On the Verification of a Self-Stabilizing Algorithm

Stephan Merz
merz@informatik.uni-muenchen.de

October 13, 1998

1 The Problem

In a seminal paper [3], Dijkstra introduced the concept of self-stabilization and presented three elementary self-stabilizing algorithms. Regarding their correctness, he wrote

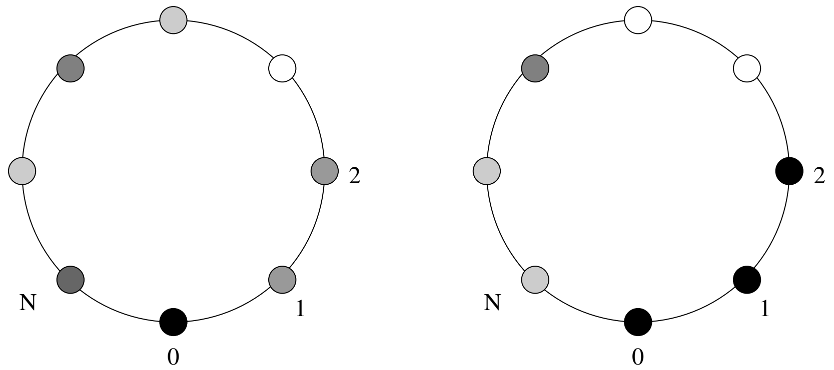
> For brevity's sake most of the heuristics that led me to find them, together with the proofs that they satisfy the requirements, have been omitted, and [...] the appreciation is left as an exercise for the reader.

More than a decade later Dijkstra published a proof for one of the three algorithms [4], acknowledging that the verification was not entirely trivial. After almost 25 years one should think that a machine proof of such algorithms should now be a routine task, given the techniques and machine support for formal analysis of concurrent algorithms that have meanwhile been developed. Nevertheless, Qadeer and Shankar [9] have recently presented a formal verification of one of Dijkstra's algorithms using the interactive prover PVS [7] and have reported considerable difficulty in constructing a convincing informal proof as well as in its formalization. This note presents an alternative correctness proof that is somewhat more economical and also easier to follow. We have found it essentially straightforward to formalize this proof in Isabelle/HOL [8]. We also discuss how to present this proof, and argue in favor of a diagrammatic presentation similar to techniques described in [2], both as a means to explain the structure of the argument and as a basis to combine interactive and algorithmic proof techniques.

A system is self-stabilizing if it will converge to some desirable ("stable") state no matter which state it is started in. Self-stabilization is important in bootstrapping distributed systems such as communication networks without central control, and also in error recovery. Dijkstra's algorithm considers a ring of $N + 1$ processors, with $N > 0$. The system is stable if and only if there exists a unique token in the ring. The processors are numbered $0, 1, \ldots, N$; each processor has a register that can hold values from the set $\mathit{Vals} = \{0, 1, \ldots M\}$, where $M \geq N$. Sample configurations of the system are illustrated in figure 1: different register values are symbolically indicated by different shades of the circles that represent the processors. It is assumed that each processor can read and write its own register and also read the register of its left-hand neighbor.

Configurations evolve according to the following rules: processor 0 can take a step provided its register value equals that of its left neighbor, processor N, and then increments its register by 1 modulo $M + 1$. A step of a non-zero processor copies the value held by its neighbor, provided that value is different from the value of its own register. Written in the form of guarded commands, the behavior of the system can thus be represented as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{var}\ v : \textbf{array}\ [0 \mathrel{..} N]\ \textbf{of}\ [0 \mathrel{..} M]} \\
& v[0] = v[N] & \longrightarrow \quad v[0] := (v[0] + 1) \bmod (M + 1) \\
\square_{i<N} & v[i + 1] \neq v[i] & \longrightarrow \quad v[i + 1] := v[i]
\end{array}$$



(a) assumed initial configuration (b) configuration after a few steps

Figure 1: Ring of processors

We call a configuration *stable* iff it contains exactly one enabled processor, which is interpreted as possessing the token. It is fairly easy to see (cf. section 3) that any system configuration contains at least one enabled processor, and that the number of enabled processors never increases. Therefore, once the system enters a stable state, it will subsequently remain stable.

