A Formal Analysis of a Dynamic Distributed Spanning Tree Algorithm

Arjan J. Mooij and Wieger Wesselink

Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. We analyze the spanning tree algorithm in the IEEE 1394.1 draft standard, which correctness has not previously been proved. This algorithm is a fully-dynamic distributed graph algorithm, which, in general, is hard to develop. The approach we use is to formally develop an algorithm that is almost equivalent to it: First, based on a formal specification and an abstraction of the network, we systematically construct an algorithm including its correctness proof. Afterwards we implement this algorithm in terms of IEEE 1394 devices under maintenance of its correctness.

1 Introduction

The IEEE 1394.1 standard is developed on top of the IEEE 1394 Standard for a High Performance Serial Bus [1], sometimes referred to as FireWire. This IEEE 1394 standard defines a bus as a (non-empty) limited collection of devices that can communicate with each other by means of messages. Each IEEE 1394 device is a computational unit that has a unique identity and that belongs to exactly one bus. The buses are dynamic in the sense that the following topology changes can occur: creation and removal of a bus with one device, merging two buses into one bus, and splitting a bus into two (non-empty) buses. Upon a topology change, all devices on the related buses are notified that changes have occurred.

To lift the limitation on the number of devices that can communicate with each other (on a single bus), in 1996 the development of the IEEE 1394.1 Standard for High Performance Serial Bus Bridges was initiated. It introduces bi-directional bridges to dynamically interconnect pairs of buses. Such a bridge consists of two portals, which are special devices on the buses that are connected by the bridge. Apart from the usual communication capabilities on the buses, the two portals of a bridge can also directly communicate with each other.

The IEEE 1394.1 standard, which is still a draft [2] at the moment of writing, contains a distributed algorithm called net-update[1], which maintains a spanning tree on the network of buses and bridges. Although this algorithm is also involved in some message routing and bus identification issues, we will not consider these. Until now the correctness of this algorithm has not been proven.

[1] We studied this algorithm as a case study of the project "Improving the Quality of Protocol Standards", funded by the NWO under project number 016.023.015.

There exist a lot of distributed algorithms for computing spanning trees, but distributed maintenance of spanning trees under dynamic topology changes turns out to be far more complicated (see e.g. [3]). Because we consider maintenance under both additions and removals of edges, [4] classifies this problem as a fully-dynamic graph problem. And since all devices of a bus are notified about topology changes in which the bus is involved, [5] classifies an IEEE 1394.1 network as a topology-aware network. Because of this topology-awareness, the algorithm is not required to be self-stabilizing (see e.g. [6,7]) with its inherent performance drawbacks. Performance is also the reason that we do not simply apply the transformation of static network protocols as proposed in [8].

Nowadays attempts to analyze and to prove the correctness of such algorithms are frequently based on automated formal verification techniques like model-checking. A well-known problem in model-checking is the so-called explosion of the state space of the system, which mainly occurs in systems with many interacting processes, and systems containing data, see e.g. [9]. Since the algorithm we want to analyze fits both profiles, model-checking is quite likely to give rise to problems. Although [9] reports considerable progress in handling large state spaces, the application of model-checking techniques by [10] to IEEE 1394.1's net-update algorithm did not result in a complete verification. Nevertheless, some error traces were found for which fixes were proposed.

A typical problem of a-posteriori verifying an algorithm against a specification, is that the information about how the algorithm was intended to fulfill the specification is usually not available. Instead of traditional a-posteriori verification, in this paper we will formally reconstruct the algorithm from its specification. More specifically, we will manually, but systematically, develop an algorithm that is almost equivalent to net-update, starting from a set of formal requirements. Our algorithm differs from net-update in some of the details that are still under discussion. Such a formal development shows how the requirements influence the algorithm under development; and as an important side-effect it provides a correctness proof. In order to focus on the essence of the algorithm, we will not provide a fully-detailed implementation in this paper.

In the current IEEE 1394.1 draft standard, algorithms are described as implementations for the portals. Attempts to formalize and to analyze such protocols are often based on such an implementation-level description, see e.g. [10]. However, we will use another approach since we think that in order to understand the essence of the algorithm, the portals are not the right entities to start reasoning about. Since this algorithm is related to spanning trees, we prefer to reason about a graph of buses and bridges. We will first abstract from the portals such that we obtain a proper graph formalization. In this graph context we then develop, and prove the correctness of, a distributed spanning tree algorithm with a net-update alike behavior. Finally we implement this algorithm on the portals.

This paper is organized as follows: Section 2 presents our graph specification. Then Section 3 introduces some notations, abbreviations and methodology that will be used in later sections. In Section 4 we develop an algorithm, which, in Section 5, is implemented on the portals. Finally, Section 6 gives the conclusions.

2 Graph specification

In this section we present a specification in terms of graphs by abstracting from the portals. An IEEE 1394.1 network can be straightforwardly modelled as a graph if nodes represent buses and edges represent bridges. Consequently, the portals are represented by the connections between nodes and edges. However, in graph algorithms these connections are usually not the computational units.

Therefore we abstract from the portals and assume that the nodes and the edges are the (potentially parallel) computational units. Furthermore, we assume that each edge has a unique identity, it can maintain persistent data, and it can communicate with the nodes it connects; and we assume that each node can communicate with its incident edges. We return to this abstraction in Section 5, where we show how to implement an algorithm that is based on this abstraction using the original portals.

Since the dynamic topology changes can partition the network, we extend the notion of a spanning tree to a spanning forest. So we have to develop a distributed algorithm that maintains a spanning forest of the graph, under the following dynamic topology changes: creating and removing nodes and edges, and merging and splitting nodes. The spanning forest must be "maintained under dynamic topology changes" in the sense that it will eventually be computed if (during a sufficiently large period of time) no more topology changes occur.

We define a forest as a directed graph which

A contains for each node at most one outgoing edge; and which
B contains no cycle.

The edges in such a forest are directed towards the roots of the forest, which are the nodes without outgoing edges. We define a spanning forest of a graph as a forest that is a subgraph of the graph (with the same set of nodes), where

C each two neighbor nodes in the graph are connected (in the forest).

3 Preliminaries

3.1 Graphs

To avoid confusion between the two graphs we consider, namely an undirected graph (for the network graph) and a directed subgraph (for a spanning forest of the graph), we introduce some nomenclature, notations and abbreviations.

An edge (with identity) e between nodes u and v in the undirected graph is denoted by $e : u \sim_e v$, and an edge e from node u to node v in the directed subgraph is denoted by $e : u \rightarrow_e v$. Each edge from the undirected graph occurs in the directed subgraph in at most one direction. An edge of the undirected graph that is not contained in the directed subgraph is called a muted edge; for an edge $e : u \sim_e v$ we correspondingly have $e.muted \equiv \neg(u \rightarrow_e v \ \vee \ v \rightarrow_e u)$.

Frequently we want to indicate all edges or all self-loops of a node in the graph, or all outgoing edges, all incoming edges or all muted edges of a node in the subgraph. To that end, we introduce for each node v the sets[2] $v.edges$ and $v.loops$, and $v.out$, $v.in$ and $v.muted$ respectively with e.g. invariants $v.in \cup v.out \cup v.muted = v.edges$ and $v.in \cap v.out \subseteq v.loops \subseteq (v.in \cap v.out) \cup v.muted$.

To avoid always explicitly referring to these sets, we introduce the following (short-hand) operations for an edge $e : v \sim_e w$:

mute $e : v \rightarrow_e w$	$\equiv v.out, v.muted := v.out \backslash \{e\}, v.muted \cup \{e\} \parallel$ $w.in, w.muted := w.in \backslash \{e\}, w.muted \cup \{e\}$
unmute $e : e.muted$ as $v \rightarrow_e w$	$\equiv v.muted, v.out := v.muted \backslash \{e\}, v.out \cup \{e\} \parallel$ $w.muted, w.in := w.muted \backslash \{e\}, w.in \cup \{e\}$
turn $e : v \rightarrow_e w$	$\equiv$ **mute** e ; **unmute** e as $w \rightarrow_e v$

3.2 Programs

In this section we briefly summarize some aspects of parallel programs and their development. We describe programs using the following constructions:

- **skip** : the empty statement;
- $x, y := E, F$: first evaluate expressions E and F, and afterwards assign their values to variables x and y respectively;
- x: $P.x$: non-deterministically assign variable x a value X satisfying $P.X$;
- $S; T$: first execute statement S and then execute statement T;
- **if** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **fi** : wait until one of the guards B_0 or B_1 holds, and then execute one statement $S_{\{0,1\}}$ for which the corresponding guard holds;
- **await** (B) : shorthand for **if** $B \rightarrow$ **skip fi**;
- **do** $true \rightarrow S$ **od** : repeat its body, i.e. statement S, infinitely often;
- **parallel for** x: $P.x$ **do** $S.x$: execute statement $S.x$ for all x: $P.x$ in parallel.

We refer to [11] for formal definitions of most of these constructions. For the sequential composition we sometimes use a new line instead of a semicolon. Apart from the usual atomic actions like the **skip** statement, the assignments and the evaluations of guards, we can create larger atomic actions by placing a series of statements within atomicity brackets $\langle ... \rangle$.

