

Verifying Self-stabilizing Population Protocols with Coq

Yuxin Deng* and Jean-François Monin†

*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

†Université de Grenoble 1, France

Abstract—Population protocols are an elegant model recently introduced for distributed algorithms running in large and unreliable networks of tiny mobile agents. Correctness proofs of such protocols involve subtle arguments on infinite sequences of events. We propose a general formalization of self-stabilizing population protocols with the Coq proof assistant. It is used in reasoning about a concrete protocol for leader election in complete graphs. The protocol is formally proved to be correct for networks of arbitrarily large size. To this end we develop an appropriate theory of infinite sequences, including results for reasoning on abstractions. In addition, we provide a constructive correctness proof for a leader election protocol in directed rings. An advantage of using a constructive setting is that we get more informative proofs on the scenarios that converge to the desired configurations.

Keywords-formal verification; population protocols; self-stabilization; proof assistant; Coq

I. INTRODUCTION

With the rapid development of mobile *ad hoc* networks, a great number of distributed algorithms have been proposed for solving various problems. However, their correctness is often informally proved. When an algorithm becomes complicated, an informal correctness proof is error-prone. Higher confidence can be gained if a proof can be rigorously verified.

In the area of formal verification one aims to establish system correctness with mathematical rigor. Industrial practice has shown that in developing complex hardware and software systems, more effort is spent on verification rather than on construction. Formal methods are playing more and more important roles in verifying applied systems. There are roughly two kinds of approaches in formal verification: *model checking* and *theorem proving*. Model checking explores the state space of a system model exhaustively to see if a desirable property is satisfied. Model checking is largely automatic and it generates a counterexample when the checked property fails to hold. However, model checkers usually face the state explosion problem when verifying large systems. On the other hand, the basic idea for theorem proving is to translate a system specification into a mathematical theory and then construct a proof of a theorem by generating the intermediate proof steps, or refute it. The strength of theorem proving is to deal with large or even infinite state space by using proof principles such as induction and co-induction. Interactive theorem provers are also called *proof assistants*.

Coq is a proof assistant in which high level proof search commands construct fully formal proofs behind the scene, which are then verified by a very reliable proof checker. Beyond formally verifying mathematical theorems, e.g. the famous four color theorem [9], Coq was successfully applied to ensure reliability of hardware and software systems in various fields [2]. Examples include multiplier circuits [14], concurrent communication protocols [8], devices for broadband protocols [12], and compilers [11], just to name a few.

In this paper, we report some preliminary results on verifying self-stabilizing population protocols [3] with Coq. The population protocol model has emerged as a new computation paradigm for describing mobile *ad hoc* networks that consist of a number of mobile nodes interacting with each other to carry out a computation. A central property of such protocols is that all nodes must eventually converge to the correct configurations, under certain fairness assumption.

In [13], Pang, Luo and Deng have used the Spin model checker [10] to formally verify the self-stabilizing population protocol for leader election in complete graphs [7] and the self-stabilizing population protocol for token circulation in directed rings [3]. However, the automatic part of the verification was limited to networks of size only up to six, due to the state explosion problem for larger networks.

To verify population protocols for large networks, we consider it more appropriate to use the approach of theorem proving rather than model checking. As the behavior of self-stabilizing protocols is based on infinite executions, we prove some meta-theorems with Coq in an abstract way about some typical operations on infinite sequences such as mapping operations, which map one sequence into another, and temporal logic operations like “eventually” and “always” operators in linear temporal logic. These meta-theorems are useful to reason about any type of infinite sequences, by instantiating the element of an abstract infinite sequence with appropriate types. We then present a general formalization of population protocols with Coq. It is used in analyzing a concrete population protocol for leader election in complete graphs that was proposed by Fischer and Jiang in [7]. Compared with the work of Fischer and Jiang, our proof is more formal and constructive. For example, [7, Lemma 1] was shown by contradiction, but we construct a proof for it by deduction. In [7] a protocol for leader election in directed rings was also proposed. However,

some key parts of its correctness proof are highly informal. We provide an alternative (also manual) correctness proof, which is more constructive than the original one and easier to be formalized. Realizing the formalization with Coq is tractable, but would be quite complex because of some subtleties that heavily involve a strong notion of fairness, so we leave it as future work.

