From Sequential Layers to Distributed Processes

Deriving a Distributed Minimum Weight Spanning Tree Algorithm *

(EXTENDED ABSTRACT)

Wil Janssen & Job Zwiers
University of Twente †

Abstract

Analysis and design of distributed algorithms and protocols are difficult issues. An important cause for those difficulties is the fact that the *logical structure* of the solution is often invisible in the *actual implementation.* We introduce a framework that allows for a formal treatment of the design process, from an abstract initial design to an implementation tailored to specific architectures. A combination of algebraic and axiomatic techniques is used to verify correctness of the derivation steps. This is shown by deriving an implementation of a distributed minimum weight spanning tree algorithm in the style of [GHS].

1 Introduction

Protocols for distributed systems can not only be complicated to develop but even more complicated to understand by others than the designers. Such protocols are often the result of a process of transforming, refining and optimizing a basically simple algorithm. In order to explain and clarify the final resulting protocol, as opposed to mere verification, the structure of a correctness proof should reflect the structure of the original design. A notorious example is the algorithm for determining minimum weight spanning trees by Gallagher, Humblett and Spira [GHS]. There are several published correctness proofs of the [GHS] algorithm [WLL, CG, SR], some of which rely on a protocol structure for the verification process that differs from the structure of the final algorithm. Yet we feel that these proofs fall short of clarifying certain relevant aspects of the [GHS] algorithm. In this paper we identify such aspects and we show how each of them can be understood in a series of relatively easy transformations where at each step only a few new aspects are introduced. This leads to a natural decomposition of our correctness proof that has moreover the desirable property that it closely follows a (possible) design trajectory. Explanation in the form of systematic design allows for a comparison of algorithms by means of a "genealogy"; the earlier during the design that a different design decision was taken, the more different the finally resulting algorithms are. This genealogy often suggests other algorithms and improvements. For the [GHS] algorithm we present a design trajectory that starts with an initial solution from which algorithms can be obtained as divers as the Prim and Dijkstra algorithms [Pri, Dijk], Kruskal's algorithm [Kru], Boruvka's algorithm ([Bor, Tar]) and, indeed, the algorithm of [GHS]. Already at a very early stage in our design trajectory most of these, except Boruvka's, are excluded. We thus obtain a variant where essentially the time complexity claimed by [GHS] is achieved.

The transformational design that we propose goes from a *sequential program* (essentially Boruvka's algorithm) via a *sequentially phased parallel program* to a *distributed program.* A sequentially phased parallel program [SR] can be described as a sequential composition of a number of *layers* [EF, KP], each of which is a (relatively simple) parallel program. Many protocols for distributed systems admit such a "layered" presentation which is much easier to analyze than the final distributed version. In [JPZ] a formulation of this principle in the form of an *algebraic transformation law* has been put forward.

In the present paper we apply this transformation law and show that it can be applied to a situation as complex as the GHS protocol. We do so by systematically deriving a GHS-like protocol in a number of steps, starting with a simple sequential Boruvka-like algorithm, distributing it over nodes and introducing optimizations.

*Part of this work has been supported by Esprit/BRA Project 6021 (REACT)

†Address: University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands. E-mail: {janssenw,zwiers}@cs.utwente.nl

This leads to a correctness proof in a number of relatively simple steps, reflecting decisions in the design process.

By several other authors it is also argued that the correctness proof should be able to represent the intuitive explanation given by the protocol designers. Chou and Gafni [CG] group classes of actions and define a sequential structure on such classes (so-called *stratification*). In the actual verification however they use singleton classes and a total order on actions which does not comply with the abstract protocol structure.
Stomp and de Roever on the other hand [SR] introduce what they call a *principle for sequentially phased reasoning* which allows them to introduce *semantically defined* layers that should correspond to the intuitive ideas of the designers. In [Sto] this principle is applied to the *derivation* of a broadcast protocol. The main difference to our approach that we use a formulation of this principle in an *algebraic* setting.

Both approaches are closely related to the idea of *Communication Closed Layers* by Elrad and Francez.

In order to get an idea of aspects of the GHS protocol that we can explain we provide some detail of the protocol as described in [GHS]. The protocol determines the minimum weight spanning tree (MST) of a given connected undirected graph with N nodes and E edges. A connected subgraph of the MST is called a *fragment*; virtually all algorithms for determining the MST start with trivial fragments in the form of single nodes and "grow" one or more fragments until the complete MST has been obtained. The basic principle to enlarge a fragment is to calculate its (uniquely determined) minimum-weight outgoing edge: this edge can be shown to be part of the MST. Two fragments can be combined by connecting them via edge e if e is the minimum weight outgoing edge of at least one of those fragments. In [GHS] each fragment finds its minimum-weight outgoing edge concurrently and asynchronously with regard to other fragments, and then tries to combine with the fragment at the the other end of the edge by sending a "connect" message. How and when this combination takes place is quite intricate and must be regarded as one of the typical characteristics of the GHS protocol. It depends on so called *levels* attached to fragments. Apart from single nodes which are defined to be at level 0, fragments F have a level $L > 0$ which, according to [GHS], depends on previous combinations. We quote [GHS]:

> "Suppose a given fragment F is at level $L \geq 0$ and the fragment F' at the other end of $F's$ minimum-weight outgoing edge is at level L'. If $L < L'$, then fragment F is immediately absorbed as part of fragment F', and the expanded fragment is at level L'. If $L = L'$ and fragments F and F' have the same minimum-weight outgoing edge, then the fragments combine immediately into a new fragment at level $L + 1$. In all other cases, fragment F simply waits until fragment F' reaches a high enough level for combination under the above rules."

One important reason why it is difficult to get a clear intuitive understanding of the protocol is that various fragments with totally different levels are active at the same time. Our analysis is based on a clear distinction between *causal order* and *temporal order* of events. It is shown that the apparent "chaos" from the temporal point of view corresponds to a highly regular pattern from a causal order point of view. Actually, in terms of causal order the protocol is closely related to Boruvka's algorithm, where there is a strict alternation between phases where the minimum weight outgoing edges of all fragments are determined and phases where fragments combine *until no further combination is possible anymore.* Conceptually, i.e. from a causal order point of view, *all* fragments of a given level L are created together, in a *single* phase as sketched. From a temporal point of view though, the creation of fragments, by means of creating a 'core' and followed by 'absorbing' other fragments, is spread out over time. Actually it is quite possible for a given fragment that it is already becoming part of some higher level fragment before the fragment itself is completed! Within this setting it now becomes possible to clarify an aspect such as the necessity of tagging messages with level numbers in [GHS]: In the intermediate stages of our design trajectory there are no such tags or any other (explicit) references to level numbers within the program itself. However, in order to apply our communication closed layers law we introduce separate sets of communication channels, one for each level so as to fulfill the side conditions for the law. After applying the law we have obtained a distributed algorithm with still the same sets of channels. In one of the last transformation steps we then *merge* the channels for different levels between two given nodes, by a straightforward *multiplexing* technique.