We use the programming methodology of [11] to systematically develop algorithms. To that end we annotate our programs with assertions. An assertion is a predicate on the state space of the system and it is placed at a control point, i.e. in between two subsequent atomic statements. An assertion at a control point is correct if the state of the system satisfies the assertion whenever a process is at the control-point. A *queried* assertion is an assertion that has not yet been proven to be correct. A pre-assertion of a statement is an assertion at the control point preceding the statement. Usually an assertion Q is denoted by $\{Q\}$, a queried assertion Q by $\{?\ Q\}$, and a statement S with pre-assertion Q by $\{Q\}S$.

[2] Using sets instead of bags is a major modelling decision with respect to self-loops.

In practice we use the Owicki-Gries theory [12] (see also [11]) for determining the correctness of an annotation. It states that an assertion in a process is correct if

- local correctness is guaranteed, i.e. if it is an initial assertion it is implied by the precondition of the algorithm, and if it is preceded in the process by atomic statement $\{Q\}S$ it is established by this statement; and
- global correctness (sometimes called maintenance or interference freedom) under each atomic statement $\{Q\}S$ in the other processes is guaranteed, i.e. it is maintained by these statements.

A system invariant of a system is an assertion that is placed at each control point of the system. So a system invariant is correct whenever it is implied by the precondition of the algorithm, and it is maintained by each atomic statement in the algorithm. Instead of explicitly mentioning all system invariants at all control points, we usually record them separately.

A repetition invariant of a repetition is an assertion that is placed at the first and the last control point of the body of the repetition. For local correctness it must be established by the statement preceding the repetition, and it must be (re-)established by the body of the repetition. Usually, we also record repetition invariants separately.

Programs can be developed by expressing requirements in terms of queried assertions, and then ensuring that, one-by-one, all queried assertions become correct assertions. A way to ensure that a queried assertion becomes correct, is to strengthen the annotation with fresh queried assertions. This is a valid approach, since such a strengthening cannot endanger the correctness of the current annotation. Other ways are to strengthen a guard, or to insert a statement preceding that queried assertion. Strengthening guards cannot endanger the annotation, but inserting a statement can potentially endanger all assertions.

4 Abstract algorithm

In this section we develop an algorithm for the graph abstraction of Section 2. We start by massaging the specification into a more appropriate shape. Then we will develop an initial version of our algorithm that is *partially correct*, i.e. whenever the algorithm stabilizes all requirements are fulfilled. To ensure that the algorithm *stabilizes* if (during a sufficiently large period of time) no more topology changes occur, we will afterwards reduce the possible behavior of the algorithm. Finally we will prevent that (unwanted) *deadlocks* occur, again by reducing the possible behavior. After completing the development of the algorithm, we will consider its initialization and the dynamic topology changes.

While developing the algorithm, we regularly focus on fragments of the algorithm and its annotation, and temporarily omit the rest. For completeness reasons, Appendix A contains a fully-annotated version of the whole algorithm. Since we treat the algorithm in a non-operational way, this section contains no examples, but one execution scenario has been included in Appendix B.

4.1 Specification

Requirements B and C in the specification from Section 2 have a rather global nature in the sense that they consider a large part of the network. For developing a distributed algorithm, it is more convenient to have a specification with a more local nature, like requirements about nodes (or edges) and their direct neighbors.

Therefore we will transform our specification into a more local specification. Such transformation usually involves the introduction of some variables in the nodes or edges. Since only the edges can maintain persistent data, we will only introduce variables in the edges.

Requirement A itself is sufficiently local, so we maintain it as:

A_1 for each node v: $|v.out| \leq 1$

To make requirement B more local, we associate with each edge e an integer variable *dist.e*. If we require that for each outgoing edge f and incoming edge g of a node, *dist.f* must be strictly smaller than *dist.g* (or vice versa), then we can prove that there is no cycle as follows: assume that there is a cycle containing edge e, then by transitivity of $<$ we could derive the contradiction $dist.e < dist.e$; so there is no cycle. So we transform requirement B into:

B_1 for each node v, and edges $f, g : f \in v.out \wedge g \in v.in$: $dist.f < dist.g$

To make requirement C more local, note that two nodes are connected in a forest if they belong to the same tree. So if each node stores the unique identity of the tree it belongs to, a local version of requirement C would be that each two neighbor nodes in the graph store the same tree identity. For a unique identity of a tree, we could exploit that the tree is rooted by using the unique identity of its root node.

Since nodes cannot store data, we distribute these stored tree identities to the edges and require that all incident edges of a node store the same identity (see C_1 below). And since a root node itself has no unique identity and no outgoing edges, we use the identity of one[3] of its incoming edges or self-loops unless it has no edges (see C_2 below). Note that since a muted edge is symmetric with respect to the nodes it connects, we cannot use the identity of a muted non-self-loop as a unique identity of a root node. So we associate with each edge e a variable *root.e* of type edge identity, and transform requirement C into:

C_1 for each node v, and edges $f, g : f \in v.edges \wedge g \in v.edges$: $root.f = root.g$
C_2 for each node v: $v.edges = \emptyset \vee v.out \neq \emptyset \vee (\exists f : f \in v.in \cup v.loops : root.f = f)$

So requirements A, B and C are fulfilled if the more local requirements A_1, B_1, C_1 and C_2 are fulfilled; and that is what we exploit in order to develop a distributed algorithm.

[3] Net-update jargon: "The prime portal of the tree"

4.2 Partial correctness

In this section we define the overall shape of the algorithm by developing an initial, but partially-correct, version of the algorithm. In general, numerous algorithms can be developed for a given specification, and thus a lot of design decisions will be made in this section. To stay close to net-update, we include some short high-level descriptions of the way net-update is supposed to behave before we actually formally develop the corresponding part of the algorithm.

In net-update each node locally tries to establish the requirements "related" to the node. However, it turns out that when a node tries is establishing its own related requirements, some requirements related to its neighbor nodes may be endangered. Therefore nodes signal[4] their neighbor nodes when some of their related requirements may be violated.

Because at any time a node can be signalled, or a dynamic topology change can occur in the network, the algorithm may never terminate. Therefore net-update *stabilizes* when no node has been signalled anymore and all nodes have established their related requirements.

To formalize the ideas in this description, we first explicitly associate with each node v its "related" requirements $Q.v$. We define $Q.v$ as the conjunction of

$$
\begin{array}{lll}
Q.v.0 & \equiv & (\forall f, g : f \in v.out \ \wedge \ g \in v.out : f = g) \\
Q.v.1 & \equiv & (\forall f, g : f \in v.out \ \wedge \ g \in v.in : dist.f < dist.g) \\
Q.v.2 & \equiv & (\forall f, g : f \in v.edges \ \wedge \ g \in v.edges : root.f = root.g) \\
Q.v.3 & \equiv & v.edges = \emptyset \vee (\exists f :: f \in v.out) \vee (\exists f : f \in v.in \cup v.loops : root.f = f)
\end{array}
$$

Note that we slightly rephrased requirements A_1 and C_2 (i.e. $Q.v.0$ and $Q.v.3$) to make them more similar to the other requirements (for later use). Note that if $Q.v$ holds for all nodes v, the original requirements are fulfilled.

For nodes $v : v.edges = \emptyset$, related requirements $Q.v$ reduce to *true*. Moreover, such nodes cannot communicate with other nodes or edges. For simplicity reasons, we will not further consider these isolated nodes.

To formalize the signals, we could associate with each node a single boolean variable to indicate whether the node has been signalled. But in order to have more manipulative freedom, we associate with each combination of a node v and an edge $f : f \in v.edges$ a boolean variable $sig.v_f$ to indicate whether node v has been signalled via edge f. Then we are heading for an algorithm, for each node v, of the following shape:

```
  {v.edges ≠ ∅}
do true →
     parallel for f : f ∈ v.edges do sig.v_f := false
     ...
       {? Q.v ∨ (∃f : f ∈ v.edges : sig.v_f)}
     await( (∃f : f ∈ v.edges : sig.v_f) )
od
```

[4] Net-update jargon: "A bus reset is initiated on the neighbor bus."

Observe that this is just a partial algorithm in the sense that we still have to fill in the gap "..." in such a way that the queried assertion becomes a correct assertion. So each node v starts to reset variable $sig.v_f$ for all edges $f : f \in v.edges$, and then it executes the gap to establish $Q.v$ unless it gets signalled. Afterwards, when it has been signalled, it starts over again. Upon stabilization of the network, in each node v this (currently queried) assertion holds and the negation of the **await**-guard holds. Hence $(\forall v :: Q.v)$ holds, which fulfills the specification.

In net-update, the (currently queried) assertion is established in two phases, which turns out to be related to the internal structure of condition $Q.v$. Observe that all terms in $Q.v.0$, $Q.v.1$ and $Q.v.2$ are about *pairs* of edges, while $Q.v.3$ can be witnessed by *one* edge. In net-update, first a witness edge[5] is elected for $Q.v.3$, and then based on this edge $Q.v.0$, $Q.v.1$ and $Q.v.2$ are established.