Our contributions are threefold:

- 1) We develop some meta-theorems for manipulating infinite sequences. Some properties about mapping operations and temporal logic operations are of independent interest and would be useful for other Coq users.
- 2) We present a general formalization of population protocols with Coq.
- 3) We analyze a concrete population protocol for leader election in complete graphs. We prove its correctness for networks of arbitrarily large size. The proof is conducted in a formal and rigorous way. We also provide a constructive correctness proof for the leader election protocol in directed rings.

The paper is structured as follows. In Section II we briefly recall the definition of population protocols and a few fairness conditions. In Section III we give a brief introduction to Coq, develop some meta-theorems about infinite sequences, and then provide a general formalization of population protocols. In Section IV we formally prove a concrete population protocol for leader election in complete graphs. In Section V we consider a leader election protocol in rings and propose an alternative correct proof. Finally, we conclude the paper in Section VI.

II. THE POPULATION PROTOCOL MODEL

A population protocol [3] runs on an underlying network that can be described by a directed graph $G = (V, E)$ without multi-edges and self-loops. Each vertex represents a simple finite-state sensing device – the same on each vertex – and each edge (u, v) means that u as an *initiator* could possibly interact with v as a *responder*.

Devices are specified by a (unique) *protocol*, formalized by a tuple $P(Q, \mathcal{C}, X, Y, O, \delta)$ which contains

- a finite set Q of states,
- a set \mathcal{C} of configurations,
- a finite set X of input symbols,
- a finite set Y of output symbols,
- an output function $O : Q \rightarrow Y$, and
- a transition function $\delta : (Q \times X) \times (Q \times X) \rightarrow 2^{Q \times Q}$.

If $(p', q') \in \delta((p, x), (q, y))$, then we denote it by $((p, x), (q, y)) \rightarrow (p', q')$ and call it a transition. A *configuration* C is a mapping $C : V \rightarrow Q$ assigning to each node its internal state, and an *input assignment* $\alpha : V \rightarrow X$ specifies the input for each node. Let C and C' be configurations, α be an input assignment, and u, v be different nodes. If there is

a pair $(C'(u), C'(v)) \in \delta((C(u), \alpha(u)), (C(v), \alpha(v)))$, we say that C goes to C' via edge $e = (u, v)$ by transition $((C(u), \alpha(u)), (C(v), \alpha(v))) \rightarrow (C'(u), C'(v))$, abbreviated to $(C, \alpha) \xrightarrow{e} C'$ (the nodes different from u and v are not affected by edge e and thus keep their states unchanged). A pair of a transition r and an edge e constitutes an *action* $\sigma = (r, e)$. If C goes to C' via some edge, then C can go to C' in one *step*, written as $(C, \alpha) \rightarrow C'$.

An *execution* is an infinite sequence of configurations and assignments $(C_0, \alpha_0), (C_1, \alpha_1), \dots, (C_i, \alpha_i), \dots$ such that $C_0 \in \mathcal{C}$ and for each i it holds that $(C_i, \alpha_i) \rightarrow C_{i+1}$. A *behavior* can be represented by a trace of input or output assignments specifying desirable input or output of each node. P is an *implementation* of output behavior B_{out} under input behavior B_{in} , if any execution of P matches a trace in B_{out} provided that the input trace is in B_{in} . A *stable behavior* specifies a desirable suffix for each trace. Intuitively, it means that a desirable property eventually holds, whatever the initial conditions. If P is an implementation of stable behavior B^s from all possible configurations, then P is a *self-stabilizing implementation* of B^s .

In the area of formal verification, fairness is typically needed to prove liveness properties. It is concerned with a fair resolution of non-determinism, i.e., fairness conditions are used to rule out some *unrealistic* runs due to non-determinism. Let $E = (C_0, \alpha_0), (C_1, \alpha_1), \dots, (C_i, \alpha_i), \dots$ be an execution.

Definition 1 (Strong global fairness) *For every C, α , and C' such that $(C, \alpha) \rightarrow C'$, if $(C_i, \alpha_i) = (C, \alpha)$ for infinitely many i , then $(C_i, \alpha_i) = (C, \alpha)$ and $C_{i+1} = C'$ for infinitely many i . (Hence, the step $(C, \alpha) \rightarrow C'$ is taken infinitely many times in E .)*

Definition 2 (Strong local fairness) *For every action σ , if σ is enabled in (C_i, α_i) for infinitely many i , then $(C_i, \alpha_i) \xrightarrow{\sigma} C_{i+1}$ for infinitely many i . (Hence, the action σ is taken infinitely many times in E .)*

It should be noticed that global fairness is strictly stronger than local fairness [7]. The former requires that each *step* that can be taken infinitely often is actually taken infinitely often, while the latter asserts that each *action* which is enabled infinitely often is actually taken infinitely often. Since one action can be enabled in different configurations, the global fairness condition insists that an action should be taken infinitely often in all such configurations, whereas the local fairness condition only requires that it occurs infinitely often in one of such configurations.