The framework we introduce in this paper allows to formulate principles like communication closed layers in a *compositional and algebraic setting.* The formulation of such laws strongly depends upon a new type of composition operator called *layer composition.* It is resembles sequential composition but allows more parallelism between actions. The definition of this operator is given in a partial order model, but does not depend on that. It relies on a symmetric and irreflexive relation between actions, called the *conflict relation*, akin to conflicts in distributed databases [BHG].

The introduction of such operators yield a language that allows us to follow the whole trajectory starting with an *initial design* that is free of architectural bias to the actual *physical implementation* where considerations like the number of processors or nodes and the network structure. The derivation steps in this trajectory cannot be done by using algebraic laws only. At some moments in the derivation process *state-based* and *axiomatic* reasoning is needed to show the correctness. By combining both styles of reasoning only, we can bridge the gap between abstract specification and low-level implementation.

Before giving the derivation we first introduce the language used and the underlying model.

2 The design language and its properties

In this section we present the language used in the derivation process and some of the properties needed. In order to get some intuition for the validity of these laws and properties we informally describe the underlying model, based on runs consisting of a set of events and a *causal* order and a *temporal* ordering relation. For a more detailed treatment of the model and the algebraic properties of our language we refer to [JPZ, Zwi].

The language that we use is intended to be appropriate for the *initial design stage* – during which we prefer to have no bias towards a certain network architecture – as well as for the description of the final program that *should* fit the network structure. The reason for having a single language rather than two separate languages, one for initial design and another for coding the final program, is that we aim at a *gradual* transformation from initial design towards final implementation, which requires a single language that can represent all stages, including intermediate ones. Since we introduce some rather unconventional language operators, which are difficult to appreciate without a basic knowledge of the underlying model, we start with a sketch of the latter.

The model

Basically, we describe the execution of distributed systems by *histories h* that consist of a partially ordered set of *events*. This model is related to the *pomset* model as introduced in [Pratt]. Typical examples of events that we actually use in this paper include send and receive actions and read and write operations to local or shared memory. The precise interpretation of an event e is determined by its *attributes* $a(e)$, some of which will be mentioned below. For each system many different histories are possible, due to different behavior of the concurrent environment of the system and other causes of nondeterminism. Therefore a system *semantically* denotes a *set of possible histories.*

Events e and f that are *unordered* in some history h, are said to be independent. Potentially such events execute in parallel, i.e. at the same time or at overlapping time intervals. Within our design formalism there are two causes for *ordering* events which consequently do *not* execute in parallel:

- The first one is because e and f *conflict* in the sense that they both access a common resource that does not allow simultaneous access. The generic example (and the terminology) stems from conflicts between concurrent database actions [BHG] due to read and write operations to the same shared memory locations. When this happens e and f simply *cannot* execute (fully) in parallel and so must *logically* be ordered, which we denote as either as $e \longrightarrow f$ or as $f \longrightarrow e$, depending on which is the case. Only conflicting actions are ordered logically.
- The second cause is that actions are *temporally ordered* as the result of the use of language operators that explicitly require such ordering. Such operators are typically used in the last design stage where actions are actually *allocated* on specific processors, or to specific network nodes. Clearly, actions that should run on a single processors have to be ordered temporally. Temporal precedence of e over f is denoted by $e \twoheadrightarrow f$.

Because of the sharp difference between logical and temporal precedence, conceptually from the point of view of a designer as well as from a more technical point of view, we use a formal semantic model where histories are structures of the form $(E, \longrightarrow, \twoheadrightarrow)$, and where E is a set of events, with a dual ordering defined on it: $(E, \longrightarrow)$ is a directed acyclic graph (DAG), i.e., the transitive closure of $\longrightarrow$ is a *partial order* on E. $(E, \twoheadrightarrow)$ is simply a partial order itself. The two ordering relations are weakly related. Temporal order obviously does not imply logical precedence. If two events e and f are *logically* ordered, say $e \longrightarrow f$, then they cannot be ordered *temporally* in the reversed direction, i.e.

$e \longrightarrow f$ implies $f \not\twoheadrightarrow e$

Also the following relation must hold

$e \twoheadrightarrow f \longrightarrow g \twoheadrightarrow h$ implies $e \twoheadrightarrow h$

Informally one can think of $e \longrightarrow f$ as e *influencing* f which cannot be the case if f completely precedes e in time. Any stronger relationship cannot be assumed; for instance from database serializability theory there are well known examples of atomic transactions e, f and g such that $e \longrightarrow f \longrightarrow g$, yet $g \twoheadrightarrow e$!

Informal semantics and algebraic properties

The two main composition operators of the language, parallel composition and conflict composition, are defined purely in terms of logical precedence, i.e. no temporal order is enforced by these operators.

The histories for a *parallel composed* system S of the form $Q \parallel R$ can be described as follows. The events executed by S in some history h can be partitioned into subhistories h_Q and h_R that are possible histories for Q and R. Moreover, the logical precedence relation *between* h_R and h_Q is such that all conflicting events are logically ordered, where the direction of the precedences are nondeterministically chosen. This nondeterminism is constrained of course by the fact that logical precedence is an *order*, so cycles of the form $e_0 \longrightarrow e_1 \longrightarrow \cdots \longrightarrow e_0$ are not allowed.

Layer composition can be considered an *asymmetric form of parallel composition.* For $Q \parallel R$ the logical precedence between conflicting actions of Q and R is nondeterministically determined. For layer composition of Q and R, which is denoted by $Q \bullet R$, actions from Q take logical precedence over actions from R in case of conflicts. In the case of independent actions no order is enforced however, just as is the case for parallel composition. We also use *iterated layer composition* $S^{\odot}$, analogously to iterated sequential composition $S^{\star}$.

Layer composition should be compared with *sequential composition* of the form $Q \,;\, R$. This is somewhat like layer composition except that we also enforce *temporal ordering* between Q and R actions: all Q actions temporally precede all R actions, regardless of conflicts. So whereas conflict composition admits parallel execution of certain actions, sequential composition does not. A sharp difference between the two forms of composition shows up when we consider Elrad and Francez' "communication closed layers" [EF]. The essence of communication closed layers is that under certain conditions a parallel system $S \parallel T$ where S and T are sequential programs of the form $S_0 \,;\, S_1$ and $T_0 \,;\, T_1$, is "equivalent" to a sequential composition of "layers" $S_0 \parallel T_0$ and $S_1 \parallel T_1$, thus:

$(S_0 \,;\, S_1) \parallel (T_0 \,;\, T_1) \equiv (S_0 \parallel T_0) \,;\, (S_1 \parallel T_1) (*)$

The side condition is that there is no communication, or in our parlance *no conflict*, between actions from S_0 and T_1, nor should there be conflicts between action from S_1 and T_0. Generalized forms of this principle appear also in [SR]. The equivalence used in $(*)$ is sometimes called IO-equivalence, referring to the fact that although the histories of left hand and right hand sides of $(*)$ are not the same, the relation between initial and final states of the system *is* the same nevertheless. A problem with this equivalence is that it is not a congruence, so we cannot simply interchange left and right hand side of $(*)$ *within contexts!* Within our framework we can replace the sequential composition in $(*)$ by conflict composition however, resulting in the following algebraic law given for the case of two layers consisting of two parallel components (with the same side conditions as for $(*)$).