The terms in queried assertion $\{?\ Q.v\ \vee\ (\exists f : f \in v.edges : sig.v_f)\}$ (as well as in $Q.v$) contain variables about multiple edges, which is not a convenient basis for developing an algorithm in which the data is distributed. Therefore we strengthen this assertion into a more convenient assertion $\{R.v\}$ in which each term refers to the variables about at most one edge. Such a transformation usually requires the introduction of some variables (like in Section 4.1).

For the parts of the queried assertion that are related to conditions $Q.v.2$ and $Q.v.0$, we will exploit the transitivity and the symmetry of $=$. We strengthen these parts into conjuncts $R.v.0$ and $R.v.1$ of $R.v$ by introducing in each node v fresh local variables r and e of type edge identity as follows

$$R.v.0 \ \equiv\ (\forall f : f \in v.edges : sig.v_f\ \vee\ root.f = r)$$
$$R.v.1 \ \equiv\ (\forall f : f \in v.out : sig.v_f\ \vee\ f = e)$$

For the parts related to condition $Q.v.1$, we exploit its asymmetry by distinguishing between outgoing and incoming edges. We strengthen these parts into conjuncts $R.v.2$ and $R.v.3$ by introducing in each node v a fresh local integer variable d as follows

$$R.v.2 \ \equiv\ (\forall f : f \in v.out : sig.v_f\ \vee\ dist.f \leq d)$$
$$R.v.3 \ \equiv\ (\forall f : f \in v.in : sig.v_f\ \vee\ d < dist.f)\ \vee\ (\forall f : f \in v.out : sig.v_f)$$

Analogous to $R.v.3$, we could also introduce a disjunct $(\forall f : f \in v.in : sig.v_f)$ in $R.v.2$. However, this stronger asymmetric combination turns out to simplify the rest of the development.

What remains are the parts related to condition $Q.v.3$. For simplicity reasons, we strengthen these parts into conjunct $R.v.4$ by using variable e as a witness of $Q.v.3$ as follows

$$R.v.4 \ \equiv\ sig.v_e\ \vee\ v.edges = \emptyset\ \vee\ e \in v.out\ \vee\ (e \in v.in \cup v.loops\ \wedge\ root.e = e)$$

Because $R.v$ implies the original queried assertion in node v, it is sufficient to turn queried assertion $R.v$ into a correct assertion.

[5] Net-update jargon: "An alpha portal, which leads towards the prime portal"

Note that assertion $R.v$ (as well as its predecessor) can be made correct by inserting an assignment $sig.v_f := true$ in a node v for an edge $f : f \in v.edges$. However, such an assignment can easily endanger stabilization. Later on we will deal with stabilization in more detail, but for the moment we at least avoid to introduce assignments in a node v that can reduce to $sig.v_f := true$.

We first consider local correctness of assertion $R.v$ (i.e. that the assertion is established by its preceding statement in this node). Using that in $R.v.0$, $R.v.1$, $R.v.2$ and $R.v.3$ the variables of different edges are fully decoupled by local variables of node v, these conditions can be established for all edges independently, e.g. in parallel. Since $R.v.4$ imposes extra requirements on edge e, we will consider the conditions on edge e separately. More specifically, for local correctness of assertion $R.v$ we insert a parallel construct that establishes for each edge $f : f \neq e$ the conjuncts in $R.v$ about edge f, and we require the conjuncts in $R.v$ about edge e as invariants of the parallel construct. Note that these invariants must be established as pre-assertion of the parallel construct.

Global correctness of assertion $R.v$ (i.e. that the assertion is maintained by the statements in the other nodes) will follow (see Appendix E) from the global correctness of the assertions and invariants that we introduced for local correctness. So we are heading for a program fragment of the following shape:

```
...
 {inv ? sig.v_e ∨ root.e = r} {inv ? sig.v_e ∨ e ∉ v.out ∨ dist.e ≤ d}
 {inv ? sig.v_e ∨ e ∉ v.out ∨ e ∉ v.in ∨ d < dist.e}
 {inv ? sig.v_e ∨ e ∈ v.out ∨ (e ∈ v.in ∪ v.loops ∧ root.e = e)}
parallel for f, u : u ∼_f v ∧ f ≠ e do
    ...
     {? sig.v_f ∨ root.f = r} {? sig.v_f ∨ f ∉ v.out} {? sig.v_f ∨ f ∉ v.in ∨ d < dist.f}
 {R.v}
```

We first deal with the last series of queried assertions. For simplicity reasons, we will guarantee their local correctness by inserting one large atomic statement in the parallel construct. We first discuss some straightforward ways to establish the individual queried assertions, and then we combine them into alternatives of an alternative construct. We can introduce as many alternatives as we like, since alternatives can safely (with respect to partial correctness) be eliminated.

Recall that we do not want to establish these assertions using an assignment to $sig.v_f$; but if $sig.v_f$ already holds, a **skip** is sufficient to fulfill all of them. Alternatively, we can fulfill the first assertion by inserting an assignment $root.f := r$. For the second assertion we can insert a statement $\{f \in v.out\}$ **mute** f or $\{f \in v.out \wedge u \neq v\}$ **turn** f. Note that this second assertion forces that self-loops must (eventually) become muted edges. To establish the last assertion we can insert an assignment $\{f \in v.in\}$ $dist.f := d + 1$. Although for this last assertion we could also mute incoming edges, this turns out to only complicate the rest of the development. For reasons that will become clear later, we do consider the statement $\{f \in v.muted \wedge u \neq v\}$ **unmute** f as $u \rightarrow_f v$.

For global correctness of these assertions, we consider the statements in node v and this series of assertions in a node $u : u \neq v$. We accompany each statement that affects an edge $f : u \sim_f v$ and that can possibly endanger these assertions with an assignment that evaluates to $sig.u_f := true$. Thus we obtain:

```
...
 {inv ? sig.v_e ∨ root.e = r} {inv ? sig.v_e ∨ e ∉ v.out ∨ dist.e ≤ d}
 {inv ? sig.v_e ∨ e ∉ v.out ∨ e ∉ v.in ∨ d < dist.e}
 {inv ? sig.v_e ∨ e ∈ v.out ∨ (e ∈ v.in ∪ v.loops ∧ root.e = e)}
parallel for f, u : u ∼_f v ∧ f ≠ e do ⟨
    if f ∈ v.out →
        root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
        mute f
    [] f ∈ v.out ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, true
        turn f
    [] f ∈ v.in ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
    [] f ∈ v.muted ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, true
        unmute f as u →_f v
    [] f ∈ v.muted →
        root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
    [] sig.v_f →
        skip
    fi ⟩
     {sig.v_f ∨ root.f = r} {sig.v_f ∨ f ∉ v.out} {sig.v_f ∨ f ∉ v.in ∨ d < dist.f}
```

We continue with the remaining queried invariants. Maintenance of these invariants under the statement in the parallel construct in node v is guaranteed, thanks to condition $f \neq e$. Global correctness of these invariants (in a node $u : u \neq v$ under the statements in node v) is guaranteed for the first and the third invariant. For the second invariant, we extend the assignment to $sig.u_f$ for an edge $f : f \in v.in \wedge u \neq v$ with a disjunct $dist.f \leq d$. For the last invariant, we extend the assignment to $sig.u_f$ for muting an edge $f : f \in v.out$ with a disjunct $(root.f = f \ \wedge \ u \neq v)$.

What remains is local correctness. Note that using the second invariant the last disjunct of the third invariant is redundant. For local correctness of the first two invariants we insert an assignment to r and d that establishes the first two invariants. Note that such an assignment cannot endanger global correctness of the current annotation. For local correctness of the other two invariants we require them as pre-assertions of this assignment:

```
...
 {? sig.v_e ∨ e ∉ v.out ∨ e ∉ v.in}
 {? sig.v_e ∨ e ∈ v.out ∨ (e ∈ v.in ∪ v.loops ∧ root.e = e)}
r, d := root.e, dist.e

 {inv sig.v_e ∨ root.e = r} {inv sig.v_e ∨ e ∉ v.out ∨ dist.e ≤ d}
 {inv sig.v_e ∨ e ∉ v.out ∨ e ∉ v.in}
 {inv sig.v_e ∨ e ∈ v.out ∨ (e ∈ v.in ∪ v.loops ∧ root.e = e)}
parallel for f, u : u ∼_f v ∧ f ≠ e do ⟨
    if f ∈ v.out →
        root.f, sig.u_f := r, sig.u_f ∨ ((root.f ≠ r ∨ root.f = f) ∧ u ≠ v)
        mute f
    [] f ∈ v.in ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r ∨ dist.f ≤ d
    ...
    fi ⟩
```

Global correctness of these queried assertions follows from the same arguments as above. For local correctness of the second assertion, we insert a statement that selects an edge e that fulfills it. Such a local selection cannot endanger global correctness of an annotation. For local correctness of the other assertion, we require its generalization to all edges of the node as a pre-assertion:

$$\begin{array}{l}\ldots \\ \quad \{?\ (\forall f : f \in v.edges : sig.v_f \ \vee\ f \notin v.out \ \vee\ f \notin v.in)\} \\ e{:}\ e \in v.out \ \vee\ (e \in v.in \cup v.loops \ \wedge\ root.e = e)\end{array}$$

Global correctness of the queried assertion is guaranteed by the same argument as above. Since this assertion is a post-assertion of the assignments $sig.v_f := false$, we will establish its local correctness by requiring the equivalent condition $v.loops \subseteq v.muted$ as loop invariant.