In [7], two weak versions of fairness are also presented, and the relationship between these four kinds of fairness conditions is thoroughly discussed. They do not insist that particular steps occur infinitely often in E but only that the configurations that would result from those steps occur infinitely often. The relationship between these four kinds

of fairness conditions are discussed thoroughly in [7].

III. FORMAL VERIFICATION WITH COQ

In this section we briefly introduce Coq, develop some meta-theorems about infinite sequences, and then present a general formalization of population protocols.

A. A brief introduction to Coq

Coq is one of the most popular proof assistants for formal verifications. It is based on a constructive type theoretic setting, called the *Calculus of (co-)Inductive Constructions* (CIC), which can be summarized both as a polymorphic typed lambda-calculus enriched with universes, inductive and co-inductive types and a language for describing mathematical definitions and proofs [1], [4]. These two aspects are actually related thanks to the well-known Curry-Howard-De Bruijn isomorphism, which maps propositions to types and proofs to functional objects or strongly normalizing programs.

Let us illustrate some concepts by a few examples. One of the simplest inductive types is `nat`, with the usual constructors `O` and `S`; `nat` has itself the type `Set`, the realm of data structures:

```
Inductive nat : Set :=
  O : nat
  | S : nat -> nat.
```

Total recursive functional programs on arguments of type `nat` can be defined. For example, the binary sum of two natural numbers can be defined as follows.

```
Fixpoint plus (n m: nat) {struct n} : Set
:= match n with
  | 0 => m
  | S p => S (plus p m)
end.
```

In order to ensure termination, a structurally decreasing argument is specified by `struct n`. At the same level as `Set`, we have the realm `Prop` of propositions. For instance, a predicate over natural numbers has the type `nat -> Prop`. Given such a predicate `P`, a proof `p0` of `P 0` and a proof `step` of `forall n, P n -> P (S n)`, we can construct a proof of `P n` for all natural numbers `n`, using the following functional (primitive recursive) program:

```
Fixpoint natind (n:nat) {struct n}: P n
:= match n return (P n) with
  | 0 => p0
  | S q => step q (natind q)
end.
```

The type of `natind` is `forall n, P n`, that is a *dependent type*, since the type of the result depends on the value of the argument; `step`, seen as a function from numbers `n` and proofs of `P n` and returning a proof of `P (S n)`, has a slightly more complex dependent type. In the `match` construct itself, the type of the result depends

on the branch – it could be `P 0` or `P (S q)` for some `q`. Abstracting `P`, `p0` and `step` in `natind` yields a proof of the usual induction principle over natural numbers. As a function, it illustrates some important features of the type theory of Coq: polymorphism, inductive and dependent types. `Set` and `Prop` have themselves the type `Type`; there is actually a cumulative hierarchy of types called universes, but the technical details are not important here. We just need to know that `Type` is the right level to be used for general notions such as higher order data structures.

Altogether, the features of Coq allow us to formalize mathematical theories in a typed and precise but still very general setting. It should be noticed that this setting is basically *constructive*: existence proofs carry a computation of the witness. This can be relaxed if the principle of excluded middle (*PEM*) is used: *PEM* can safely be added as an axiom. A proof is always more informative when it is constructive, hence most Coq users prefer to forget *PEM* whenever possible.

Other constructs used in the sequel, such as *records*, are special cases of inductive types (i.e with only one constructor; *fields* are just projections). When defining inductive types, dependent types can also be used for constructors. It is especially convenient for formalizing algebraic structures (a carrier, operations and algebraic laws) and we use them extensively – see the simple illustrations below, e.g. `Protocol`.

Coq offers an environment where users can state mathematical definitions using types, concrete objects, functions over them, then interactively prove theorems. Obvious proof steps are automated, but clever ones, e.g. inductive arguments or intermediate subgoals, require user interactions.

Infinite objects such as streams or traces are dealt with using *co-inductive* types, which are defined like inductive types but it is not assumed that the number of constructors for making an inhabitant is finite. Fixpoint definitions over co-inductive arguments are not allowed. Co-inductive objects are constructed using `cofixpoints` definitions.