$(S_0 \bullet S_1) \parallel (T_0 \bullet T_1) = (S_0 \parallel T_0) \bullet (S_1 \parallel T_1)$ (CCL)

Note that we not only have a congruence, but even semantic *equality* here, which is to be understood as the fact that both sides of the equation admit exactly the same partial order based histories. We also use a number of derived laws, see [Zwi]. A special case is the well-known independence law:

If P and Q are non-conflicting, then

$P \bullet Q = P \parallel Q$

The process term $\mathbf{io}(P)$ denotes execution of a single action that captures the net effect of executing P without admitting interference by other events. The $\mathbf{io}(\cdot)$ operation is also called the *contraction* operation, since it contracts complete P runs into single events. Intuitively $\mathbf{io}(P)$ represents the input-output behavior of a process P if we execute that process in isolation, i.e. without interference from outside. This operation induces an interesting process equivalence, called IO-equivalence, and an associated IO-refinement relation. Such equivalences play an important role in the book by Apt and Olderog [AO].

$P \overset{IO}{=} Q$ iff $\mathbf{io}(P) = \mathbf{io}(Q)$

Specification of what is often called the *functional behavior* of a process P is really a specification of $\mathbf{io}(P)$, i.e. of the IO-equivalence class of P. The $\mathbf{io}(\cdot)$ operation does (obviously) not distribute though parallel composition. For the case of layer composition we have the following law:

$P \bullet Q \overset{IO}{=} \mathbf{io}(P) \bullet \mathbf{io}(Q)$

The intuition here is that although execution of "layer" P might overlap execution of "layer" Q *temporally*, one can pretend that all of P, here represented as an atomic action $\mathbf{io}(P)$, precedes all of Q as far as IO-behavior is concerned.

IO-behavior of a system can also be *specified* by means of classical pre- and postconditions. We interpret a Hoare style formula of the form:

$\{pre\}\ S\ \{post\ \}(**)$

where *pre* and *post* are *state formulae* as usual, as follows. For each history h in $\mathbf{io}(S)$ let $s_0(h)$ and $s(h)$ denote the initial and final state of the (unique) S event in h. Then $(**)$ requires that if the initial state $s_0(h)$ satisfies precondition *pre* the corresponding final state $s(h)$ satisfies the postcondition *post*. Hoare style program verification for concurrent systems is more complicated than verification of sequential programs due to the possibility of interference. The classical proof system for shared variables by Owicki and Gries [OG] for instance includes extra interference freedom checks for assertions used in proof outlines. It has been shown by Apt and Olderog [AO] that for *restricted cases* it is possible to verify parallel programs relying on techniques for *sequential* programs however. This work relies on classical Hoare style verification in combination with program transformation based on IO-equivalence. We use similar techniques in the derivation of the algorithm, where we exploit the fact that conflict composition, although it does admit parallelism, behaves just like sequential composition when we apply the **io**() operation! This follows from the fact that the contraction of some history h can be determined without taking temporal ordering into account; logical precedence as such is sufficient to determine the cumulative state transformation associated with h.

This implies that to verify a pre-post specification for a program of the form $S \bullet T$ it suffices to verify the associated *sequential* program $S\,;T$.

In the derivation we use the combination of Hoare style formulas and program transformation to guarantee the correctness of some transformation steps. This can be seen as *proof outline transformation,* in the style of Reynolds ([Rey].) For example we have the following rule for iterated layer composition.

Define **layer** l : $P(l)$ **until** B as $(P \bullet B)^{\odot} \bullet \neg B$. If $P(l)$ is of the form

$P(l) \triangleq$ **for** $v \in V$ **dopar** $P(v)(l)$ **rof** ,

and B is of the form

$B \triangleq \forall v \in V(B(v))$,

and if furthermore the following premis are satisfied. For $l \neq l', v \neq v'$:

$P(v)(l)$ does not conflict with $P(v')(l')$

and

$\{B\}\ P\ \{B \vee (\forall v \in V(\neg B(v))\}$

then

```
  layer l
    for v ∈ V dopar P(v)(l) rof
  until B
=   { CCL- iteration }
  for v ∈ V dopar
    layer l : P(v)(l) until B
  rof
```

Informally the last premisse states that all parallel components must stop at the same number of iterations.

We conclude this section with a somewhat more detailed description of the shared memory model and the communication mechanism used in the description of the algorithm.

Shared memory and communication

In our model the basic actions are *guarded assignments* of the form

$b\&x_1, x_2, \ldots, x_m := exp_1, exp_2, \ldots, exp_m$

Informally such an assignment is postponed until the guard b holds, where after the values of the expressions exp_i are assigned simultaneously to all x_i. So our guarded assignments are really limited forms of the well known **await** statement. If the guard is *true* it is omitted.

We assume that there exists a given *conflict relation* between actions, for example two action conflict if one writes into a variable the other action reads or writes. We could also assume read-read conflict too, but will not do so in this paper. At later stages we also use communication via channels. We can model *undirectional, asynchronous* channels by shared variables. A channel c can be defined as a pair $(c.flag, c.val)$ where $c.flag$ is a boolean that is true iff a value is available on the channel, and $c.val$ the value to be read. Send and receive actions can now be modeled as guarded assignments. The *channel name* c of send and receive actions is a triple given by the node the emitting the message, the node receiving it, and a name. Let $c = (u, v, \text{MSG})$:

$send(u)(v)(\text{MSG}(e)) \stackrel{\text{def}}{=}$
$\quad \neg c.flag\&\ c.flag, c.val := true, e$

$receive(u)(v)(\text{MSG}(x)) \stackrel{\text{def}}{=}$
$\quad c.flag\&\ c.flag, x := false, c.val$

We can take a more liberal view, where we have buffered channels, which is needed in the final stages of the derivation. We do not present a full syntax of the language used in the derivation. The operators used are straightforward abbreviations of expressions using the operators given above.

3 Derivation of the algorithm

As we explained in the introduction, the derivation follows a number phases, starting of with a simple and easy to prove sequential program and finally arriving at a distributed and partially optimized set of processes. In this section we give an outline of the total derivation process and exemplify a number of crucial steps in the development. The derivation is presented as a top-down structured process. This does not comply with the true derivation process as both the initial design and the final implementation were known on beforehand. The derivation given is the result of closing the gap from both sides, eventually resulting a clear derivation showing the correctness of the distributed implementation. The final result of the derivation follows the GHS protocol closely, but has some improvements from the point of view of top-down design of programs. Furthermore not all optimizations are introduced. See [JZa] for a derivation of the whole protocol.

In the derivation we can distinguish a number of different stages, each given by a number of relatively simple transformation steps:

1. The initial (sequentially structured) design

2. Distributing data

3. Recursively computing the minimum weight outgoing edge

4. Synchronization and information diffusion by means of message passing

5. Applying the Communication Closed Layers law to get a distributed implementation

6. Multiplexing channels

7. Optimizing the algorithm

Of all these steps, the application of CCL is a purely algebraic one, although it requires some non-algebraic transformations in order to satisfy the premisses of CCL. Other parts of the derivation are proven in an axiomatic way or as a combination of both strategies. We will emphasize the first few steps and the application of the CCL laws.