This loop invariant is a local condition in the sense that it cannot be endangered by other nodes, so its global correctness is always guaranteed. Its maintenance under a loop-body is also guaranteed, since each atomic statement in node v maintains it. Initialization of this invariant can be established by inserting the following straightforward program fragment before the loop:

$$\begin{array}{l}\textbf{parallel for}\ f : f \in v.loops \backslash v.muted\ \textbf{do mute}\ f \\ \\ \quad \{\textbf{inv}\ v.loops \subseteq v.muted\} \\ \textbf{do}\ true \rightarrow \ldots\ \ \textbf{od}\end{array}$$

Thus we obtained a partially-correct algorithm, for which we have not yet guaranteed that it stabilizes nor that it is deadlock-free.

4.3 Stabilization

In this section we will modify the algorithm such that it stabilizes if (during a sufficiently large period of time) no more topology changes occur. Of course these modifications must maintain partial correctness, which is guaranteed under reductions of the possible behavior of the algorithm, e.g. by strengthening the guards. To guarantee stabilization we will impose a well-founded function on the state space of the system and adapt the algorithm such that the function is a variant function for the algorithm, i.e. it (or rather its value) is descending (i.e. it never increases) and it decreases regularly.

We propose a variant function that consists of three parts. Upon stabilization we have $\neg sig.v_f$ for each edge $f : f \in v.edges$, so we head for a variant function that decreases under an assignment $\{sig.v_f\}\ sig.v_f := false$. Since in the spanning tree setting we need in fact a minimum number of (non-muted) edges, we want a variant function that decreases under **mute**-statements. Furthermore, for such an algorithm it turns out to be (at least) convenient to introduce a total order $\leq$ on the edge identities, which we use to choose a variant function that decreases if $root.f$ for an edge f decreases. As a variant function we impose the three-tuple $[(\sum f\ ::\ root.f), (\#f\ ::\ \neg f.muted), (\#v, f\ ::\ sig.v_f)]$[6] with the lexicographical order, in which # is used as "the number of"-quantor.

[6] Although we seem to assume that the addition is defined on edge identities, in fact we use addition as an abbreviation of concatenation with the lexicographical order.

We first ensure that this function is well-founded. Using that each edge has one unique identity and that there is a total order defined on the edge identities, this function is well-founded if we reasonably assume the network to be finite.

Then we ensure that this function decreases regularly. Note that upon passing the **await** statement, guard $(\exists f : f \in v.edges : sig.v_f)$ holds stably up to execution of the parallel construct with assignments $sig.v_f := false$ (see the beginning of Section 4.2), which then decrease the variant function. So after one execution of the loop-body, each further execution yields a decrease of the function.

What remains is to ensure that this function is descending under all (other) atomic statements. The only possibly-endangering statement is the statement in the large parallel construct. For each assignment $root.f := r$ we must require pre-assertion $r \leq root.f$; and for unmuting or turning an edge we must even require pre-assertion $r < root.f$. For updating an incoming edge we must require the additional pre-assertion $r < root.f \vee d < dist.f$, which can be combined into pre-assertion $(r, d) < (root.f, dist.f)$. Since this pre-assertion is within an atomic region, we directly strengthen the guards of the selection with the corresponding required pre-assertions. Furthermore, we use these guards (and invariant $v.loops \subseteq v.muted$) to simplify some statements. Thus we obtain:

```
parallel for f, u : u ∼_f v ∧ f ≠ e do ⟨
    if f ∈ v.out ∧ r ≤ root.f →
        root.f, sig.u_f := r, sig.u_f ∨ root.f ≠ r ∨ root.f = f
        mute f
    [] f ∈ v.out ∧ r < root.f →
        root.f, dist.f, sig.u_f := r, d + 1, true
        turn f
    [] f ∈ v.in ∧ (r, d) < (root.f, dist.f) →
        root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
    [] f ∈ v.muted ∧ r < root.f ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, true
        unmute f as u →_f v
    [] f ∈ v.muted ∧ r ≤ root.f →
        root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
    [] sig.v_f →
        skip
    fi ⟩
```

We obtained a partially-correct and stabilizing algorithm, which is not yet guaranteed to be deadlock-free. From this version of the algorithm we have for each $f, u, v : u \sim_f v$ the following three important properties:

Descendence : Both $(root.f, dist.f)$ and $root.f$ are descending in time.
Direction : Every statement in node v that affects f ensures $f \notin v.out$.
Signalling : A decrease of $root.f$ by node $u : u \neq v$ also establishes $sig.v_f$.

4.4 Deadlock freedom

In this section we will ensure that there are no unwanted deadlocks. The only statements that can cause a deadlock are the possibly-blocking statements, namely the statement that selects an edge e, the **if**-statement and the **await**-statement. Since stabilization is in fact a "desired deadlock" and it is achieved by the **await**-statements, we will not further consider these statements.

We will ensure that the other statements are non-blocking. An **if**-statement is a blocking statement whenever none of its guards hold. To ensure that in our algorithm these statements are non-blocking instances, we require them to have the disjunction of their guards as pre-assertion. Similarly, the selection of an edge e: $P.e$ is a blocking statement in case $\neg(\exists f :: P.f)$ holds. To ensure that in our algorithm these statements are non-blocking instances, we require them to have $(\exists f :: P.f)$ as pre-assertion.

Dealing with these required pre-assertions turns out to be rather technical: quite some assertions must be added, and some of the guards must be strengthened. In this section we only provide the most important results, but the details can be found in Appendix C.

We introduce the following system invariants:

P_0 $(\forall v : v.edges \neq \emptyset : (\exists h : h \in v.edges : S.v.h))$
P_1 $(\forall f :: root.f \leq f)$

$$\begin{aligned} \text{with } S.v.h \equiv \quad & (h \in v.out \vee (h \in v.in \cup v.loops \wedge root.h = h)) \\ & \wedge (\forall f : f \neq h \wedge f \in v.edges : root.h \leq root.f) \\ & \wedge (\forall f : f \neq h \wedge f \in v.in : (root.h, dist.h) < (root.f, dist.f)) \\ & \wedge (\forall f : f \neq h \wedge f \in v.out \wedge root.f = f : root.h < f) \end{aligned}$$

And we obtain the following algorithm (in which we left out the annotation):

```
 {v.edges ≠ ∅}
parallel for f : f ∈ v.loops\v.muted do mute f

do true →
    parallel for f : f ∈ v.edges do sig.v_f := false

    e: S.v.e
    r, d := root.e, dist.e

    parallel for f, u : u ∼_f v ∧ f ≠ e do ⟨
        if f ∈ v.out ∧ r = root.f ∧ root.f ≠ f →
            mute f
        [] f ∈ v.out ∧ r < root.f →
            root.f, dist.f, sig.u_f := r, d + 1, true
            turn f
        [] f ∈ v.in ∧ (r, d) < (root.f, dist.f) →
            root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
        [] f ∈ v.muted ∧ r < root.f ∧ u ≠ v →
            root.f, dist.f, sig.u_f := r, d + 1, true
            unmute f as u →_f v
        [] f ∈ v.muted ∧ r ≤ root.f ∧ (r = root.f ∨ u = v) →
            root.f := r
        [] sig.v_f →
            skip
        fi ⟩

    await( (∃f : f ∈ v.edges : sig.v_f) )
od
```

Thus we obtained a partially correct, stabilizing and deadlock-free algorithm. What remains is the initialization and the dynamic topology changes.

4.5 Initialization and dynamic topology changes

In this section we deal with the following two related issues: initialization of the system invariants and dealing with the dynamic topology changes. Since our algorithm has to deal with dynamically changing topologies, for initialization we can simply assume that initially there are no nodes and no edges. All other (possibly more realistic) "initial" configurations can be obtained using the dynamic topology changes, which need to be considered anyhow.

Recall that there are two basic building blocks of the network topology, viz. a node without edges (i.e. a bus without bridge portals), and an edge that connects two nodes with only one incident edge (i.e. a bridge). To be able to build all possible network topologies, we consider the following topology changes: creating and removing such basic building blocks, and merging and splitting nodes.

Also recall that upon a topology change, all nodes that are involved in the topology change are notified. Since the structure of the main computational unit of the algorithm (i.e. the node/bus) may be changed by topology changes, we adopt the net-update proposal to abort and afterwards restart the algorithm in the nodes that are notified about a topology change. In this way, we can guarantee the correctness of the algorithm and its annotation by only ensuring that the system invariants are maintained by the dynamic topology changes.

Both the creation and the removal of nodes without edges obviously maintain the invariants. The creation of an edge f that connects two nodes containing only that edge maintains the invariants if the edge is non-muted and $root.f$ has initial value f; removal of such an edge cannot endanger the invariants. Furthermore invariant P_1 is not affected by either merging or splitting nodes.