For example, consider the configuration shown on the left-hand side of figure 1, in which processor 0 holds a value distinct from the values of all other processors (although some duplicate values may occur among the non-zero processors). In particular, $v[0]$ and $v[N]$ are different, and therefore processor 0 is disabled. The other processors may copy the values held by their left-hand neighbors into their own registers. In particular, the value held by processor 0 will spread around the circle, but processor 0 remains disabled until its register value has arrived at processor N. When this happens, all the processors share the same value, and only processor 0 is enabled: the system has stabilized. Our goal, of course, is to prove that the system will eventually stabilize from *any* initial configuration, and this requires some additional considerations. Nevertheless, the above argument gives an essential insight into the correctness of the algorithm. In fact, the proof by Qadeer and Shankar shows explicitly that the system will enter a state where processor 0 holds a value different from all other processors.

2 Formal Representation of the Algorithm

This section gives an overview of the formal description of Dijkstra's algorithm in Isabelle/HOL. We show actual excerpts from Isabelle's theory file (modulo some pretty-printing of operators that need to be input using ASCII symbols) because this gives a good idea of the clarity and crispness that is possible with such a tool. At the beginning we introduce a few type abbreviations that make the subsequent definitions more readable:

```
types
  proc    =   nat
  vals    =   nat
  state   =   proc => vals
  run     =   nat => state
```

The types of processors and values are both declared as synonyms for the predefined type of natural numbers.[1] A state is simply an assignment of values to processors, and a run

[1] Isabelle has a static and decidable type system that resembles that of ordinary functional programming languages and does not provide subrange types. Valid indices of processors and register values are defined via the sets `Proc` and `Vals` defined below.

is an infinite sequence of states, represented as a state-valued function of natural numbers.

Next, we declare the parameters N and M and assert the associated assumptions.

```
consts
  N, M          :: nat
rules
  N_pos         "0 < N"
  M_atleast_N   "N ≤ M"
```

The following definitions refine the type abbreviations shown above and define the actions of processors and their enabledness.

```
constdefs
  Proc     :: proc set
           Proc ≡ {i . i < Suc N}
  Vals     :: vals set
           Vals ≡ {i . i < Suc M}
  act      :: [proc, state, state] => bool
           act i v w ≡
             (case i of
                 0      => (v 0 = v N) ∧ (w 0 = (Suc (v 0)) mod (Suc M))
               | Suc j => (v j ≠ v (Suc j)) ∧ (w (Suc j) = v j))
            ∧ (∀ k. k ≠ i ⇒ (w k = v k))
  enab     :: [proc, state] => bool
           enab i v ≡ ∃ w. act i v w
  isState  :: state => bool
           isState v ≡ (v `` Proc) ⊆ Vals
  isRun    :: run => bool
           isRun r ≡ ∀ n. isState (r n)
                          ∧ (∃ i∈Proc. act i (r n) (r (Suc n)))
```

The definitions of `Proc` and `Vals` are straightforward. The predicate `act i v w` holds if state `w` results from an action of process `i` in state `v`. Its formalization is a rather obvious transcription of the conditional assignment used in the informal presentation, using case distinction on whether `i` is zero or a successor number. Predicate `enab i v` holds if process `i` can make a step in state `v`, that is, if there exists some state `w` that satisfies `act i v w`.

The predicates `isState` and `isRun` define when objects of type `state` and `run` represent "type correct" system configurations and runs. A valid configuration assigns values in `Vals` to every process[2]. A valid run is an infinite sequence of valid configurations related by actions of some process.