The initial design closely follows the algorithm introduced by Boruvka which can be found in [Tar]. As it is a *sequentially structured* algorithm its correctness can be shown using classical Hoare style techniques [Lam]. It is not purely *sequential* however as it is formulated using layer composition instead of sequential composition. This to allow the program to be transformed and distributed using the algebraic framework we introduced. However, the overall behavior of this system can be viewed as if it executes sequentially. According to our viewpoint the use of sequential composition should be restricted to those cases when one really means to introduce a *temporal* relation instead of a causal relationship. In an initial design this is hardly ever the case, as no architectural decisions have been taken into account yet.

Before describing this algorithm we introduce some notation and theorems on minimum weight spanning trees.

3.1 Spanning trees and fragments

Assume we have a given connected and undirected graph $G = (V, E)$. We assume every edge i has a distinct weight $w(i)$. We assume all nodes have distinct names and are totally ordered. In the following let u, v, x and y denote vertices, and let $i, j, k, \ldots$ denote edges. Edges are also denoted by two-element sets $\{u, v\}$. For a graph G the following theorem holds

Theorem 3.1

If G is a connected graph where every edge has a distinct weight, the minimum weight spanning tree $MST(G)$ is uniquely determined. □

The proof can be found in [GHS].

For any node $v \in V$ let $inc(v)$ denote the set of edges incident to v, i.e.

$inc(v) \overset{\text{def}}{=} \{ \{v, u\} \in E\}$

For an edge $j = \{u, v\}$ let the *destination* of j with respect to u, $dest(u)(j)$ be v. We also use the source or destination of an edge w.r.t. a fragment or set of nodes, e.g. for fragment f and edge $j = \{u, v\}$ such that $v \notin f$ and $u \in f$ we have that $src(f)(j) = u$.

A *fragment* is a connected subgraph of $MST(G)$. For any fragment f let $\mu(f)$ be the minimum weight outgoing edge of f.

The basic idea of the algorithm follows from the following lemma, which is proven in [GHS].

Lemma 3.2

Let $G = (V, E)$ be a connected graph where every edge has a distinct weight, and let f be a fragment of $MST(G)$. If k is the minimum weight outgoing edge of f then joining k and its adjacent nonfragment node to f yields another fragment of $MST(G)$. □

In the same way we can combine two *fragments* with a connecting minimum weight outgoing edge into a new fragment.

The algorithm we introduce in the next section is based on Boruvka's algorithm [Bor, Tar]. The rough idea is as follows. We compute a set of fragments *frag* by iteratively combining fragments and their minimum weight outgoing edges. Initially *frag* is the set of all fragments that consist of a single node and no edges (which is a fragment by definition). Then every fragment determines it minimum weight outgoing edge and combines with the fragment on the other side of the edge. If two fragments share the same minimum weight outgoing edge j, then j is called the *core* of the newly formed fragment. The node adjacent to the core with the least name is called the *core node.* [1] If a fragment is not combined via a core to another fragment it is said to be *absorbed.*

The algorithm terminates when we only have a single fragment left, $MST(G)$.

Every fragment in this algorithm consists of a *core node* that is the root of the tree consisting of all other branches and other nodes in the fragment. This tree structure is used to gather information in the tree or to broadcast information.

In the derivation we also need the following lemma. One of the characteristic features of the GHS protocol – postponement of absorption – is partially based on this lemma.

Lemma 3.3

Let f be a fragment and let j be an edge of f. Removing j (but not its endpoints) from f disconnects f in two disjoint trees, at least one of which – say f_1 – does *not* contain the core of f. (if j is the fragment core any of the two subtrees can be taken.) We then have that $\mu(f_1) = j$.

Proof: see [JZa] □

[1]This is different from [GHS] where both nodes adjacent to a core play equivalent roles. In the top-down design of the algorithm the choice for a single core node is more straightforward and leads to more elegant solutions, without essentially changing the ideas of GHS. We therefore take this choice.

In the rest of this paper we furthermore use the following operations on graphs and trees. Let $G = (V, E)$ and $H = (V', E')$ be graphs. We now define the union of G and H as

$G \cup H \overset{\text{def}}{=} (V \cup V', E \cup E')$

For node v and edge k let

$v \in G \overset{\text{def}}{=} v \in V$ and $k \in G \overset{\text{def}}{=} k \in E$,

and

$G \cup \{v\} \overset{\text{def}}{=} (V \cup \{v\}, E)$,

$G \cup \{k\} \overset{\text{def}}{=} (V, E \cup \{k\})$.

Furthermore, for any fragment f let $inc(f)$ be the set of edges leaving f (i.e. the set of all $\{u, v\}$ such that $u \in f$ and $v \notin f$). For a node u we define $out(f)(u)$ as the set of edges incident to u leaving f, i.e.

$out(f)(v) \overset{\text{def}}{=} inc(v) \cap inc(f)$

If f is clear from the context it is omitted.

We define the minimum weight edge of a non-empty set E, $min\_edge(E)$ as

$min\_edge(E) \overset{\text{def}}{=} e \in E$ such that
$w(e) = MIN\{w(e') \mid e' \in E\}$

and define $min\_edge(\emptyset) \overset{\text{def}}{=} nil$. We take $w(nil) \overset{\text{def}}{=} \infty$. Finally, for a graph $G = (V, E)$, let $concomp(G)$ be the set of connected componenents in G, i.e. the set of maximal and connected graphs in G.

3.2 The initial design

The first implementation is based on the construction of a set of fragments *frag* which determine their minimum weight outgoing edges and combine connected fragments. Initially the set *frag* consists of all trees consisting of a single node and no edges, which are by definition fragments. Furthermore we compute the set of edges B that are branches, i.e. that are part of the spanning tree. This is the basic idea of Boruvka's algorithm. It can be described as:

```
$MST_0 \triangleq$
  $B := \emptyset \bullet$
  $frag := concomp((V, B)) \bullet$
  **layer**
    $M := \{min\_edge(inc(f)) \mid f \in frag, inc(f) \neq \emptyset\} \bullet$
    $B := B \cup M \bullet$
    $frag := concomp((V, B))$
  **until** $M = \emptyset$
```

The total correctness of this algorithm follows from loop invariant I_0, the definition of $\mu(f)$, and bound function τ, that are defined as:

$I_0 : |frag| \geq 1 \land$
$\forall f \in frag(f$ is a subtree of $MST(G)) \land$
$\{V' \mid (V', E') \in frag\}$ is a partitioning of V

and

$\tau \overset{\text{def}}{=} \log(|frag|)$

The invariance follows from the initialization and lemma 3.2. The number of fragments $|frag|$ is at least divided by two in each iteration and therefore $\log(|frag|)$ decreases. From the invariant and the termination condition the postcondition

$P_0 : frag = \{MST(G)\}$

easily follows.

Although we take MST_0 as the initial design in our trajectory, it is also possible to give an even more general algorithm that comprises the Prim-Dijkstra and Kruskal algorithms, by not adding M to B, but only adding a subset of M to B. In that case however we loose the logarithmic complexity of the algorithm, as the number of fragments decreases, but is not necessarily divided by two.