To ensure that invariant P_0 is maintained by merging nodes u and v into a node w, we study for $w.edges \neq \emptyset$ the choice of an edge $h : S.w.h$ by considering the four conjuncts of $S.w.h$. To fulfill the first conjunct we can choose an h such that $S.u.h$ or $S.v.h$; the second conjunct requires $root.h$ to be minimal. The third conjunct even requires $(root.h, dist.h)$ to be minimal, while the fourth conjunct even requires $h \in v.out \ \wedge \ root.h = h$ if possible. The last requirement follows from the one but last requirement if we adopt invariant P_2 (using invariant P_1):

$$P_2 \ \ (\forall f : f \in v.edges \ \wedge \ \neg f.muted : f \leq root.f \ \equiv \ dist.f \leq 0)$$

To maintain this invariant under creation of an edge f, we require initial value 0 for $dist.f$. This invariant is also maintained by the algorithm if we adopt the following invariant (which is also maintained):

$$P_3 \ \ (\forall f : f \in v.edges : 0 \leq dist.f)$$

In general invariant P_0 is not maintained under splitting nodes. Since the algorithm cannot avoid topology changes, it should detect violations of the invariant and restore the invariant after such a violation. Since the invariant is of

the shape "for each node some locally checkable condition holds", its violation can always be locally detected by at least one node. Since the invariant can only be violated by a topology change, this local condition only needs to be checked after a notification of a topology change.

What remains is restoring invariant P_0 whenever it has been violated. In general it cannot be locally restored by the nodes that can detect the violation. So a more global approach is needed, like a network reset[7] (see e.g. [8], but using our more-restricted start criterion) that re-initializes the reachable part (with respect to the node(s) that detected the violation) of the network under maintenance of the invariants. This is possible since the invariants hold in an empty graph, and both adding edges and nodes, and merging nodes maintain the invariant. We will not further discuss such a network reset.

5 Implementation

In Section 4, we developed an algorithm for the abstraction of IEEE 1394.1 networks as described in Section 2. Since we abstracted from the portals, which are the actual computational units, we have to implement the abstract algorithm on these portals. To that end both the data and the algorithms for the nodes and the edges must be distributed over the portals.

The algorithm can be implemented independently from its annotation. But we make an exception for statement $e : S.v.e$ in which the variables of all incident edges are involved. A standard implementation requires a snapshot of the incident edges. However, it turns out that the snapshot is not required to be consistent, which enables a better performance. This is a real modification of the algorithm; the technical details can be found in Appendix D.

To formalize this snapshot, we introduce in each node fresh local variables $root'$, $dist'$ and $state'$. And we precede the selection of an edge e with for each edge $f : f \in v.edges$ an assignment $root'.f, dist'.f, state'.f := root.f, dist.f, state.v.f$, in which $state.v.f$ denotes whether edge f is an incoming edge, an outgoing edge or a muted edge with respect to node v.

The remaining implementation issues are annotation independent. We will first transform the current algorithm into a more convenient shape. Using the snapshot, assignment $r, d := root.e, dist.e$ can be transformed into the local assignment $r, d := root'.e, dist'.e$ (see Appendix E). We also make the ranges of the parallel constructs more homogeneous by extending them to all incident edges of the node and introducing additional selection statements within the bodies of the parallel constructs. Furthermore we want to abstract from certain details of some statements by introducing functions F and G (which we will not explicitly specify). Thus in the rest of this section we consider the following algorithm:

[7] Net-update jargon: "panic", but it has not been included in draft version [2]. The proposed start criterion exploits that if a network reset is necessary, then $dist.f$, for some edge f, would eventually exceed (no proof yet!) a certain large value (like [13]).

```
 {v.edges ≠ ∅}
parallel for f : f ∈ v.edges do
    if ⟨ f ∈ v.loops\v.muted → mute f ⟩
    []   f ∉ v.loops\v.muted → skip
    fi

do true →
    parallel for f : f ∈ v.edges do
        sig.v_f := false
        root'.f, dist'.f, state'.f := root.f, dist.f, state.v.f

    e, r, d := F(root', dist', state')

    parallel for f, u : u ∼_f v do
        root.f, dist.f, state.v.f, state.u.f := G(e, r, d, f, root.f, dist.f, state.v.f)

    await( (∃f : f ∈ v.edges : sig.v_f) )
od
```

We implement this algorithm on the portals in two phases: first we make the parallelism in the nodes more explicit by introducing extra processes, and then we map these processes and the data to the portals.

For each node v we introduce one process[8] $C.v$ and a process $B.f.v$ per edge $f : f \in v.edges$. Process $C.v$ is a version of the algorithm that delegates the statements in the parallel constructs with respect to edge f to process $B.f.v$.

The communication and synchronization between these processes must be via the bus, so we will have to introduce message communication. In what follows we briefly describe the implementations of the three constructions that need to be implemented: Sequential compositions $S;T$ can be replaced by distributed sequential compositions, i.e. by sending a message upon completion of statement S, and only starting the execution of statement T after receiving the message. Assignments $x := E$ can be replaced by distributed assignments, i.e. by sending a message with value E, and after receiving this value assigning it to local variable x. And synchronization construct **await**(E) can be replaced by waiting for a message that guarantees condition E, and sufficiently often sending the message.

We introduce 6 types of messages, some of which have parameters, with names that weakly reflect their purpose: "ready", "request", "response" (r, d, s), "update" (e, r, d), "done" and "awake". Then we obtain for $C.v$:

```
 {v.edges ≠ ∅}
parallel for f : f ∈ v.edges do
    receive "ready" from B.f.v

do true →
    parallel for f : f ∈ v.edges do
        send "request" to B.f.v
        receive "response" (root'.f, dist'.f, state'.f) from B.f.v

    e, r, d := F(root', dist', state')

    parallel for f : f ∈ v.edges do
        send "update" (e, r, d) to B.f.v
        receive "done" from B.f.v

    receive at least one "awake" from B.f.v, for at least one edge f
od
```

[8] Net-update jargon: "The coordinator of the bus"

And for $B.f.v$, with $f, v, u : u \sim_f v$, we obtain:

```
if ⟨ f ∈ v.loops\v.muted → mute f ⟩
[]   f ∉ v.loops\v.muted → skip
fi
send "ready" to C.v

do true →
[] ∃ "request" from C.v →
      receive "request" from C.v
      sig.v_f := false
      send "response" (root.f, dist.f, state.v.f) to C.v

      receive "update" (e, r, d) from C.v
      root.f, dist.f, state.v.f, state.u.f := G(e, r, d, f, root.f, dist.f, state.v.f)
      send "done" to C.v

[] sig.v_f →
      send "awake" to C.v
od
```

Finally we map the processes and the data to the portals like in Figure 1. For each edge $f : u \sim_f v$, we equip the corresponding bridge with processes $B.f.u$ and $B.f.v$, and with the variables of the edge. As the identity of the edge, we use the identity of one of its portals. Less straightforward is to ensure that on each bus (i.e. a node) with at least one portal there is exactly one process C. To ensure that there is at least one process C, we equip each portal with such a process; and to ensure that there is at most one process C, a leader election protocol can be used on the bus to activate only one of them. We will not further describe such a leader election protocol, nor the details of the message communication.

6 Evaluation and conclusions

We analyzed the core of the net-update algorithm in the IEEE 1394.1 draft standard by formally reconstructing it from its specification. The algorithm we obtained is almost equivalent to the net-update algorithm, but not completely identical. Hence, this analysis is not a correctness proof of net-update draft [2], but it exposes the core ideas of this algorithm. There are mainly two differences: The first one is that our algorithm uses another condition for starting a network reset, which is caused by our system invariant P_0. The second one is that we

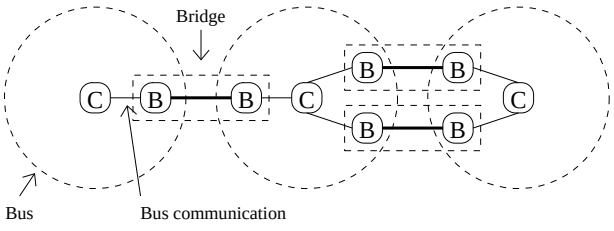

Fig. 1. Architecture

better exploited the potential parallelism on the bus, although we are not sure about whether this can be maintained for the extra functionality of net-update.

This non-trivial algorithm is in fact distributed on two levels, namely on the network level and on the portal level. By abstracting from the IEEE 1394.1 network, we were able to first focus on the behavior of the algorithm in the network, and afterwards we could focus on the local portal-implementation issues.

The way of reconstructing this algorithm was based on ideas of [11], although we applied them slightly less formal than they were intended to be applied. For further work we consider verifying the correctness proof using a theorem prover in order to gain extra confidence in the proof. The reconstruction of the algorithm was mainly guided by the proof obligations for its correctness, and hence the algorithm was developed hand-in-hand with its correctness proof. After previous experiences [14] with inherently highly-parallel non-blocking algorithms, this is again an example in which the techniques of [11] prove to be valuable.