B. Infinite sequences and temporal logic

As we have seen in Section II, the behavior of self-stabilizing protocols is inherently based on infinite execution sequences. We have proved some meta-theorems in an abstract way about many typical operations on infinite sequences such as mapping operations which map one infinite sequence into another and temporal logic operations to express properties like certain event occurs infinitely often in a sequence. In principle, these meta-theorems are useful for manipulating any type of infinite sequences (by instantiating the elements of the abstract infinite sequence with appropriate types), not just for executions of population protocols. We formalized such a theory in Coq. Here are the main definitions and results to be used in the rest of the paper. In the following definitions, `T` is a `Type`, `s` is an

infinite sequence of elements of type T and P is a predicate on such sequences (except for `now`, where P is a predicate on T). Note that `eventually` is inductive while `always` co-inductive.

```
CoInductive infseq (T: Type) : Type :=
  Cons : T -> infseq T -> infseq T.
```

```
Definition now P s : Prop :=
  match s with Cons x s => P x end.
```

```
CoInductive always P : infseq T -> Prop :=
  | Always : forall s, P s ->
    always P (tl s) -> always P s.
```

```
Inductive eventually P : infseq T -> Prop :=
  | E0 : forall s, P s -> eventually P s
  | E_next : forall x s, eventually P s ->
    eventually P (Cons x s).
```

```
Definition inf_often P s : Prop :=
  always (eventually P) s.
```

A key for reasoning is to use abstract views, as given by a function f . For instance, an abstract view of a configuration will be the number of nodes satisfying a given property. Such a function is lifted on infinite sequences using the `map` operator:

```
CoFixpoint map
  (f: A->B) (s: infseq A): infseq B :=
  match s with
  Cons x s => Cons (f x) (map f s)
  end.
```

In order to reason using abstract views, we proved a number of lemmas saying that, for instance, if for all s , we have $P\ s$ entails $Q\ (\text{map } f\ s)$, then $\text{always } P\ s$ entails $\text{always } Q\ (\text{map } f\ s)$ and conversely. A similar lemma is available for the `eventually` operator but an additional technical condition – extensionality, which is satisfied in most practical cases – is required on P and Q .

C. A general formalization of population protocols

The behavior of a population protocol is usually described by a set of interaction rules (see e.g. Figure 1). On the left hand side of each rule, the state and the input of the initiator and the responder should be matched by the rules. On the right hand side, the rule specifies the state of the initiator and the responder after the transition has been taken.

We now present a general way of formalizing population protocols in Coq. See Section IV for the modeling of a concrete protocol. When verifying population protocols in Coq, we do not explicitly model output behavior as it can be ignored without loss of generality as far as correctness is concerned. Instead, we find it convenient to directly reason about executions, which involve no output symbols. So the type of protocols is defined by

```
Record Protocol : Type := mkProtocol {
```

```
  Q : Set ;           (* states *)
  X : Set ;           (* input symbols *)
  G : Graph ;        (* underlying graph *)
  rules : Set        (* transition rules *)
}.
```

where states, input symbols, and transition rules can be inductively defined. Underlying graphs can be specified in various ways. For example, we may use a finite set of natural numbers to represent the vertices of a finite graph, thus edges become relations on natural numbers.

```
Record Graph : Type := mkGraph {
  E : nat -> nat -> Prop      (* edges *)
}.
```

To facilitate our reasoning, we record the action that makes a configuration evolve into another one in a step of transition. Consequently, an execution is defined as an infinite sequence of *events*, where an event is a triple in the form (C, α, σ) composed of a configuration C , an input assignment α , and an action σ . Every two consecutive events in the sequence are related by a transition. The infinite sequence as a whole is defined co-inductively: if s is an infinite sequence of events, two events $(C_1, \alpha_1, \sigma_1)$ and $(C_2, \alpha_2, \sigma_2)$ are related by one step of transition, the concatenation $(C_2, \alpha_2, \sigma_2)s$ is an execution, then appending the event $(C_1, \alpha_1, \sigma_1)$ in the front of the execution gives rise to an enlarged execution $(C_1, \alpha_1, \sigma_1)(C_2, \alpha_2, \sigma_2)s$.

```
Definition events :=
  (configs * inputs * actions)%type.
```

```
CoInductive execution:infseq events -> Prop:=
  Cons_exec :
    forall (C1 C2 : configs)
      (alpha1 alpha2 : inputs)
      (sigma1 sigma2 : actions)
      (s : infseq events),
    transition C1 C2 alpha1 sigma1 ->
    execution (Cons (C2, alpha2, sigma2) s) ->
    execution (Cons (C1, alpha1, sigma1)
      (Cons (C2, alpha2, sigma2) s)).
```