As a second step we split the computation of M into the layered computation of the minimum weight outgoing edge for every fragment. The reason for doing so is that we want to distribute data. Firstly per fragment, eventually per node. We do so by introducing a variable $mo(f)$ for every fragment f. This is an instance of straightforward top-down design and Hoare style verification for sequential programs. The following representation function for M will hold:

$M(mo) \overset{\text{def}}{=} \{mo(f) \mid f \in frag, inc(f) \neq \emptyset\}$

This leads to the following refined program:

```
$MST_1 \triangleq$
  $B := \emptyset \bullet$
  $frag := concomp((V, B)) \bullet$
  **layer**
    **for** $f \in frag$ **layer**
      **if** $inc(f) \neq \emptyset$ **then**
        $mo(f) := min\_edge(inc(f)) \bullet$
        $B := B \cup mo(f)$
      **else** $mo(f) := nil$
      **fi**
    **rof** $\bullet$
    $frag := concomp((V, B))$
  **until** $\bigwedge\{mo(f) = nil \mid f \in frag\}$
```

The correctness of this transformation step can be proven by means of the representation function M and the structure of the conditional statement that implies that

$mo(f) = nil$ iff $inc(f) = \emptyset$

as $inc(f) \neq \emptyset$ implies $min\_edge(inc(f)) \neq nil$.

3.3 Distributing data

The transformation of MST_1 to a program where all data are distributed takes a number of steps. First we will distribute B by introducing variables SE for every edge. This allows for the introduction of parallelism between the different fragments as conflicts due to the acces to B are resolved.

Thereafter we introduce variables $lmo(v)$ giving the local minimum weight outgoing edges of every node of every fragment. Finally we introduce a core node for every fragment, by giving defining boolean variables $core(v)$ for every node v, such that for every fragment there is a single node in the fragment with $core(v)$.

We first introduce variables

$SE(u)(v) \in \{branch, basic\}$

for every $\{u, v\} \in E$. The following representation function $B(SE)$ will hold:

$B(SE) \overset{\text{def}}{=} \{\, \{u, v\} \in E \mid SE(u)(v) = branch\}$

The transformation consists of adding initialization of every $SE(u)(v)$ and of replacing

$B := B \cup \{mo(f)\}$

by

$SE(src(f)(mo(f)))\ (dest(f)(mo(f))) := branch$

The guard $inc(f) \neq \emptyset$ can also be replaced by $mo(f) \neq nil$ after computing $mo(f)$ as $min\_edge(\emptyset) = nil$.

The correctness of this step can be proven by VDM style data refinement with atomicity constraints. These constraints are fulfilled as every layer is interference free, because all actions are placed in the same layer and no parallel interfering processes exist.

After this transformation we can replace the layered construct **for** $f \in frag$ **layer** by **for** $f \in frag$ **dopar** , as all conflict are resolved. The correctness of this transformation is guaranteed by the independence law.

The code of the resulting program is omitted.

Now we introduce local minimum weight outgoing edges for every node in every fragment, $lmo(v)$. For these variables the following invariant must hold.

$I_1 :\ mo(f) = min\_edge(\{lmo(f) \mid f \in frag\})$

This, plus the previous changes, leads to the following algorithm:

```
$MST_2 \triangleq$
  $Init \bullet$
   layer
     for $f \in frag$ dopar
       for $v \in f$ dopar
         $lmo(v) := min\_edge(out(f)(v))$
       rof $\bullet$
       $mo(f) := min\_edge(\{lmo(v) \mid v \in f\}) \bullet$
       if $mo(f) \neq nil$ then
         $SE(src(f)(mo(f)))\ (dest(f)(mo(f))) := branch$
       fi
     rof $\bullet$
     $frag := concomp((V, B(SE)))$
   until $\bigwedge\{mo(f) = nil \mid f \in frag\}$
```

where

```
 $Init \triangleq$
    for $v \in V$ dopar
      for $\{u, v\} \in inc(v)$ dopar
        $SE(u)(v) := basic$
      rof
    rof $\bullet$
    $f := concomp((V, B(SE)))$
```

The correctness of these transformation steps can again be easily verified (see [JZa]).

In MST_2 we still have the set of fragments $frag$ as a variable. This information must be localized too. We do so by introducing boolean variables $core(v)$ for every node v, and defining a function $\mathcal{F}rag(u)$ giving the fragment u belongs to, i.e. the set of nodes and edges connected to u by branches.

$\mathcal{F}_b(v) \overset{\text{def}}{=} \{u \in V \mid connected(v, u)\}$

$\mathcal{F}rag(v) \overset{\text{def}}{=}$

$(\mathcal{F}_b(v), \{\, \{u, u'\} \in \mathcal{F}_b(v)^2 \mid SE(u)(u') = branch\})$

where *connected* means that v and u are connected via a path of branches. We leave its definition implicit.

We define the following representation function and (data) invariants:

$F(core) \overset{\text{def}}{=} \{\mathcal{F}rag(u) \mid u \in V, core(u)\}$

$MO(\mathcal{F}rag(u)) \overset{\text{def}}{=} mo(v)$ such that $v \in \mathcal{F}rag(u) \wedge core($

$I_2 :\ \forall v \in V\, (\exists! u \in \mathcal{F}rag(v)\, (core(u)))$

$I_3 :\ \forall \{u, v\} \in E\, (SE(u)(v) = SE(v)(u))$

Data invariant I_3 is now needed to guarantee the correctness of the definition of *connected*. Furthermore we will make use of this in later stages.

We define the set *Core* as

$Core \overset{\text{def}}{=} \{v \in V \mid core(v)\}$

How can we now achieve I_2, i.e. how do we choose a unique core node for every fragment? Initially this is no problem: every fragment consists of a single node. After that we know that for every new fragment there were two exactly subfragments that had the same minimum weight outgoing edge, the core edge. As all nodes are ordered we can take the first of the nodes adjacent to the core. In order to determine which nodes are adjacent to minimum weight outgoing edges we introduce variables $con\_req(u)(v)$ stating that $\{u, v\}$ was the minimum weight outgoing edge of $\mathcal{F}rag(u)$. The following invariant will therefore hold:

$I_4 :\ \forall u \in Core\ (con\_req(v)(v')$ iff

$\{v, v'\} = mo(u) \wedge v \in \mathcal{F}rag(u))$

Apart from some minor changes in MST_1 we have to establish I_4 at the end of the loop, i.e. the final statement in the **layer** ... **until** will be *ComputeCore*, defined as

```
$ComputeCore \triangleq$
  for $u \in Core$ dopar $core(u) := false$ rof •
  for $v \in V$ dopar
    for $\{v, x\} \in inc(v)$ dopar
      if $con\_req(v)(x)$ then
        $con\_req(v)(x) := false$ •
        if $SE(v)(x) = branch$ then
          if $v < x$ then
            $core(v) := true$
          fi
        else $SE(v)(x) := branch$
        fi
      fi
    rof
  rof
```

The correctness of this transformed MST_2, which we will call MST_3, can again be proven by proof outline transformation and Owicki-Gries style verification with trivialized interference freedom tests because of local variables. In this proof lemma 3.2 and theorem 3.1 are needed. The proof itself is omitted.