The remaining definitions are related to the verification of the algorithm. The informal discussion of section 1 suggests that the following entities play a role in the proof:

- the homogeneous prefix of the ring, that is, the initial segment of the ring where all processes share the same register value as process 0,
- the set *others* of register values that occur outside the prefix, and in particular the distance *minfree* (computed modulo $M + 1$) of $v[0]$ from some value that does not occur outside the prefix (when that distance is 0, process 0 has some value that does not occur outside the prefix and we have reached a situation as in figure 1), and
- a bit vector *enabs* $= \langle e_1, \ldots, e_{N-1} \rangle$ where e_i is 1 iff process i is currently enabled.

```
constdefs
  prefix   :: state => proc set
           prefix v ≡ { i. i∈Proc ∧ (∀ j. j ≤ i ⇒ v j = v 0) }
  others   :: state => vals set
           others v ≡ v `` (Proc \ (prefix v))
  minfree  :: state => vals
           minfree v ≡ LEAST n. n ∈ Vals
                                ∧ (v 0 + n) mod (Suc M) ∉ others v
```

[2] The set `f``A` denotes the image of set `A` under the mapping `f`.

The careful reader may worry whether the value *minfree* is actually well-defined and in particular what happens if the set in the definition of *minfree* becomes empty. HOL does not have undefined terms, but the value of the defining expression would in that case be unspecified. However, we will prove in corollary 8 in section 3 that this case does not occur.

In order to define the bit vector *enabs*, we introduce two auxiliary functions. The first one simply converts booleans to bits (this avoids defining an ordering on booleans); the second function computes prefixes of the bit vector of a given length and is defined via primitive recursion.[3]Finally, *meas* is defined as the ordered pair whose first component is *minfree* and whose second component is *enabs*.

```
consts
  bit       :: bool => nat
  enabsh    :: [state, proc] => nat list
  enabs     :: state => nat list
  meas      :: state => nat * (nat list)
primrec
  enabsh v 0       = []
  enabsh v (Suc k) = (enabsh v k) @ [bit (enab (Suc k) v)]
defs
  bit_def     bit b   ≡ if b then 1 else 0
  enabs_def   enabs v ≡ enabsh v N
  meas_def    meas v  ≡ (minfree v, enabs v)
```

Our last definition designates stable states; the quantifier ∃! represents unique existence:

```
constdefs
  stable    :: state => bool
            stable v ≡ ∃! i. i∈Proc ∧ enab i v
```

3 Verification of the Algorithm

We now present our proof of the correctness of Dijkstra's algorithm, via a series of lemmas, which correspond to the main steps in our machine verification. The actual proof script contains several initial lemmas about finite sets, modulo arithmetic, and downward-closed sets, which we omit in this presentation. For every lemma we give both the informal statement and the actual Isabelle representation, to give an impression of their close correspondence. However, the proofs are presented in ordinary mathematical style. Presently, Isabelle and similar interactive theorem provers do not provide a notation for proofs, and we feel this to be one of their main drawbacks.

Lemma 1 *At least one process is enabled, in any state.*

```
∃ i∈Proc. enab i v
```

Proof. Assume that all non-zero processes are disabled in state v. Then $v[i] = v[i+1]$ for $0 \leq i < N$, which implies $v[0] = v[N]$. Hence, process 0 is enabled. $\triangleleft$

Lemma 2 *A step of process i disables process i.*

```
act i v w ⟹ ¬ enab i w
```

Proof. If $i \neq 0$, the assertion is a trivial consequence of the definition of `act i v w`. For $i = 0$, the assertion follows from $(x+1) \bmod (M+1) \neq x$ (for any x). $\triangleleft$

[3]The standard library of Isabelle/HOL includes a theory of finite lists, where `[]` denotes the empty lists, `[a]` denotes the one-element list that contains `a`, and `@` represents list concatenation.

Lemma 3 *A step of process i affects at most the enabledness of process i and of its right-hand neighbor.*

```
[| i∈Proc; act i v w; j∈Proc; j≠i; j≠(i+1) mod (N+1) |]
⟹ enab j v = enab j w
```

Proof. The enabledness of process j at state v depends only on the register values of process j and its left-hand neighbor $(j-1) \bmod (N{+}1)$. Since steps of process i change only its own register, the assumptions that j is neither i nor i's right-hand neighbor ensure that $w[j] = v[j]$ and $w[(j-1) \bmod (N{+}1)] = v[(j-1) \bmod (N{+}1)]$. $\lhd$

Corollary 4 *If the system is stable at state v, then it is stable at any successor state of v.*

```
[| stable v; i∈Proc; act i v w |] ⟹ stable w
```