An important ingredient in verifying population protocols is to model fairness conditions. An action σ is considered to be enabled in event x if the configuration component in x allows the action. The action is considered to occur in x if it coincides with the action component of x . Lifting these to infinite sequence, we can discuss if an action is enabled and occurs infinitely often. Thus, the definition of strong local fairness follows, where `inf_enabled` and `inf_occurred` involves the predicate `inf_often` defined in Section III-B.

```
Definition strong_local_fairness
  (s : infseq events) : Prop :=
  forall sigma : actions,
  inf_enabled sigma s ->
  inf_occurred sigma s.
```

A distributed system or a population protocol is said to be

self-stabilizing [6] if it satisfies the following two properties:

- *convergence*: starting from an arbitrary configuration, the system is guaranteed to reach a correct configuration;
- *closure*: once the system reaches a correct configuration, it cannot become incorrect any more.

Our verification follows this guidance and roughly proceeds in two stages. In the first stage, we prove that a correct configuration can always be reached eventually, regardless of the starting configurations. In the second stage, we prove that a network remains stable once it enters into a correct configuration. As usual, we must make appropriate assumptions about fairness conditions.

IV. SELF-STABILIZING LEADER ELECTION IN COMPLETE GRAPHS

In this section, we show that self-stabilizing leader election in complete graphs can be achieved under strong local fairness with the help of an eventually correct leader detector.

A. Algorithm

The algorithm was originally given in [7]. Every node has one bit memory which represents two states, being a leader (L) or not ($-$). The leader detector gives each node an input true (T) or false (F) to indicate that whether there is a leader in the network. The detector may give wrong answers sometimes, but it will eventually return a correct answer permanently. A non-leader becomes a leader, when the leader detector signals the absence of a leader, and the responder is not a leader. When two leaders interact, the responder becomes a non-leader. Otherwise, no state change occurs. The algorithm is described by the three interaction rules in Figure 1. On the left hand side, the state and input of an initiator and a responder should be matched. The symbol “ $*$ ” denotes that the input can always be matched. On the right hand side, the state of the two nodes would be updated by the rule.

$$\begin{array}{l}
 \text{Rule 1. } ((L, *), (L, *)) \rightarrow ((L), (-)) \\
 \text{Rule 2. } ((-, F), (-, *)) \rightarrow ((L), (-)) \\
 \text{Rule 3. } ((-, T), (-, *)) \rightarrow ((-), (-))
 \end{array}$$

Each node outputs its own state.

Figure 1. Leader election in complete graphs

In [7], it has already been manually shown that the algorithm implements self-stabilizing leader election in complete graphs under both global fairness and local fairness, provided the existence of an eventual leader detector. However, an important Lemma [7, Lemma 1] was proved by contradiction, which is less informative than a direct and constructive proof (cf. the lemma `ev_eq_1` below). In [13] the Spin model checker [10] is employed to show that the

algorithm is valid under an even weaker notion of fairness which says that if an activity is continuously often enabled then it has to be executed infinitely often. However, the verification is done for complete graphs of size only up to six. For larger sizes, the model checker fails and the correctness of the algorithm is again manually shown.

B. Verification

We first specify a population protocol model in Coq. Let `Max_num` be a variable. We consider a protocol that consists of nodes labeled with natural numbers ranging from 0 to `Max_num`. In complete graphs every two different nodes are connected via an edge. So we define a graph `G1` in the following way.

```

Definition Complete_graph_edge (u v : nat)
  : Prop :=
  u <= Max_num /\ v <= Max_num /\ u <> v.

```

```

Definition G1 := mkGraph Complete_graph_edge.

```

We use two kinds of states: L means being a leader and N is the opposite. Two kinds of input symbols are possible: T means the presence of at least one leader in the network and F is the opposite. For the transition rules, we explicitly model two kinds of idle transitions (`Stay1` and `Stay2` below) that keep the state of each node unchanged, besides the three transition rules given in Figure 1. A protocol is then defined as follows.

```

Inductive states : Set := L | N.
Inductive input_symbols : Set := T | F.
Inductive tran_rules : Set :=
  Rule1 | Rule2 | Rule3 | Stay1 | Stay2.
Definition P1 := mkProtocol
  states input_symbols G1 tran_rules.

