = c \vdash r(m_1)$ and c' denotes the configuration $(c : m_1)$. By Observation 10, only the case that m_2 and m_1 are neighboring instances is considered.

Case a) $r_2 = 1 \wedge r_1 = 1$: The assumption yields rule 2 of both m_2 and m_1 is enabled in c and that $c \vdash u_1.q = u_2.q = \textsc{Repaired}$. Because of $c \vdash u_1.q$, the changeset of c' is $\{v_1.r_{u_1}, v_1.d_{u_1}\}$. Hence, $c' \vdash u_2.q = \textsc{Repaired}$ holds and thus $c' \vdash \neg \mathit{waitCond}_{R_{u_2}}(v_2)$. Besides $\mathit{waitCond}_{R_{u_2}}(v_2)$, the guard of rule 2 of m_2 only references the variables $v_2.r_{u_2}$, $u_2.s$, and $u_2.q$ which are not in the changeset of c'. So rule 2 of m_2 remains enabled in c' and thus $c' \vdash r(m_2) = c \vdash r(m_2)$. Since $c' \vdash u_2.q = \textsc{Repaired}$, the output of $(c' : m_2)|_{m_2}$ only depends on the variable $u_2.q$, primary states, and copy variables which are not part of the changeset of c'.

Case b) $r_2 = 2 \wedge r_1 \in \{1, 2\}$: At worst, the changeset of c' is $\{v_1.r_{u_1}, v_1.c_{u_1}, v_1.d_{u_1}\}$. The predicate $\mathit{waitCond}_{R_{u_2}}(v_2)$ is the only part of the guards of m_2 which might reference these variables, namely in the case $v_2 = u_1$. By [8, Lemma 4] $c \vdash \mathit{repaired}(v) \Rightarrow c' \vdash \mathit{repaired}(v)$. Hence $c \vdash \mathit{waitCond}_{R_{u_2}}(v_2) \Rightarrow c' \vdash \neg \mathit{waitCond}_{R_{u_2}}(v_2)$. and the rule of m_2 that is enabled in c is still enabled in c'. It follows that $c' \vdash r(m_2) = c \vdash r(m_2)$. Since m_2 is of rank 2, it sets $v_2.r_{u_2}$ to $u_2.q$ and possibly updates $v_2.c_{u_2}$ with $u_2.p$. Both $u_2.q$ and $u_2.p$ are not among the changeset of c' and thus $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

Case c) $r_2 \in \{3, 4, 5\} \wedge r_1 \in \{1, 2\}$: If $u_1 \neq u_2$, then Lemma 16 applies. If $c \vdash \mathit{dialogConsistent}(u_2)$, then Lemma 15 applies. If $\neg(c \vdash \mathit{dialogConsistent}(u_2))$, then Lemma 14 applies.

Case d) $r_2 \in \{3, 4, 5\} \wedge r_1 = 3$: Then $v_1 \neq v_2$, since there is only one instance of protocol Q per node. The changeset of c' is $\{v_1.q\}$. These variables are not

referenced by any guard or statement of m_2. Hence, $c' \vdash r(m_2) = c \vdash r(m_2)$ and $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

Case e) $r_2 = 4 \wedge r_1 = 4$: Again $v_1 \neq v_2$. The changeset of c' is a subset of $\{v_1.s, v_1.q, v_1.p\}$. The guard of rule 3 of m_2 does not reference any of these variables. Hence $c' \vdash r(m_2) = c \vdash r(m_2)$. The statement of rule 3 doesn't reference any variables in the changeset of c', except for $v_1.p$ which is referenced only during the execution of protocol P. Hence, $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$, if m_2 and m_1 are sorted according to an invariancy ranking for protocol P.

Case f) $r_2 = 5 \wedge r_1 \in \{4, 5\}$: Again $v_1 \neq v_2$. The changeset of c' is a subset of $\{v_1.s, v_1.q, v_1.p\}$. Because of r_2, $c \vdash v_2.q$ and $c' \vdash v_2.q$ differ from EXECUTED. In this case, these variables in the changeset of c' are not referenced by m_2. Hence $c' \vdash r(m_2) = c \vdash r(m_2)$ and $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$. □

In the following, the operator $\circ$ is used to denote the concatenation of sequences.

Lemma 19. *Let e_1 be an execution and r_1 the first round of e_1 such that $e_1 = r_1 \circ e_1'$. If e_2 is a suffix of e_1 and r_2 is the first round of e_2 such that $e_2 = r_2 \circ e_2'$, then e_2' is a suffix of e_1'.*

Proof. Only the case that e_1' is a proper suffix of e_2 is considered. Otherwise, the claim is obviously true. Let r_0' be a prefix of r_1 and r_1' a suffix of r_1 such that $e_1 = r_0' \circ e_2$ and $r_1 = r_0' \circ r_1'$. The claim follows if r_1' is a prefix of r_2, which is shown in the following.

Let $r_1 = \langle c_0, c_1, \ldots, c_k \rangle$. There exists an instance $m \in \mathcal{M}$ that is enabled in all $c_0, c_1, \ldots, c_{k-1}$ and that is executed or becomes disabled during the transition $c_{k-1} \to c_k$, but not earlier. If c_k is the first configuration of r_2, then r_1' is clearly a prefix of r_2. Otherwise, some c_i, $i < k$ is the first configuration of r_2 and m is enabled in c_i. Hence, r_2 must include c_k and thus r_1' is a prefix of r_2. □

Lemma 20. *Let e be an execution of x rounds. Any suffix of e has at most x rounds.*

Proof. Let e be an execution and e' a suffix of e. Assume that e is partitioned into rounds $r_1, r_2, \ldots, r_k$ and e' into rounds $r_1', r_2', \ldots, r_l'$. Let $e_i = r_i \circ r_{i+1} \circ \ldots \circ r_k$ and $e_i' = r_i' \circ r_{i+1}' \circ \ldots \circ r_l'$. The term e_{k+1} is defined to be the empty sequence. By induction on j it is shown that any e_j' is suffix of e_j. In particular, e_{k+1}' is a suffix of e_{k+1} which is the empty sequence. Hence, e_{k+1}' is the empty sequence and thus $l \leq k$. By assumption e_1' is suffix of e_1. Assume that e_j' is suffix of e_j for $j \leq k$. By Lemma 19 it follows that e_{j+1}' is a suffix of e_{j+1}. □

Proof (Theorem 9). Let e be an execution under the distributed scheduler that is partitioned into rounds $r_1, r_2, \ldots, r_k$. By Corollary 8, an execution e' exists such that e is subsequence of e'. Let x be the number of rounds within e' and let e' be partitioned into $r_1', r_2', \ldots, r_k'$ such that r_i' starts with the configuration that coincides with the first configuration of r_i. Let e_i' denote $r_i' \circ r_{i+1}' \circ \ldots \circ r_k'$ and e_{k+1}' the empty sequence. By induction over j it is shown, that every e_j' is an execution of at most $x - j + 1$ rounds. It follows, that e_{k+1}' is a sequence of at most $x - k$ rounds and thus $k \leq x$.

e'_1 is obviously an execution of at most x rounds. For $j \leq k$, assume that e'_j is an execution of at most $x - j + 1$ rounds. For any instance $m \in \mathcal{M}$, r_j either contains an execution of m or a configuration in which m is disabled. Since r_j is a subsequence of r'_j, the same holds for r'_j and the first round of e'_j is a prefix of r'_j. By Lemma 20, e'_{j+1} consists of at most $x - j$ rounds. □