Proof. Since v is assumed to be stable and process i makes a step from v, i must be the unique process that is enabled at state v. Lemmas 2 and 3 imply that at most process i's right-hand neighbor is enabled at state w. On the other hand, there exists some enabled process at state w, by lemma 1, and therefore state w is stable. $\lhd$

Corollary 5 *Steps of non-zero processes lexicographically decrease enabs.*

```
[| Suc i∈Proc; act (Suc i) v w |]
⟹ (enabs w, enabs v) ∈ lexn less_than N
```

Proof. Since we assume process $i+1$ to make a step at state v, it is certainly enabled at v. Lemma 2 implies that process $i+1$ is disabled at w, hence the step of process $i+1$ causes bit e_{i+1} to decrease. On the other hand, it follows from lemma 3 that the enabledness of process $j+1$, for any $j < i$, does not change, i.e., the bits $e_1, \ldots, e_i$ are unaffected. $\lhd$

Lemma 6 *Steps of non-zero processes do not decrease the homogeneous prefix.*

```
act (Suc i) v w ⟹ prefix v ⊆ prefix w
```

Proof. If process $i+1$ makes a step at state v, it cannot be in *prefix*(v), because this would imply $v[i] = v[i+1] = v[0]$. Therefore, we have $k < i+1$, for any $k \in$ *prefix*(v). The definition of the step of process $i+1$ ensures $w[k] = v[k]$ for all $k < i+1$, which implies the assertion. $\lhd$

Lemma 7 *For any type correct state v, others$(v) \subset$ Vals.*

```
isState v ⟹ others v ⊂ Vals
```

Proof. The assumption that v is type correct immediately ensures that *others*$(v) \subseteq$ *Vals*. On the other hand, since $0 \in$ *prefix*(v) for any state v, the set *Proc* $\setminus$ *prefix*(v) has at most N elements, and so does *others*(v), which is the image of *Proc* $\setminus$ *prefix*(v) under v. By the pigeonhole principle, *others*(v) must be a proper subset of *Vals*, which has $M+1 > N$ elements. $\lhd$

Corollary 8 *For any type correct state, minfree is well-defined.*

```
isState v ⟹ (minfree v) ∈ Vals
             ∧ (v 0 + minfree v) mod (Suc M) ∉ others v
```

Proof. Lemma 7 ensures that there is some value $k \in$ *Vals* that does not occur in *others*(v). By elementary facts of modulo arithmetic, $k = (v[0]+n) \bmod (M{+}1)$, for some $n \in$ *Vals*. Hence the set in the definition of *minfree*(v) is non-empty and therefore contains a least number. $\lhd$

Lemma 9 *Steps of non-zero processes do not increase the set others.*

```
act (Suc i) v w ⟹ others w ⊆ others v
```

Proof. Assume $k \in \mathit{Proc} \setminus \mathit{prefix}(w)$. We have to show that $w[k] \in \mathit{others}(v)$. By lemma 6, it follows that $k \in \mathit{Proc} \setminus \mathit{prefix}(v)$ and thus $v[k] \in \mathit{others}(v)$. Moreover, if $k \neq i + 1$, the definition of `act` implies that $w[k] = v[k]$, which proves the assertion. If $k = i + 1$, then $w[k] = v[i]$. Again, if $i \in \mathit{Proc} \setminus \mathit{prefix}(v)$, we are done. Finally, if $i \in \mathit{prefix}(v)$, then $w[k] = v[i] = v[0]$, and $w[j] = v[j]$ for all $j < k$, which proves $k \in \mathit{prefix}(w)$, contradicting the assumption. ◁

Lemma 10 *If the register of process* 0 *doesn't change, minfree is monotonic in others.*

```
[| isState v; w 0 = v 0; others w ⊆ others v |] ⟹ minfree w ≤ minfree v
```