References

1. The Institute of Electrical and Electronics Engineers, Inc.: IEEE Standard for a High Performance Serial Bus. (1996) IEEE Std 1394-1995.
2. The Institute of Electrical and Electronics Engineers, Inc.: IEEE P1394.1 Draft Standard for High Performance Serial Bus Bridges. (2002) Version 1.04.
3. Cheng, C., Cimet, I.A., Kumar, S.P.R.: A protocol to maintain a minimum spanning tree in a dynamic topology. In: Symposium proceedings on Communications architectures and protocols, ACM Press (1988) 330–337
4. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. CRC Handbook of Algorithms and Theory of Computation, Chapter 8. (1999)
5. Mooij, A.J., Goga, N., Wesselink, W.: A distributed spanning tree algorithm for topology-aware networks. Computer Science Report 03-09, Technische Universiteit Eindhoven, Eindhoven (2003)
6. Perlman, R.: An algorithm for distributed computation of a spanning tree in an extended LAN. ACM SIGCOMM **15** (1985) 44–53
7. Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its applications to self-stabilization. Theoretical Computer Science **186** (1997) 199–229
8. Afek, Y., Awerbuch, B., Gafni, E.: Applying static network protocols to dynamic networks. In: 28th Annual Symposium on Foundations of Computer Science, IEEE (1987) 358–370
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
10. Langevelde, I.v., Romijn, J., Goga, N.: Founding FireWire Bridges through Promela Prototyping. In: Proceedings Formal Methods for Parallel Programming: Theory and Applications, IEEE Computer Society Press (2003)
11. Feijen, W.H.J., van Gasteren, A.J.M.: On a method of multiprogramming. Springer-Verlag (1999)
12. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6** (1976) 319–340
13. Arora, A., Gouda, M.G.: Distributed Reset. IEEE Transactions on Computers **43** (1994) 1026–1039
14. Mooij, A.J.: Formal derivations of non-blocking multiprograms. Computer Science Report 02-13, Technische Universiteit Eindhoven, Eindhoven (2002)

A Full annotation

For compactness reasons, we frequently leave out parts of the algorithm and its annotation in Section 4 and Appendix C. In this appendix we provide the full algorithm for each node v and its annotation:

$\{v.edges \neq \emptyset\}$
parallel for $f : f \in v.loops \backslash v.muted$ **do**
  mute f

$\{$**inv** $v.loops \subseteq v.muted\}$
do $true \rightarrow$
  parallel for $f : f \in v.edges$ **do**
    $sig.v_f := false$

  $\{v.loops \subseteq v.muted\}$
  e: $S.v.e$

  $\{v.loops \subseteq v.muted\}$ $\{e \in v.out \ \vee \ (e \in v.in \cup v.loops \ \wedge \ root.e = e)\}$
  $\{(\forall f : f \neq e \ \wedge \ f \in v.edges : sig.v_f \ \vee \ root.e \leq root.f)\}$
  $\{(\forall f : f \neq e \ \wedge \ f \in v.in : (root.e, dist.e) < (root.f, dist.f))\}$
  $\{(\forall f : f \neq e \ \wedge \ f \in v.out \ \wedge \ root.f = f : root.e < f)\}$
  $r, d := root.e, dist.e$

  $\{$**inv** $sig.v_e \ \vee \ root.e = r\}$ $\{$**inv** $sig.v_e \ \vee \ e \notin v.out \ \vee \ dist.e \leq d\}$
  $\{$**inv** $v.loops \subseteq v.muted\}$ $\{$**inv** $e \in v.out \ \vee \ (e \in v.in \cup v.loops \ \wedge \ root.e = e)\}$
  parallel for $f, u : u \sim_f v \ \wedge \ f \neq e$ **do** $\langle$

    $\{sig.v_f \ \vee \ r \leq root.f\}$ $\{(\forall f : f \neq e \ \wedge \ f \in v.in : (r, d) < (root.f, dist.f))\}$
    $\{(root.e, dist.e) \leq (r, d)\}$ $\{(\forall f : f \neq e \ \wedge \ f \in v.out \ \wedge \ root.f = f : r < f)\}$
    if $f \in v.out \ \wedge \ r = root.f \ \wedge \ root.f \neq f \rightarrow$
      mute f
    [] $f \in v.out \ \wedge \ r < root.f \rightarrow$
      $root.f, dist.f, sig.u_f := r, d + 1, true$
      turn f
    [] $f \in v.in \ \wedge \ (r, d) < (root.f, dist.f) \rightarrow$
      $root.f, dist.f, sig.u_f := r, d + 1, sig.u_f \ \vee \ root.f \neq r$
    [] $f \in v.muted \ \wedge \ r < root.f \ \wedge \ u \neq v \rightarrow$
      $root.f, dist.f, sig.u_f := r, d + 1, true$
      unmute f as $u \rightarrow_f v$
    [] $f \in v.muted \ \wedge \ r \leq root.f \ \wedge \ (r = root.f \ \vee \ u = v) \rightarrow$
      $root.f := r$
    [] $sig.v_f \rightarrow$
      skip
    fi $\rangle$
    $\{sig.v_f \ \vee \ root.f = r\}$ $\{sig.v_f \ \vee \ f \notin v.out\}$ $\{sig.v_f \ \vee \ f \notin v.in \ \vee \ d < dist.f\}$

  $\{R.v\}$ $\{v.loops \subseteq v.muted\}$
  await($(\exists f : f \in v.edges : sig.v_f)$)
od

Invariants:
P_0: $(\forall v : v.edges \neq \emptyset : (\exists h : h \in v.edges : S.v.h))$
P_1: $(\forall f :: root.f \leq f)$
P_2: $(\forall f : f \in v.edges \ \wedge \ \neg f.muted : f \leq root.f \ \equiv \ dist.f \leq 0)$
P_3: $(\forall f : f \in v.edges : 0 \leq dist.f)$

Definitions:
$S.v.h \ \equiv \ \ (h \in v.out \ \vee \ (h \in v.in \cup v.loops \ \wedge \ root.h = h))$
$\wedge \ (\forall f : f \neq h \ \wedge \ f \in v.edges : root.h \leq root.f)$
$\wedge \ (\forall f : f \neq h \ \wedge \ f \in v.in : (root.h, dist.h) < (root.f, dist.f))$
$\wedge \ (\forall f : f \neq h \ \wedge \ f \in v.out \ \wedge \ root.f = f : root.h < f)$

$R.v \ \equiv \ \ (\forall f : f \in v.edges : sig.v_f \ \vee \ root.f = r)$
$\wedge \ (\forall f : f \in v.out : sig.v_f \ \vee \ f = e)$
$\wedge \ (\forall f : f \in v.out : sig.v_f \ \vee \ dist.f \leq d)$
$\wedge \ ((\forall f : f \in v.in : sig.v_f \ \vee \ d < dist.f) \ \vee \ (\forall f : f \in v.out : sig.v_f))$
$\wedge \ sig.v_e \ \vee \ v.edges = \emptyset \ \vee \ e \in v.out \ \vee \ (e \in v.in \cup v.loops \ \wedge \ root.e = e)$

B Example

To get an operational idea of how the algorithm developed in Section 4 can behave, we included in Figure 2 one possible behavior of the algorithm. It is coarse-grained in the sense that the steps are executions of a full loop-body of a node. Furthermore, it does not contain a dynamic topology change.

The figure contains a sequence of seven networks linked by horizontal and vertical arrows. The upper-left network is a just initialized network, and the lower-left network is a stabilized network. The networks contain four nodes that are interconnected by four edges with identities $a, b, c, d : a < b < c < d$. The nodes are represented by circles, the edges $\sim_f$ are represented by lines labelled with $root.f$, $dist.f$, and the edges $\rightarrow$ in the spanning forest are represented by arrows between nodes. For clarity reasons, we did not explicitly include the identities of the edges, but they can easily be derived from the initial network since initially for each edge f we have $root.f = f$. A filled dot near a node v and an edge f denotes that $sig.v_f$ holds. Diagonal arrows indicate the node that is going to perform the loop body of the algorithm according to this scenario.

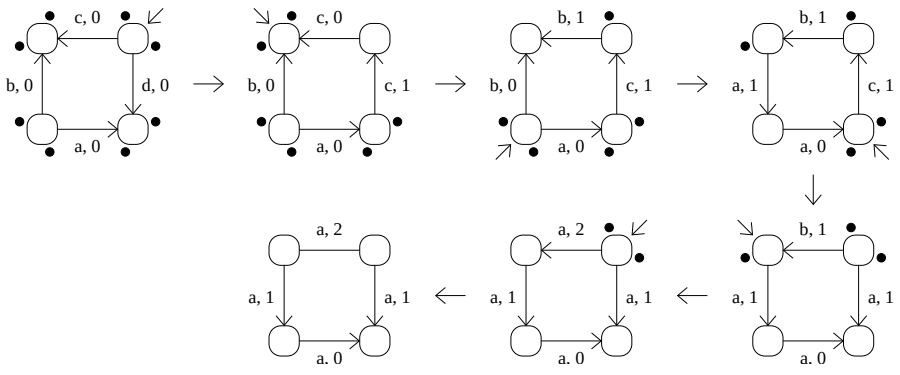

Fig. 2. Example scenario

C Deadlock freedom

This appendix provides the technical details that we omitted in Section 4.4 to improve the accessability of that section. We will ensure that, apart from the **await**-statement, all statements are non-blocking. We first consider the **if**-statement. To ensure that it is a non-blocking statement, we require it to have the disjunction of its guards as pre-assertion:

parallel for $f, u : u \sim_f v \ \wedge \ f \neq e$ **do**

$\{?\ sig.v_f \ \vee \ r \leq root.f\}$ $\{?\ sig.v_f \ \vee \ f \notin v.in \ \vee \ (r, d) < (root.f, dist.f)\}$
$\langle$ **if** ... **fi** $\rangle$

To guarantee local correctness of such assertions, they are usually explicitly required (for all $f, u : u \sim_f v \wedge f \neq e$) as pre-assertion of the parallel construct. Exploiting that this is straightforward and it introduces no new proof obligations for global correctness (see Appendix E), we will not explicitly do this. Furthermore, we will not exploit disjunct $trig.v_f$ of the last assertion for either local or global correctness, such that we can safely drop this disjunct later on.