3.4 Recursively computing the minimum weight outgoing edge

Now we have introduce core nodes we can make use of the fact that we can view a fragment as a rooted tree to compute $mo(u)$ for every core node u. To do so we introduce variables $up(v)$ denoting the edge toward the root of node v in the fragment (for $\neg core(v)$). Furthermore we introduce a variable $mo(v)$ for every node v, not only core nodes, denoting the minimum weight outgoing edge of the tree in the fragment of which v is the root.

We also need to *synchronize* the nodes in a fragment: a node v needs the values of all its successors in the tree to compute $mo(v)$. For this purpose we define synchronization flags $mo\_comp(u)$ that are set to true when the value of $mo(u)$ has been computed. Synchronization is also needed to inform edges that have to change their root and upward edge, i.e. nodes that are on the path from the core to the minimum weight outgoing edge, as we do not want to implement this by core actions only. For this purpose we introduce three-valued flags $change(u) \in \{true, false, \perp\}$. In the algorithm below these synchronization variables are indexed by the number of the layer. This in order to guarantee synchronization w.r.t. to *that* layer, or, viewed differently, to make the layers communication closed. We come back to that later.

The following notation is used:

$$down(v) \stackrel{\text{def}}{=} \{j = \{v, u\} \in inc(v) \mid SE(v)(u) = branch \wedge j \neq up(v)\},$$

$$tree(v) \stackrel{\text{def}}{=} \{(\{v\}, down(v))\} \cup \bigcup\{tree(u) \mid \{v, u\} \in down(v)\},$$

and

$$root(v) \stackrel{\text{def}}{=} \begin{cases} v, & \text{iff } core(v) \\ root(dest(v)(up(v))), & \text{iff } \neg core(v) \end{cases}$$

We furthermore define the path between two (connected) nodes u and v as the sequence of nodes on the path. The full definition is omitted. For a path $p = [uvw \ldots]$ we define the first edge of p, $first(p)$, as the pair $\{u, v\}$.

For the next algorithm MST_4 the following invariants will hold:

I_5 : $mo\_comp(v) \Rightarrow mo(v) = min\_edge\{mo(u) \mid u \in tree(v)\}$

I_6 : $(mo\_comp(v) \wedge down(v) = \emptyset) \Rightarrow mo(v) = lmo(v)$

I_7 : $\forall u, v \in V((core(v) \wedge connected(u, v)) \Rightarrow up(u) = first(path(u, v)))$

For sake of brevity we immediately introduce a second addition in this algorithm. Let $be(v)$ be the edge leading to the minimum weight outgoing edge, i.e.

I_8 : $\forall u, v \in V((u \neq v \wedge lmo(u) = mo(v) \wedge connected(v, u)) \Rightarrow be(v) = first(path(v, u)))$

I_9 : $lmo(u) = mo(u) \Rightarrow be(u) = lmo(u)$

All this leads to the following algorithm:

```
$MST_4 \triangleq$
  $Init$ •
  layer $l$
    $ComputeLocal(l)$ •
    $ComputeGlobal(l)$ •
    $ChangeRootPath(l)$ •
    $ComputeCore(l)$
  until $\bigwedge\{mo(v) = nil \mid v \in V\}$
where
 $Init \triangleq$
   for $u \in V$ dopar
     $core(u) := true \| up(v) := nil \| be(v) := nil \|$
     for $\{u, v\} \in inc(u)$ dopar
       $SE(u)(v) := basic \| con\_req(u)(v) := false$
     rof
   rof ,
 $ComputeLocal(l) \triangleq$
   for $u \in Core$ dopar
     for $v \in \mathcal{F}rag(u)$ dopar
       $lmo(v) := min\_edge(out(\mathcal{F}rag(u))(v)) \|$
       $mo\_comp(v)(l) := false$   rof
   rof ,
$ComputeGlobal(l) \triangleq$
  for $u \in Core$ dopar
    for $v \in \mathcal{F}rag(u)$ dopar
      $mo(v), be(v) := lmo(v), lmo(v)$
      for $\{v, x\} \in down(v)$ dopar
        await $mo\_comp(x)(l)$ do
          if $w(mo(x)) < w(mo(v))$ then
            $mo(v), be(v) := mo(x), \{v, x\}$
          fi
        od
      rof ;
      $mo\_comp(v)(l) := true$
    rof
  rof
```

Note that we need the sequential composition at this stage to enforce the right moment of synchronization. Furthermore let

```
$ChangeRootPath(l) \triangleq$
  for $u \in Core$ dopar
    $change(u)(l) := (mo(u) \neq nil) \bullet$
    for $v \in \mathcal{F}rag(u)$ dopar
      await $change(v) \neq \bot \bullet$
      if $change(v)(l)$ then
        $up(v) := be(v) \bullet$
        if $SE(v)(dest(v)(be(v)) = branch$ then
          $change(dest(v)(be(v)))(l) := true$
        else
          $SE(v)(dest(v)(be(v)) := branch \bullet$
          $con\_req(v)((dest(v)(be(v)) := true$
        fi
      fi $\bullet$
      for $i \in down(v) - \{be(v)\}$ dopar
        $change(dest(v)(i))(l) := false$
      rof
    rof
  rof
```

and $ComputeCore(l)$ analogously to MST_3.

The correctness of the above solution is quite involved as it includes proving deadlock freedom and correctness of the recursive definition. [2] The proof however can be restricted to a single layer within a single execution of the loop which simplifies matters to a large extend. It can be proven correct using Hoare logic [Lam] or temporal logic [MP].

3.5 Introducing message passing

In MST_4 we had to introduce variables to synchronize actions and we had to copy values computed. As we are thriving for a distributed solution it is very well possible to introduce *communication over channels* to enforce synchronization and to pass values. As send and receive actions are defined as guarded assignments this transformation is straightforward, and simplifies matters to a large extend.

Furthermore we want to remove all shared accesses from the algorithm, as these are not possible in distributed implementations. We therefore have to adapt the computation of lmo, remove the shared accesses to con_req, $mo(x)$, and references to $\mathcal{F}rag(u)$. This is done by introducing message passing and variables $fn(v)$ denoting the fragment name of v.

Some further simplifications are possible: we hardly ever refer to the edge $mo(v)$, but often to its weight. We therefore use variables bw instead of mo. Also the variables lmo are oblivious as their function can be taken by bw. Finally we can join the parallel executions over all core nodes and all nodes in the corresponding fragment to the parallel execution over every node in V.