Proof. By corollary 8, we know $(v[0] + \mathit{minfree}(v)) \bmod (M{+}1) \notin \mathit{others}(v)$, and the assumptions $w[0] = v[0]$ and $\mathit{others}(w) \subseteq \mathit{others}(v)$ imply $(w[0] + \mathit{minfree}(v)) \bmod (M{+}1) \notin \mathit{others}(w)$, that is, $\mathit{minfree}(v)$ occurs in the set in the definition of $\mathit{minfree}(w)$. The assertion follows because $\mathit{minfree}(w)$ is the least element of that set. ◁

Corollary 11 *Steps of non-zero processes do not increase minfree.*

```
[| isState v; act (Suc i) v w |] ⟹ minfree w ≤ minfree v
```

Proof. Immediate consequence of lemmas 9 and 10, because steps of non-zero processes do not change the register of process 0. ◁

Corollaries 5 and 11 imply that the pair *meas* decreases lexicographically with each step of a non-zero process. It now remains to study the effect of steps of process 0.

Lemma 12 *Actions of process* 0 *may increase others at most by inserting* $v[0]$.

```
act 0 v w ⟹ others w ⊆ insert (v 0) (others v)
```

Proof. Assume $k \in \mathit{Proc} \setminus \mathit{prefix}(w)$. Then k must be non-zero and therefore we have $w[k] = v[k]$. If $v[k] = v[0]$, we are done. But if $v[k] \neq v[0]$, it cannot be the case that $k \in \mathit{prefix}(v)$, and therefore we have $v[k] \in \mathit{others}(v)$ as desired. ◁

Lemma 13 *If minfree is positive then steps of process* 0 *strictly decrease minfree.*

```
[| isState v; act 0 v w; minfree v = Suc m |] ⟹ minfree w ≤ m
```

Proof. The assumptions imply $w[0] = (v[0] + 1) \bmod (M{+}1)$ and $(v[0] + m + 1) \bmod (M{+}1) \notin \mathit{others}(v)$. Thus, we know $(w[0] + m) \bmod (M{+}1) \notin \mathit{others}(v)$. Using lemma 12, we may deduce $(w[0] + m) \bmod (M{+}1) \notin \mathit{others}(w)$ and therefore the assertion if we can prove that $(w[0] + m) \bmod (M{+}1) \neq v[0]$. By modulo arithmetic, this is equivalent to proving that $(v[0] + m + 1) \bmod (M{+}1) \neq v[0]$. Now, corollary 8 implies that $0 < m{+}1 < M{+}1$, and in particular $m{+}1 \not\equiv 0 \pmod{M{+}1}$. This suffices. ◁

Lemma 14 *If some non-zero process is enabled at state v, then* $N \notin \mathit{prefix}(v)$.

```
[| Suc i ∈ Proc; enab (Suc i) v |] ⟹ N ∉ prefix v
```

Proof. The assumptions imply $v[i] \neq v[i + 1]$ where $i + 1 \leq N$. Therefore, it is not the case that $v[j] = v[0]$ for all $j \leq N$. ◁

Lemma 15 *If* $\mathit{minfree}(v) = 0$ *and some non-zero process is enabled at state* v *then process* 0 *is disabled at state* v.

```
[| isState v; minfree v = 0; Suc i ∈ Proc; enab (Suc i) v |]
⟹ ¬ enab 0 v
```

Proof. The assumptions and lemma 14 imply that $v[N] \in \mathit{others}(v)$. On the other hand, corollary 8 ensures that $v[0] \notin \mathit{others}(v)$ because $\mathit{minfree}(v) = 0$. In particular, it follows that $v[0] \neq v[N]$, and therefore process 0 is disabled. ◁

Corollary 16 *Steps of process* 0 *decrease minfree, assuming that at least one non-zero process is enabled.*

```
[| isState v; act 0 v w; Suc i ∈ Proc; enab (Suc i) v |]
⟹ minfree w < minfree v
```

Proof. The assumptions and lemma 15 imply that *minfree*(v) is positive. Hence, the assertion follows by lemma 13. ◁

We now have all the bits and pieces to prove our main theorem that immediately implies eventual stabilization.