Global correctness of these assertions is guaranteed by the signalling property and the direction property ($f \notin v.in$) respectively. Since these assertions are required post-assertions of assignment $r, d := root.e, dist.e$, their local correctness is guaranteed by requiring the following pre-assertions for that assignment:

$$\begin{array}{|l} \{?\ (\forall f : f \neq e \ \wedge\ f \in v.edges : sig.v_f \ \vee\ root.e \leq root.f)\} \\ \{?\ (\forall f : f \neq e \ \wedge\ f \in v.in : sig.v_f \ \vee\ (root.e, dist.e) < (root.f, dist.f))\} \\ r, d := root.e, dist.e \end{array}$$

Global correctness of these assertions is guaranteed by the descendence property of edge e, and the signalling and direction properties of edge f respectively. Since these assertions are required post-assertions of the selection of an edge e, their local correctness can be established by strengthening the criteria for that selection as follows:

$$\begin{array}{|l} \langle\ e{:}\ S.v.e\ \rangle \\ \{(\forall f : f \neq e \ \wedge\ f \in v.edges : sig.v_f \ \vee\ root.e \leq root.f)\} \\ \{(\forall f : f \neq e \ \wedge\ f \in v.in : sig.v_f \ \vee\ (root.e, dist.e) < (root.f, dist.f))\} \end{array}$$

$$\begin{array}{rl} \text{with } S.v.h \ \equiv & (h \in v.out \ \vee\ (h \in v.in \cup v.loops \ \wedge\ root.h = h)) \\ & \wedge\ (\forall f : f \neq h \ \wedge\ f \in v.edges : root.h \leq root.f) \\ & \wedge\ (\forall f : f \neq h \ \wedge\ f \in v.in : (root.h, dist.h) < (root.f, dist.f)) \end{array}$$

To guarantee that the selection of an edge e: $S.v.e$ causes no deadlock, we must require as pre-assertion of the selection that there exists such an edge e. We do so by requiring the following system invariant:

$P_0\ \ (\forall v : v.edges \neq \emptyset : (\exists h : h \in v.edges : S.v.h))$

Maintenance of invariant P_0 can only be endangered by the statements in the large parallel construct. In what follows, we focus on condition $(\exists h : h \in w.edges : S.w.h)$ for a node $w : w.edges \neq \emptyset$, usually in relation with an edge $g : S.w.g$ and a statement in node v. We use r and d for variables of node v.

We first consider the statements in nodes $v : v \neq w$ that affect an edge $f : v \sim_f w$ with $g \neq f$. We will ensure that in case $S.w.g$ is violated by such a statement, $S.w.f$ is established, and hence invariant P_0 is maintained. Using the direction property ($f \notin w.in$), $S.w.g$ can only be violated by an assignment $\{r < root.g\}\ root.f := r$. Using $S.w.g$ this assignment establishes $(\forall h : f \neq h \ \wedge\ h \in w.edges : root.f < root.h)$. What remains to establish $S.w.f$ is to ensure that $f \in w.out$ holds. The only possibly-violating statements that do not guarantee this are muting an outgoing edge and maintaining a muted edge. Using $S.w.g$ and condition $r < root.g$ (and hence $r < root.f$), we can exclude them by strengthening their guards with a conjunct $r = root.f$ (unless $u = v$).

Then we consider the statements in nodes $v : v \neq w$, that affect an edge $f : v \sim_f w$ with $g = f$. We will ensure that $S.w.g$ cannot be violated by such a statement. Since $S.w.g$ implies $g \notin w.muted$, we do not have to consider statements for muted edges. Fortunately the statements for $f \in v.in$ and turning $f \in v.out$ cannot violate $S.w.g$. So what remains is the statement with guard $f \in v.out \wedge r = root.f$ (see last paragraph) that mutes an outgoing edge. Since this statement can violate $S.w.g$, we want to strengthen its guard with a conjunct that implies $\neg S.w.f$. Using $f \in v.out$ (i.e. $f \in w.in$), a local way to do this is using a conjunct $root.f \neq f$.

Finally, we consider the statements in node $v : v = w$ that affect an edge $f : f \in v.edges \wedge f \neq e$. In case $g = e$, $S.w.g$ is maintained if we require pre-assertion $(root.e, dist.e) \leq (r, d)$. For the case $g \neq e$ we ensure that $S.w.g$ is maintained in case that $\neg S.w.e$ holds: for the case $f = g$ we require pre-assertion $(\forall h : \neg S.v.e \wedge S.v.h : root.h < r)$, which is also sufficient for the case $f \neq g$.

Since we strengthened some guards related to $f \in v.out$ and $f \in v.muted$, we must still ensure as a pre-assertion that at least one of the guards evaluates to *true*. As a basis we use the two corresponding pre-assertions from the beginning of this section. Thanks to including the option to unmute a muted edge (in Section 4.2), these assertions are strong enough for the guards related to $f \in v.muted$. For the guards related to $f \in v.out$ we require an extra pre-assertion.

```
parallel for f, u : u ~_f v ∧ f ≠ e do ⟨

    {? (root.e, dist.e) ≤ (r, d)} {? (∀h : ¬S.v.e ∧ S.v.h : root.h < r)}
    {? sig.v_f ∨ f ∉ v.out ∨ r < root.f ∨ root.f ≠ f}
    {sig.v_f ∨ r ≤ root.f} {sig.v_f ∨ f ∉ v.in ∨ (r, d) < (root.f, dist.f)}
  if f ∈ v.out ∧ r = root.f ∧ root.f ≠ f →
      mute f
  [] f ∈ v.out ∧ r < root.f →
      root.f, dist.f, sig.u_f := r, d + 1, true
      turn f
  [] f ∈ v.in ∧ (r, d) < (root.f, dist.f) →
      root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
  [] f ∈ v.muted ∧ r < root.f ∧ u ≠ v →
      root.f, dist.f, sig.u_f := r, d + 1, true
      unmute f as u →_f v
  [] f ∈ v.muted ∧ r ≤ root.f ∧ (r = root.f ∨ u = v) →
      root.f := r
  [] sig.v_f →
      skip
  fi ⟩
```

We first consider assertion $(\forall h : \neg S.v.e \wedge S.v.h : root.h < r)$. Although we could continue with it in its current shape, we will try to eliminate it. First (see Appendix E) we strengthen it into the more convenient condition $(\forall f : f \in v.edges : r \leq root.f) \Rightarrow S.v.e$. Using required assertion $(root.e, dist.e) \leq (r, d)$, this condition turns out to reduce to *true*, if we strengthen an invariant and an assertion by leaving out disjunct $sig.v_f$, and generalize that assertion as follows:

$\{?\ \mathbf{inv}\ e \in v.out \lor (e \in v.in \cup v.loops \land root.e = e)\}$
parallel for $f, u : u \sim_f v \land f \neq e$ **do** $\langle$

$\{?\ (root.e, dist.e) \leq (r, d)\}\ \{?\ sig.v_f \lor f \notin v.out \lor r < root.f \lor root.f \neq f\}$
$\{?\ (\forall f : f \neq e \land f \in v.in : (r, d) < (root.f, dist.f))\}$
if ... **fi** $\rangle$

Note that the strengthened invariant is indeed maintained by the algorithm; its local correctness can be achieved by leaving out disjunct $sig.v_f$ from the related assertions. The strengthened assertion is globally correct under the other nodes using the direction property, and the node itself cannot violate it. Local correctness is guaranteed, since we already discussed that disjunct $sig.v_f$ may be left out in the related pre-assertion.