```

The five transition rules give rise to the following transition relation.

```

Definition transition (C C' : configs )
  (alpha : inputs ) (sigma : actions) : Prop :=
  match sigma with (rule, (u, v)) =>
  u <= Max_num /\ v <= Max_num /\
  match rule with
  | Rule1 => C u = L /\ C v = L
    /\ C' u = L /\ C' v = N
  | Rule2 => C u = N /\ C v = N
    /\ C' u = L /\ C' v = N
    /\ alpha u = F
  | Rule3 => C u = N /\ C v = N
    /\ C' u = N /\ C' v = N
    /\ alpha u = T
  | Stay1 => C u = L /\ C v = N
    /\ C' u = L /\ C' v = N
  | Stay2 => C u = N /\ C v = L
    /\ C' u = N /\ C' v = L
  end
  /\ forall w, w <> u -> w <> v ->
  C w = C' w
end.

```

We also need to ensure that the eventual leader detector is faithful: it signals T if there is at least one leader in the network, and N for the opposite. Faithfulness is reflected via input assignments.

```

Definition faithful (C : configs)
  (alpha : inputs) : Prop :=
  ((exists v : nat,
    v <= Max_num /\ C v = L) ->
  (forall v : nat,
    v <= Max_num -> alpha v = T))
/\
  ((forall v : nat,
    v <= Max_num -> C v = N) ->
  (forall v : nat,
    v <= Max_num -> alpha v = F)).

```

So far we have completed the specification of the protocol. To prove that the protocol is self-stabilizing, we calculate the number of leaders in the network and proceed in three steps.

- 1) We begin with two auxiliary lemmas. The first one states that if there is no leader in the network, then eventually some leaders will be created, provided that the leader detector is eventually faithful.

```

Lemma zero :
  forall s : infseq events,
  execution s ->
  eventually (now faithful_tr) s ->
  now (total_leaders (eq 0)) s ->
  eventually(now (total_leaders (le 1))) s.

```

The second one says that if there are at least one leader in the network, then the number will decrease and eventually reach one exactly, provided that the leader detector keeps being faithful and the assumption strong local fairness is employed. Note that our only use of PEM is there. It is limited to the handling of strong fairness.

```

Theorem execution_to_one :
  forall s, execution s ->
  strong_local_fairness s ->
  always (now faithful_tr) s ->
  now (total_leaders (le 1)) s ->
  eventually(now (total_leaders (eq 1))) s.

```

The above two properties imply that starting from arbitrary configurations, eventually the number of leaders reaches one, if the leader detector is always faithful.

```

Lemma ev_eq_1 :
  forall s, execution s ->
  strong_local_fairness s ->
  always (now faithful_tr) s ->
  eventually(now (total_leaders (eq 1))) s.

```

- 2) In this step we show that if the number of leaders is one, then it remains unchanged, provided that the leader detector is always faithful.

```

Lemma execution_keep_1_trans :
  forall s, execution s ->
  always (now faithful_tr) s ->
  eventually
    (now (total_leaders (eq 1))) s ->

```

```

  eventually
    (always(now (total_leaders (eq 1)))) s.

```

- 3) Our correctness theorem now follows from the above lemmas. Starting from arbitrary configurations, the number of leaders will eventually keep being one, provided that the leader detector will eventually keep being faithful and strong local fairness is assumed.

```

Theorem correctness :
  forall s : infseq events,
  execution s ->
  eventually
    (always (now faithful_tr)) s ->
  strong_local_fairness s ->
  eventually
    (always(now (total_leaders (eq 1)))) s.

```

As can be seen, the above reasoning is based on tracing the numbers of leaders in the network. Instead of reasoning about an infinite execution, which is an infinite sequence of events $\{e_i\}_{i=0}^{\infty}$, it is often more convenient to reason about an infinite sequence of numbers $\{n_i\}_{i=0}^{\infty}$, where n_i is the number of leaders determined by the configuration extracted from event e_i , for all $i \geq 0$. So the mapping operation introduced in Section III-B is extremely useful here. For example, to prove the theorem `execution_to_one`, we have used the following auxiliary lemma

```

Lemma execution_to_one_map :
  forall s, execution s ->
  strong_local_fairness s ->
  always (now faithful_tr) s ->
  now (le 1) (map abstract_nat s) ->
  eventually (now (eq 1)) (map abstract_nat s).

```

where `abstract_nat` is essentially the function of abstracting the number n_i from event e_i . Therefore, by applying the mapping operation `map abstract_nat s` yields the infinite sequence of numbers as an abstract view of the infinite execution s . The detailed Coq script can be found in [5].