The result of these transformations, MST_5 has the following structure. The full code is omitted because of space limitations.

```
$MST_5 \triangleq$
  for $v \in V$ dopar $Init(v)'$ rof $\bullet$
  layer $l$
    for $v \in V$ dopar $ComputeLocal(v)(l)'$ rof $\bullet$
    for $v \in V$ dopar $ComputeGlobal(v)(l)'$ rof $\bullet$
    for $v \in V$ dopar $ChangeRootPath(v)(l)'$ rof $\bullet$
    for $v \in V$ dopar $ComputeCore(v)(l)'$ rof $\bullet$
    for $v \in V$ dopar $ChangeName(v)(l)'$ rof
  until $\bigwedge\{bw(v) = \infty \mid v \in V\}$
```

The processes $ComputeLocal'$ and $ComputeCore'$ can both be split into two processes by means of algebraic transformations and proof outline transformations. The former can be split into a kernel process concerned with computing $be(v)$ and a test handler $TH(v)$ reacting upon TEST messages sent by other processes, the latter into a process possibly trying to connect and a connect handler $CH(v)$ responding to CONNECT messages of other nodes.

In this process we use the rule that if there are only a single send and a single receive action on a channel, executing them in parallel is the same as first sending and then receiving (see [JZ]).

We define $basic(v) \overset{\text{def}}{=} \{\{v, x\} \in inc(v) \mid SE(v)(x) = basic\}$. The processes $TH(v)$ and $CH(v)$ have the following form:

```
$TH(v) \triangleq$
  for $\{v, x\} \in basic(v)$ dopar
    $receive(v)(x)(\text{TEST}(fn(x)(v))) \bullet$
    if $fn(x)(v) = fn(v)$ then $send(v)(x)(\text{REJECT})$
    else $send(v)(x)(\text{ACCEPT})$
    fi
  rof
```

[2] The solution given above deviates from [GHS] in the fact that we only change $up(v)$ on the path from the core to the new minimum weight outgoing edge. In [GHS] every $up(v)$ variable is reset in every iteration when an INITIATE is received.

```
$CH(v) \triangleq$
  for $\{v, x\} \in basic(v)$ dopar
    $send(v)(x)(\text{NoConnect})\|$
    $(receive(v)(x)(\text{NoConnect})$ or
    $(receive(v)(x)(\text{Connect}) \bullet SE(v)(x) := branch))$
  rof
```

3.6 Applying Communication Closed Layers

What we eventually want to arrive at is a distributed implementation, i.e. an implementation of the form

```
$MST_f \triangleq$
  for $v \in V$ dopar
    $Init(v) \bullet$
     layer $P(v)$ until $B(v)$
  rof
```

The algorithm MST_5 however still is of a sequential nature. We want to apply the Communication Closed Layers Law to transform MST_5 to a distributed structure. To be able to apply the CCL law or its iterated version MST_5 must be of the correct structure and fulfill the premisses.

In order to arrive at the structure desired we first have to transform the loop body. This consists of a given number of layers that are all of the form

$L_i \triangleq$ **for** $v \in V$ **dopar** $L_i(v)$ **rof**

This is the correct structure to apply CCL. Furthermore all layers are communication closed as communication takes place within a layer, and other conflicts only exists within the process of a single node. We can transform the loop body L now as follows:

```
      $L$
=     { by definition }
      $(ComputeLocal' \| TH) \bullet$
      $ComputeGlobal' \bullet$
      $ChangeRootPath' \bullet$
      $(ComputeCore' \| CH) \bullet$
      $ChangeName'$
=     { $\|$ is commutative and associative }
      for $v \in V$ dopar $ComputeLocal(v)' \| TH(v)$ rof $\bullet$
      for $v \in V$ dopar $ComputeGlobal(v)'$ rof $\bullet$
      for $v \in V$ dopar $ChangeRootPath(v)'$ rof $\bullet$
      for $v \in V$ dopar $ComputeCore(v)' \| CH(v)$ rof $\bullet$
      for $v \in V$ dopar $ChangeName(v)'$ rof
=     { CCL }
      for $v \in V$ dopar
        $(ComputeLocal(v)' \| TH(v)) \bullet$
        $ComputeGlobal(v)' \bullet ChangeRootPath(v)' \bullet$
        $(ComputeCore(v)' \| CH(v)) \bullet$
        $ChangeName(v)'$
      rof
=     { by definition $P(v)$ }
      for $v \in V$ dopar $P(v)$ rof
=     $L'$
```

We have now transformed L to a form suitable for the application of CCL. The guard of the loop however do not satisfy the premis of the iterated CCL rule:

$\{B\}\ P\ \{B \vee (\forall v \in V(\neg B(v))\}$

The last layer of the loop however consists of a broadcast of the name of the fragment. We change *ChangeName* in such a way that the new fragment name is *term*. This allows us to restate the termination condition as:

$B' \triangleq \bigwedge\{fn(v) = term \mid v \in V\}$

This condition does satisfy the premis, as

$\exists v \in V(fn(v) = term) \Rightarrow \forall v \in V(bw(v) = \infty)$

We can now transform MST_5 as follows:

```
      $MST_5$
=     { by definition }
      $Init' \bullet$ layer $l : L$ until $B'$
=     { $L = L'$ }
      $Init' \bullet$ layer $l : L'$ until $B'$
=     { definition $L'$ }
      $Init' \bullet$
       layer $l$
         for $v \in V$ dopar $L(v)'$ rof
       until $B'$
=     { iterated CCL }
      $Init' \bullet$
       for $v \in V$ dopar
         layer $l : P(v)$ until $fn(v) = term$
       rof
=     { CCL }
       for $v \in V$ dopar
         $Init(v)' \bullet$ layer $l : P(v)$ until $fn(v) = term$
       rof
=     { by definition }
       for $v \in V$ dopar $N(v)$ rof
=     $MST_6$
```

The result of these transformations MST_6 has the desired distributed structure.

3.7 Multiplexing channels and optimizing messages

In MST_6 we used different channels for different layers. This asssumption is not realistic but it is possible to multiplex a number of channels on a single (buffered) channel. The buffer length must be at least the number of layers that are executed, which is limited by

$\log(|V|)$. A channel c is in this case not given by a pair $(flag, val)$, but by a function $c : [0..l] \rightarrow (flag, val)$. We can now assume that the level number is tagged to every message and we can implement a send action $send(v)(u)(\text{MSG}(e, ln))$ as

$send(v)(u)(\text{MSG}(e, ln) \triangleq$
$(\neg c(ln).flag) \& c(ln).flag, c(ln).val := true, e$

By now we have transformed the algorithm in MST_6 where $N(v)$ has a structure as in fig. 1, where some subroutines have been left implicit. In this algorithm a number of optimizations are possible. First of all we can group statements for core and non-core nodes by using proof outlines and transforming them. By introducing the right invariants we can furthermore show that some messages (e.g. NoConnect and NoChange) are oblivious and can be removed.

There are also some other optimizations possible w.r.t. to the number of messages: if we received a Reject message on some basic edge, we will always receive a Reject message. Introducing a *reject* status for edges will save double status requests. We can also check basic edges one by one, instead of all in parallel. In that case we have to check them in order of weight. This optimization allows us to postpone the absorption of fragments: if the edges allow which the fragment sent a Connect has to large a weight, lemma 3.3 guarantees that we can absorb it at a later stage.

The details of these optimizations can be found in the full paper ([JZa]).