$\{\mathbf{inv}\ e \in v.out \lor (e \in v.in \cup v.loops \land root.e = e)\}$
parallel for $f, u : u \sim_f v \land f \neq e$ **do** $\langle$

$\{?\ (root.e, dist.e) \leq (r, d)\}\ \{?\ sig.v_f \lor f \notin v.out \lor r < root.f \lor root.f \neq f\}$
$\{(\forall f : f \neq e \land f \in v.in : (r, d) < (root.f, dist.f))\}$
if ... **fi** $\rangle$

Global correctness of the remaining queried assertions is guaranteed using the descendence property and the signalling property respectively. Their local correctness is guaranteed by requiring the following pre-assertion of the preceding assignment:

$\{?\ (\forall f : f \neq e \land f \in v.out \land root.f = f : sig.v_f \lor root.e < f)\}$
$r, d := root.e, dist.e$

Using the descendence and the signalling property, global correctness of this queried assertion is guaranteed. For its local correctness, note that it is a post-assertion of the selection of an edge $e : S.v.e$. Therefore we strengthen $S.v.e$ with a conjunct similar to this assertion into

$$\begin{aligned} S.v.h \equiv\ & (h \in v.out \lor (h \in v.in \cup v.loops \land root.h = h)) \\ & \land (\forall f : f \neq h \land f \in v.edges : root.h \leq root.f) \\ & \land (\forall f : f \neq h \land f \in v.in : (root.h, dist.h) < (root.f, dist.f)) \\ & \land (\forall f : f \neq h \land f \in v.out \land root.f = f : root.h < f) \end{aligned}$$

By strengthening $S.v.h$ we also strengthen invariant P_0; hence we have to reconsider its correctness. Thanks to the descendence property, the additional conjunct cannot lead to more violations of $S.v.h$ in P_0 if we require the additional invariant

$$P_1 \ \ (\forall f :: root.f \leq f) \qquad ,$$

which itself is maintained by the descendence property. What remains is to ensure that whenever $S.v.h$ must be established, the additional conjunct is also established. Recall that each time that $S.v.h'$ must be established in a state where $S.v.h$ holds, it is accompanied with an assignment that establishes $root.h' < root.h$. Using $S.v.h$ the new conjunct follows from P_1.

What remains is the related assertion $(\forall h : \neg S.v.e \land S.v.h : root.h < r)$. It still reduces to *true*, if we also strengthen assertion $sig.v_f \lor f \notin v.out \lor r < root.f \lor root.f \neq f$ into $(\forall f : f \neq e \land f \in v.out \land root.f = f : r < f)$. Local correctness can be achieved by leaving out disjunct $sig.v_f$ from the related assertions. Their global correctness follows from invariant P_0.

D Buffering values

In Section 5, we claimed that statement $e : S.v.e$ can be implemented using a snapshot that is not required to be consistent. In this section we provide a correctness argument for this modification of the algorithm.

As a starting point we use the annotated algorithm from Appendix A. To formalize this snapshot, we introduce in each node fresh local variables $root'$, $dist'$ and $state'$. And we precede the selection of an edge e with for each edge $f : f \in v.edges$ an assignment $root'.f, dist'.f, state'.f := root.f, dist.f, state.v.f$, in which $state.v.f$ denotes whether edge f is an incoming edge, an outgoing edge or a muted edge with respect to node v. To show that an edge e can safely be selected using these copied values, we propose to replace the corresponding part of the algorithm of Appendix A by:

$\{v.loops \subseteq v.muted\}$
$edges' := \emptyset$

$\{\mathbf{inv}\ (\exists e : e \in v.edges : T.v.e)\}$ $\{\mathbf{inv}\ v.loops \subseteq v.muted\}$
parallel for $f : f \in v.edges$ **do**
  $sig.v_f := false$
  $root'.f, dist'.f, state'.f, edges' := root.f, dist.f, state.v.f, edges' \cup \{f\}$

$\{(\exists e : e \in v.edges : T.v.e)\}$ $\{edges' = v.edges\}$
$\{(\forall f : f \in v.in : f \in v.in')\}$ $\{(\forall f : f \in v.out \ \wedge\ root.f = f : f \in v.out' \ \wedge\ root'.f = f)\}$
$\{v.loops \subseteq v.muted\}$ $\{(\forall f : f \in v.edges : (root.f, dist.f) \leq (root'.f, dist'.f))\}$
$\{(\forall f : f \in v.edges : (sig.v_f \wedge f \notin v.in) \ \vee\ (root.f, dist.f) = (root'.f, dist'.f))\}$
$\{(\forall f : f \in v.edges : (f \in v.out' \ \vee\ (f \in v.in' \cup v.loops \ \wedge\ root'.f = f)) \Rightarrow$
  $(f \in v.out \ \vee\ (f \in v.in \cup v.loops \ \wedge\ root.f = f))\}$
e: $T.v.e$

$\{v.loops \subseteq v.muted\}$ $\{e \in v.out \ \vee\ (e \in v.in \cup v.loops \ \wedge\ root.e = e)\}$
$\{(\forall f : f \neq e \ \wedge\ f \in v.edges : sig.v_f \ \vee\ root.e \leq root.f)\}$
$\{(\forall f : f \neq e \ \wedge\ f \in v.in : (root.e, dist.e) < (root.f, dist.f))\}$
$\{(\forall f : f \neq e \ \wedge\ f \in v.out \ \wedge\ root.f = f : root.e < f)\}$

with $T.v.e \equiv (root, dist, state.v := troot, tdist, tstate).(S.v.e)$
  with for $f \in edges'$: $troot.f, tdist.f, tstate.f = root'.f, dist'.f, state'.f$
  and for $f \notin edges'$: $troot.f, tdist.f, tstate.f = root.f, dist.f, state.v.f$

We did not explicitly include all annotation for the parallel construct. The skipped parts can straightforwardly be added, and they contain nothing essentially new. Note that we used local variable $edges'$ only for the annotation; hence it can be eliminated from an implementation.

We first consider correctness of the last series of assertions. Their global correctness is maintained under our modification. Their local correctness follows from two parts: the pre-assertions of the selection of an edge e and the definition of $T.v.e$ in case $edges' = v.edges$:

$$\begin{aligned} T.v.e \ \equiv\ & (e \in v.out' \ \vee\ (e \in v.in' \cup v.loops \ \wedge\ root'.e = e)) \\ & \wedge (\forall f : f \neq e \ \wedge\ f \in v.edges : root'.e < root'.f) \\ & \wedge (\forall f : f \neq e \ \wedge\ f \in v.in' : (root'.e, dist'.e) < (root'.f, dist'.f)) \\ & \wedge (\forall f : f \neq e \ \wedge\ f \in v.out' \ \wedge\ root'.f = f : root'.e < f) \end{aligned}$$

Then we consider the pre-assertions of the selection of an edge e, apart from the first assertion. For their global correctness, note that they reflect many global correctness arguments that we used before, especially the three properties mentioned in Section 4. Usually, their local correctness can be established conjunctwise, and that is the reason that we did not include the required annotation.

Assertion and invariant $(\exists e : e \in v.edges : T.v.e)$ deserve special attention. Their local correctness is guaranteed by construction, but their global correctness could be quite complicated. Fortunately, their global correctness proofs are analogous to (the first half of) the proof of maintenance of P_0 (in Appendix C).

E Lemmas

This appendix contains some lemmas that are used in this paper.

E.1 Proof reduction for parallel constructs

Consider a parallel algorithm with a process in which the following annotated program fragment occurs:

$$\boxed{\begin{array}{l}\{A\}\\ \textbf{parallel for } x : P.x \textbf{ do}\\ \quad \{B.x\}\\ \quad \ldots\\ \quad \{C.x\}\\ \{D\}\end{array}}$$

- If assertion $B.x$ (for all $x : P.x$) is globally correct, and $A \equiv (\forall x : P.x : B.x)$, then assertion A is globally correct and assertion $B.x$ (for all $x : P.x$) is locally correct.
- If assertions A and $C.x$ (for all $x : P.x$) are globally correct, and assertion D is locally correct, then assertion D is globally correct. Nevertheless, the global correctness *proof* of assertion D may require to strengthen the assertion.

E.2 Program transformation

Consider a parallel algorithm with variables x, y and A that contains a process in which the following program fragment occurs:

$$\boxed{\begin{array}{l}x := A\\ y := A\end{array}}$$

Under the assumption that x and y are local variables (not reachable for other processes), this fragment can be correctly implemented as follows:

$$\boxed{\begin{array}{l}x := A\\ y := x\end{array}}$$

Note that this reduces the possible execution traces, but it cannot introduce a deadlock or hinder termination. Furthermore note that in both cases $\{x = A\}$ is not guaranteed to be a correct intermediate assertion.

E.3 Condition simplification

In this section we show how to rewrite and simplify some conditions. We first show that assertion $(\forall h : \neg S.v.e \wedge S.v.h : root.h < r)$ can be simplified as follows:

$$
\begin{array}{ll}
 & (\forall h : \neg S.v.e \ \wedge \ S.v.h : root.h < r) \\
\equiv & \quad \{ \text{ distribution } \} \\
 & S.v.e \ \vee \ (\forall h : S.v.h : root.h < r) \\
\Leftarrow & \quad \{ \text{ use conjunct } (\forall f : f \in v.edges : root.h \leq root.f) \text{ of } S.v.h \ \} \\
 & S.v.e \ \vee \ (\forall h :: (\exists f : f \in v.edges : root.f < r)) \\
\equiv & \quad \{ \text{ calculus } \} \\
 & S.v.e \ \vee \ (\exists f : f \in v.edges : root.f < r) \\
\equiv & \quad \{ \text{ calculus } \} \\
 & (\forall f : f \in v.edges : r \leq root.f) \ \Rightarrow \ S.v.e
\end{array}
$$