In this way, we have proved the correctness of the leader election algorithm in a complete graph of size `Max_num`. Since `Max_num` is assumed to be an arbitrary natural number, the algorithm is in fact correct for complete graphs of size n , for any natural number n .

V. SELF-STABILIZING LEADER ELECTION IN RINGS

In this section, we show that self-stabilizing leader election in rings can be achieved under global fairness with the help of a leader detector that is eventually correct.

A. Algorithm

In [7] an algorithm for self-stabilizing leader election in rings was also proposed, which is more complicated than the one for complete graphs. In this algorithm, each node has three types of memory slots for tokens: a bullet slot (B), a leader mark slot (L), and a shield slot (S). $(-)$ represents an empty slot, and a full slot is denoted by its token. The

order of slots in each node is (bullet, leader, shield). The leader detector gives each node an input true (T) or false (F) to indicate that whether there is a leader in the network. The algorithm is described by the following rules.

- Rule 1.* $((* * *, F), (* * *, *)) \rightarrow ((B L S), (* * *))$
Rule 2. $((* - S, T), (* * *, *)) \rightarrow ((* - -), (- * S))$
Rule 3. $((* L S, T), (* * *, *)) \rightarrow ((B L -), (- * S))$
Rule 4. $((* L -, T), (- * *, *)) \rightarrow ((B L -), (- * *))$
Rule 5. $((* * -, T), (B * *, *)) \rightarrow ((B - -), (- * *))$

Each node outputs its own state.

Figure 2. Leader election in rings

When two nodes interact and the initiator's input is false (F), a leader and a shield are created. At the same time, a bullet is fired (rule 1). This is the only way for leaders and shields to be created. When the initiator's input is true (T), the following rules apply: Shields move forward around the ring (rules 2 and 3), and bullets move backward (rule 5). Bullets are absorbed by any shield they encounter (rules 2 and 3) but kill any leaders along the way (rule 5). If a bullet moves into a node already containing a bullet, the two bullets merge into one. Similarly, when two shields meet, they merge into one. A leader fires a bullet whenever it is the initiator of an interaction (rules 3 and 4).

B. A correctness proof

It has been shown in [7] that leader election in rings does not work under the local fairness condition, but it works under the global fairness condition. Unfortunately, the correctness proof provided in [7] is unsatisfactory. For example, the crucial [7, Lemma 10] is shown in a highly informal way. In fact, that lemma heavily involves the global fairness condition and its proof is far from being straightforward.

Our proof differs in several respects from the original one given in [7]. In particular, the latter consists essentially in characterizing the set of infinitely recurring configurations of any given fair execution: this set happens to boil down to configurations with exactly one permanent leader and one moving shield (and innocuous bullets). This characterization is quite elegant but involves many non-constructive arguments and moreover forgets some crucial details, as mentioned above. Besides completing the missing proof steps of [7], our proof provides a better intuition of the progress towards the desired configurations.

We use the notion of *protected leader* given in [7], which is a leader such that there is no bullet between him and some shield in the ring. Transitions preserve the existence of a protected leader, which can be shown by induction on the rules in Figure 2.

We first remark that the leader detector is eventually faithful, hence for any given execution, we consider a suffix

where only faithful transitions are fired. Then we distinguish three stages in the remaining execution.

- 1) There is eventually a (protected) leader by rule 1, which is always enabled if no leader is present. From this stage, there always exists a protected leader in the ring.
- 2) At the next stage, rule 1 is disallowed because of the faithfulness of the leader detector, then no creation of shields can occur. By rules 2 and 3, the number of shields (\sharp_s) cannot increase when it is greater than 1. Moreover, Lemma 1 below shows that \sharp_s cannot always keep unchanged either because of global fairness, hence it will eventually decrease and then reaches 1.
- 3) At this stage, we have exactly one shield. Moreover, Lemma 2 below shows that the number of leaders (\sharp_l) will eventually decrease when it is greater than 1. However, when \sharp_l reaches 1, it will keep being 1 forever, which can be shown by induction on the transition rules.

Lemma 1: In a fair execution of faithful transitions starting with $\sharp_s \geq 2$, eventually \sharp_s decreases by 1.