The final implementation step is now to replace *layer composition* by *sequential composition,* and to replace *parallel composition* by *sequential iteration* within a component. This does not invalidate the correctness (for non-interfering parallelism) and results in an *implementable, distributed algorithm.*

4 Conclusion

The layering techniques used to derive the implementation of a distributed minimum weight spanning tree algorithm have proven to be a powerful means in the development of parallel systems. This also holds for a posteriori verification where it can give insight in the structure of the implementation and the intuitive ideas of the designers.

These techniques are applicable to a large number of problems, not only to this type of algorithms. Other examples – varying from parsing algorithms to database protocols – can be found in [JZ], [JPZ], and [PZ].

At this moment we are investigating the relation between the process based approach as used in this paper and logic based approaches to layering like [KP]. The use of non-static dependency relations might be useful in our context too.

Also algorithms relying on real-time synchronization

```
$N(v) \triangleq$
  $up(v) := nil \| core(v) := true \| fn(v) := v \|$
   for $\{v, x\} \in inc(v)$ dopar
     $se(v)(x) := basic \bullet con\_req(v)(x) := false$
   rof $\bullet$
   layer $l$
     $be(v), bw(v) := nil, \infty$
     ( for $\{v, x\} \in basic(v)$ dopar
         $send(v)(x)(\text{TEST}(fn(v), l)) \bullet$
         $(receive(v)(x)(\text{REJECT}(l))$ or
         $(receive(v)(x)(\text{ACCEPT}(l)) \bullet$
          $\langle$ if $w(\{v, x\}) < be(v)$ then
            $bw(v), be(v) := w(\{v, x\}), \{v, x\}$
           fi $\rangle))$
       rof $\| TH(v)) \bullet$
       for $\{v, x\} \in down(v)$ dopar
         $\langle receive(v)(x)(\text{REPORT}(b(v), l) \bullet$
          if $b(v) < bw(v)$ then
            $be(v), bw(v) := \{v, x\}, b(v)$
          fi $\rangle$
       rof $\bullet$
       if $\neg core(v)$ then
         $send(v)(dest(v)(up(v))(\text{REPORT}(bw(v), l))$
       fi $\bullet$
       if $core(v)$ then $ChangeRootPath$
       else
         $(receive(v)(dest(v)(up(v)))(\text{CHANGEROOT}(l)) \bullet$
         $up(v) := be(v) \bullet ChangeRootPath)$ or
         $(receive(v)(dest(v)(up(v)))(\text{NOCHANGE}(l)) \bullet$
         $NoChangeRoot)$
       fi $\bullet$
     $core(v) := false \bullet$
     ( if $con\_req(v)(dest(v)(be(v))$ then
         $send(v)(dest(v)(be(v))(\text{CONNECT}(l)) \bullet$
         $con\_req := false \bullet$
         $(receive(v)(dest(v)(be(v))(\text{NOCONNECT}(l))$ or
          $(receive(v)(dest(v)(be(v))(\text{CONNECT}(l)) \bullet$
          $core(v) := v < dest(v)(be(v))\ ))$
       fi $\| CH(v)) \bullet$
       if $core(v)$ then
          if $bw(v) = \infty$ then $fn(v) := term$
          else $fn(v) := v$ fi
       else
         $receive(v)(dest(v)(up(v))(\text{INITIATE}(fn(v), l))$
       fi $\bullet$
     $BroadCastName(v)$
   until $fn(v) = term$
```

Figure 1:

like atomic broadcast protocols [CASD] are studied in our framework.

References

[AO] K.R. Apt, E.-R. Olderog, *Verification of sequential and concurrent programs*, Springer, 1991.

[BHG] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bor] O. Boruvka, O jistém problému minimálním, *Práca Moravské Přírodovědecké Společnosti,* 3, 1926. (In Czech.)

[CG] C. Chou and E. Gafni, Understanding and Verifying Distributed Algorithms Using Stratified Decomposition, *Proc. of the 7th Annual Symp. on Principles of Distributed Computing,* 1988.

[CM] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[CASD] F. Critian, H. Aghili, R. Strong, D. Dolev, Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement, *Proceedings 15th International Symposium on Fault-Tolerant Computing,* 1985.

[Dijk] E. Dijkstra, Two Problems in Connection with Graphs, *Numer. Math.*, Vol. 1, pp. 269-271, 1959.

[EF] T. Elrad and N. Francez, Decomposition of distributed programs into communication closed layers, *Science of Computer Programming 2*, 1982.

[GHS] R.T. Gallager, P.A. Humblet and P.M. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM TOPLAS 5-1*, 1983.

[JPZ] W. Janssen, M. Poel and J. Zwiers, Action Systems and Action Refinement in the Development of Parallel Systems, an Algebraic Approach, *proceedings CONCUR '91*, Springer LNCS 527, 1991.

[JZ] W. Janssen and J. Zwiers, Protocol Design by Layered Decomposition, A compositional approach, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems,* LNCS 571, 1992.

[JZa] W. Janssen and J. Zwiers, From Sequential Layers to Distributed Processes: Deriving a Distributed Minimum Weight Spanning Tree Algorithm, *Memoranda Informatica,* University of Twente, 1992.

[KP] S. Katz and D. Peled, Verification of Distributed Programs using Representative Interleaving Sequences, to appear in: *Distributed Computing.*

[Kru] J. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, in: *Proc. of the American Mathematical Society,* Vol. 7, pp. 48-50, 1956.

[Lam] L. Lamport, The Hoare Logic of concurrent programs, *Acta Informatica 14,* 1980.

[MP] Z. Manna, A. Pnueli, Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, *Science of Computer Programming 4,* 1984.

[OG] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informatica 6*, 1976.

[Pratt] V. Pratt, Modelling Concurrency with Partial orders, *International Journal of Parallel Programming 15*, 1986, pp. 33-71.

[Pri] R. Prim, Shortest connection networks and some generalizations, *Bell Syst. Tech. Journal,* Vol. 36, pp. 1389-1401, 1957.

[PZ] M. Poel and J. Zwiers, Layering Techniques for the Development of Parallel Systems, An Algebraic Approach, to appear in: *Proc. Computer Aided Verification*, 1992.

[Rey] J.C. Reynolds, *The Craft of Programming,* Prentice Hall, 1981.

[SR] F.A. Stomp and W.P. de Roever, Designing distributed algorithms by means of formal sequentially phased reasoning, *Proc. of the 3rd International Workshop on Distributed Algorithms*, Nice, LNCS 392, Eds. J.-C. Bermond and M. Raynal, 1989, pp. 242-253.

[Sto] F.A. Stomp, A derivation of a broadcasting protocol using sequentially phased reasoning, in: *Stepwize Refinement of Distributed Systems,* J.W. de Bakker et al. (Eds), Springer LNCS 430, 1989.

[Tar] R.E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Apllied Mathematics, 1983.

[WLL] J.L. Welch, L. Lamport, N. Lynch, A Lattice-Structured Proof Technique Applied to a Minimum Weight Spanning Tree Algorithm, *Proc. of the 7th Annual Symp. on Principles of Distributed Computing,* 1988.

[Zwi] J. Zwiers, Layering and Action Refinement for Timed Systems, in: *Proc. REX Workshop on Real Time: Theory in Practice,* Springer Lecture Notes in Computer Science, 1992.