Theorem 17 *Every run reaches a state where no non-zero process is enabled.*

```
isRun rho ⟹ ∃ n. ∀ i. Suc i ∈ Proc ⇒ ¬ enab (Suc i) (rho n)
```

Proof. Assume, to the contrary, that ρ is a run such that at every state at least one non-zero process is enabled. Because successive states of ρ are connected by steps of some process, corollaries 5, 11, and 16 imply that the measure *meas*(v) lexicographically decreases with every step. But this is impossible because all relevant orderings are well-founded, so we reach a contradiction. ◁

Corollary 18 *Every run will eventually stabilize.*

```
isRun r ⟹ ∃ n. ∀ m. n ≤ m ⇒ stable (r m)
```

Proof. The assertion inductively follows from theorem 17 and lemma 4, since a state where no non-zero process is enabled is stable (by lemma 1). ◁

4 Discussion

Self-stabilization is a liveness property [1]. The proof presented in section 3 follows the standard technique to establish liveness properties: identify a valuation function that decreases with respect to some well-founded ordering unless the property is satisfied. With some practice, it is usually not too hard to come up with a suitable valuation function (in our case, the pair *meas*) by examining the operation of the algorithm. The only unusual aspect of this example is that we did not have to establish some auxiliary invariants before starting the liveness proof. In fact, Dijkstra's algorithm does not have any useful invariants, because by definition every possible ("type-correct") system state is reachable. (Observe that the system state is entirely given by the state of the processors' registers since there is no control state.)

In comparison, the proof in [9] is of more "local" nature in that it identifies more intermediate states than required for our proof and combines well-foundedness arguments with additional, temporal logic style, rules to deduce liveness properties.

Nevertheless, we feel that the presentation via a series of lemmas as in section 3 is unsatisfactory, in at least two respects:

- The structure of the overall argument is not evident and has to be re-established by a diligent reader. In particular, there is little motivation why a particular lemma is needed.
- From a more technical point of view, it is not obvious which parts of the proof can be left to an automatic prover and how.

We believe that both issues are conveniently addressed by a diagrammatic representation of the proof (or of the system, depending on one's point of view!) as shown in figure 2, which we now explain in some detail. The diagram shows a state transition system that consists of the states q_1, q_2, and q_3, where state q_1 is subdivided into states q_{11} and q_{12}.

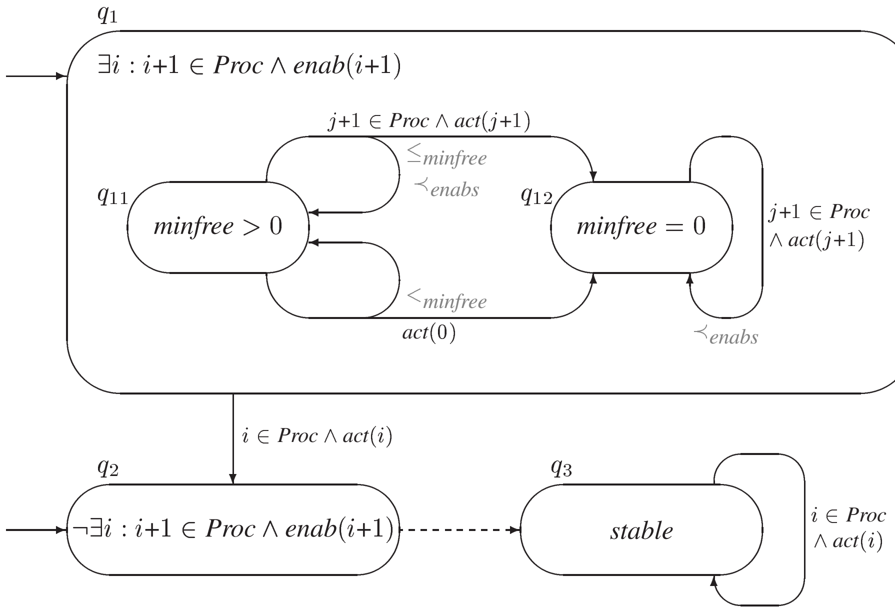


Figure 2: Proof diagram for Dijkstra's algorithm

Each state is labelled with a state predicate, defined in terms of the state functions introduced in section 2, where we have suppressed the state parameter. The label of a substate is the conjunction of its own label and the label of its superstate. For example, state q_{11} is labelled by the formula

$$\mathit{minfree} > 0 \land \exists i : i{+}1 \in \mathit{Proc} \land \mathit{enab}(i{+}1)$$