Proof: In every configuration C such that $\sharp_s \geq 2$, let i and j ($i < j$) be the two smallest locations containing a shield; let $d = j - i$ and $m = \min(d, n - d)$ where n is the size of the ring. Intuitively, d measures the distance from i to j , and $n - d$ measures the distance from j to i , while m is the smallest distance between i and j . After a faithful transition step, there are three possibilities:

- 1) The two shields at i and j are not affected by the transition, so in the new configuration C' the locations of the two shields are the same as in C , i.e. $i' = i$ and $j' = j$.
- 2) The shield at i moves forward, so in configuration C' we have $i' = i + 1$ and $j' = j$.
- 3) The shield at j moves forward, so in configuration C' we have $i' = i$ and $j' = j + 1$ (if $j < n - 1$), or $i' = 0$ and $j' = i$ (if $j = n - 1$).

The last case corresponds to the firing of rules 2 or 3 on the shield at location j , but when $j = n - 1$ the roles of i and j are reversed to ensure that $i < j$. In the first case we have $m' = m$, but in each of the last two cases we can have either $m' = m - 1$ or $m' = m + 1$. Suppose that we always have $\sharp_s \geq 2$. As the set of configurations is finite, there exists a configuration where rules 2 or 3 will be enabled on i infinitely often. Then global fairness ensures that the corresponding transitions will take place and m will eventually decrease by 1¹. So m will eventually reach 0, that is, the two shields will merge into one. ■

¹Note that m might temporarily increase if $d > \frac{n}{2}$, but will decrease whenever $d \leq \frac{n}{2}$. The requirement of global fairness is crucial here, as local fairness cannot ensure that m will eventually diminish.

Lemma 2: In a fair execution of faithful transitions starting with $\sharp_s = 1$, eventually \sharp_l decreases by 1.

Proof: Bullets are always eventually fired by fairness and rules 3 and 4. The reasoning is similar to Lemma 1, using the distance between a bullet and an unprotected leader. ■

We are confident that the formal verification of this algorithm can roughly follow the same idea used in Section IV-B, but the details would be much more complex though still tractable, because of some pitfalls like the one in Footnote 1 that involves global fairness.

VI. CONCLUDING REMARKS

We have proposed a general formalization of population protocols with Coq. It has been used in successfully verifying a concrete self-stabilizing protocol for leader election in complete graphs. To do this, we have developed some meta-theorems about abstract infinite sequences, which are of independent interest. To our knowledge, this is the first experiment of verifying population protocols by using the approach of theorem proving. In addition, we have proposed an alternative correctness proof for the leader election protocol in rings given in [7], which is more constructive than the original proof.

Constructive proofs provide interesting hints on possible implementations. For instance, if we are going to implement the leader election protocol for rings in a quantitative framework where shields and bullets move with certain speeds, then the proof of Lemma 1 suggests that shields should be given speeds as different as possible, instead of a uniform speed, in order to let the numbers of shields converge as soon as possible. Similarly, the proof of Lemma 2 suggests that letting bullets move faster than shields would accelerate the rate of convergence for the numbers of leaders. Even though we are still far from a quantitative analysis of the protocol, such hints emerge clearly.

We believe that verifying the protocol for rings with Coq is also feasible, by formalizing our correctness proof along similar lines in the verification of the protocol for complete graphs, but the details would be much more complicated though tractable. As a future case study it would be interesting to realize this verification, or to investigate if the techniques developed in this paper can be adapted to formally prove other population protocols so as to increase our confidence on their correctness.

ACKNOWLEDGMENT

We thank the anonymous referees for useful comments on an earlier version of the paper. Deng would like to acknowledge the support of the National Natural Science Foundation of China (Grant No. 60703033).

REFERENCES

- [1] *The Coq Proof Assistant Reference Manual*. Available at <http://coq.inria.fr/V8.1pl3/refman/index.html>.
- [2] *The Coq user contributions*. Available at <http://coq.inria.fr/contribs-eng.html>.
- [3] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2005.
- [4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [5] Y. Deng and J.-F. Monin. *Coq script for self-stabilizing population protocols*. <http://basics.sjtu.edu.cn/~yuxin/Coq/script.rar>.
- [6] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
- [7] M. J. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2006.
- [8] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.
- [9] G. Gonthier. A computer-checked proof of the four colour theorem. <http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf>.
- [10] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [11] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54. ACM, 2006.
- [12] J.-F. Monin. Proving a real time algorithm for ATM in Coq. In *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 1998.
- [13] J. Pang, Z. Luo, and Y. Deng. On automatic verification of self-stabilizing population protocols. In *Proceedings of the 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 185–192. IEEE Computer Society, 2008.
- [14] C. Paulin-Mohring. Circuits as streams in coq: Verification of a sequential multiplier. In *Proceedings of the International Workshop on Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 1996.