The states q_1 and q_2 are marked as initial; since none of the substates of q_1 is marked as initial, the transition system can initially be in either q_{11} or q_{12}. The dashed arrow between q_2 and q_3 indicates that the label of state q_2 implies that of q_3. The solid arrows between states indicate the possible state transitions; they are labelled with system actions that correspond to these transitions. For example, the system can move from state q_{11} to either q_{11} or q_{12} via an $\mathit{act}(0)$ transition. Similarly, it can move from q_1 or any of its substates to q_2 via a transition labelled $i \in \mathit{Proc} \land \mathit{act}(i)$. Some of the transitions are annotated (in grey) with their effects on the valuation functions *minfree* and *enabs*.

We assert that the state transition system of figure 2 is a proof diagram for the stabilization of Dijkstra's algorithm. This assertion entails two distinct proof obligations:

1. prove that every run of the algorithm corresponds to a run through the state transition system, and

2. prove that every run of the state transition system stabilizes.

The first proof obligation can be reduced to a number of non-temporal verification conditions about states and state transitions. For example, the initial condition of Dijkstra's algorithm should imply the disjunction of the labels of the states of the diagram marked as initial (trivial, since q_1 and q_2 are labelled by complementary state predicates). Also, the state transitions of the diagram should reflect all possible transitions of the original algorithm. The only non-trivial verification conditions in this category are

- a state where no non-zero processes are enabled is stable (lemma 1),

- any successor of a stable state is again stable (corollary 4),
- a step of a non-zero process from a state where *minfree* $= 0$ leads to a state where either *minfree* $= 0$ or no non-zero process is enabled (corollary 11), and
- a step of process 0 from a state where *minfree* $= 0$ and some non-zero process is enabled leads to a state where no non-zero process is enabled (lemma 15, by contradiction).

Finally, it needs to be shown that the annotations that indicate which valuation functions are (weakly or strictly) decreasing indeed hold (corollaries 5, 11, and 16). Any pre-established system invariants may be used in these proofs; in our case, this is the type-correctness predicate, which is explicitly asserted in the definition of a run. Conceptually, such invariants can be thought of as labelling a state that contains the entire diagram.

The second proof obligation corresponds to proving theorem 17 and corollary 18. In contrast to the non-temporal verification conditions, which in general require interactive prover support, this obligation can be discharged by model checking: it only remains to encode the proof diagram as a model and establish the property $\Diamond\Box q_3$ (in LTL notation). Because of the cycles in state q_1, this property only holds if we make use of the well-foundedness of the orderings that annotate some transitions. (The well-foundedness of these orderings is again easily established by the theorem prover.) In general, if f is a state function and R is a well-founded relation, then any run that contains infinitely many transitions known to decrease f with respect to R must also contain infinitely many transitions that may increase f. In our example, model checking has to be performed under the hypotheses

$$\Box\Diamond\text{``}<_{minfree}\text{''} \Rightarrow \Box\Diamond\neg(\text{``}<_{minfree}\text{''} \vee \text{``}\leq_{minfree}\text{''})$$

where "$<_{minfree}$" and "$\leq_{minfree}$" refer to the set of edges with the respective annotations, and similarly for $\prec_{enabs}$.

We strongly believe that verification diagrams are useful in system verification, both as a didactic interface for the human prover and because they separate the non-temporal aspects of the proof, which usually require support from interactive theorem provers, from the temporal aspects, which can and should be left to algorithmic provers such as model checkers. Verification diagrams are an appropriate device to present abstractions [6] of the system that imply the properties of interest. Similar styles of proof have been suggested, among others, by Lamport in [5] and by the STeP group in [2].

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás Uribe. Visual verification of reactive systems. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 334–350. Springer-Verlag, 1997.

[3] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[4] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[5] Leslie Lamport. TLA in Pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, September 1995.

[6] Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.

[7] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[8] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1994.

[9] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods*, pages 424–443, Shelter Island, N.Y., June 1998. Chapman & Hall.