To appear in *ICTL-97*,
Kluwer, July 1997.

Deductive verification of parameterized fault-tolerant systems: A case study *

Nikolaj S. Bjørner, Uri Lerner and Zohar Manna
Computer Science Department, Stanford University
nikolaj|uri|manna@cs.stanford.edu

Abstract. We present a methodology and a formal toolset for verifying fault-tolerant systems, based upon the temporal verification system STeP. Our test case is the modeling and verification of a parameterized fault-tolerant leader-election algorithm recently proposed in [9].
Our methods settle the general N-process correctness for the algorithm, which had been previously verified only for $N = 3$. We formulate the notion of *Uniform Compassion* to model progress in faulty systems more faithfully, and combine it with the more standard notions of fairness. We also show how the correctness proofs generalize to different channel models by a reduction to a simple channel model.

1 Introduction

The analysis of distributed algorithms can be rather complex when a large and intertwined set of interactions is considered. If the algorithms have only a finite set of states this analysis can be carried out automatically using *model checking* tools [16]. This yields a practical way to debug parameterized systems by checking different instantiations of the parameters. However, model checking runs quickly out of steam as the instantiated parameters take larger values.

Recently, a new *leader-election* algorithm was proposed as a challenging verification problem in [9]. Leader-election algorithms are used in a distributed environment to select one station that would serve as a coordinator in performing tasks needed by other stations in the system. In its general form the algorithm involves N independent processes and N communication links, each capable of holding k messages, yielding $O((N+kN)^N)$ states. The state explosion problem limited the model checking verification in [9] to $N = 3$ stations with only $k = 1$ messages in a link.

Contrary to model checking methods, deductive methods can handle infinite-state and parameterized systems. The Stanford Temporal Prover, STeP [3, 2], provides a mechanized deductive framework for verifying linear-time temporal properties of reactive and concurrent systems. We argue that the use of transition systems for system modeling, linear-time temporal logic for requirements

* This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

specification and STeP for verification is general and effective. We claim that the metric for the usefulness of formal validation should be the sum of the effort invested in these three areas: modeling, specification and verification. The proof of the correctness of the algorithm was presented to us as an open problem. We found and verified the proof within one week using STeP.

Using our parameterized approach we generalize the algorithm, which was originally presented only for links with capacity 1. We investigate the general case of links holding up to k messages, and show how this case can be reduced to links with no buffering capacity. We even investigate links that can store an unbounded number of messages, and show how our proof generalizes to this case as well.

To model progress more faithfully we introduce the new notion of *Uniform Compassion*, which is a stronger assumption than the standard compassion. We claim that Uniform Compassion captures the behavior of a faulty system correctly, and simplifies the formalization and proofs.

Although the analysis of parameterized systems is in general undecidable [1], a growing number of publications report sound, and in some cases complete, analysis methods for some restricted classes of parameterized systems [8, 10, 7, 11]. The algorithm we analyze in this paper does not belong to the categories handled by these methods.

In Section 2 we review the computational model of transition systems and temporal specifications. We also introduce Uniform Compassion and define a new proof rule to account for it. In Section 3 we model the leader-election algorithm as a transition system. In Section 4 we show that our modeling and analysis can be reused for different versions of the algorithm where network stations are linked by channels of various capacities. The verification of the safety and liveness requirements is finally given in Sections 5 and 6.

2 Computational Model

2.1 Fair Transition systems

In the style of [15], a fair transition system $\mathcal{S}$ is of the form $\langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, where $\mathcal{V}$ is a finite set of system variables, Θ is the initial condition, and $\mathcal{T}$ is a finite set of transitions. The vocabulary $\mathcal{V}$ contains data variables, control variables and auxiliary variables. The set of *states* (interpretations) over $\mathcal{V}$ is denoted by Σ. A *φ-state* is a state s where assertion (first-order formula) φ holds. The initial condition Θ is an assertion over $\mathcal{V}$. A transition τ maps each state $s \in \Sigma$ into a (possibly empty) set of τ-successor states, $\tau(s) \subseteq \Sigma$. The mapping associated with τ is defined by an assertion $\rho_\tau(\overline{x}, \overline{x}')$, called the *transition relation*, which relates the values $\overline{x}$ of the variables in state s and the values $\overline{x}'$ in a successor state $s' \in \tau(s)$. A transition τ is *enabled* at state s if $\tau(s) \neq \emptyset$. The assertion $En(\tau)$ characterizes the states where τ is enabled. Transition τ is *taken* from s to s' if $s' \in \tau(s)$. The Hoare-triple $\{\varphi\}\tau\{\psi\}$ denotes the verification condition $\varphi(\overline{x}) \ \wedge$

$\rho_\tau(\overline{x}, \overline{x}') \rightarrow \psi(\overline{x}')$. With $\{\varphi\}\mathcal{T}\{\psi\}$ we associate the conjunction $\bigwedge_{\tau\in\mathcal{T}} \{\varphi\}\tau\{\psi\}$. An infinite sequence of states $\sigma = s_0, s_1, \ldots$ is a *run* over system $\mathcal{S}$ if

Initiality The initial state s_0 satisfies Θ.

Consecution For each state s_i either $s_{i+1} = s_i$ (stuttering) or there is a transition $\tau \in \mathcal{T}$ such that $s_{i+1} \in \tau(s_i)$.

A run σ is a *computation* if the following fairness constraints are satisfied:

Justice For any just transition $\tau \in \mathcal{J}$: Transition τ cannot be continuously enabled without being taken.

Compassion For any compassionate transition $\tau \in \mathcal{C}$: If τ is enabled infinitely often it is taken infinitely often.

We use standard linear-time temporal logic [13] to specify requirements of reactive systems. For example the formula $\Box\varphi$ is true if φ holds on all states, and $\Diamond\varphi$ is true if φ holds some time in the future. We use $\varphi \Rightarrow \psi$ as shorthand for $\Box(\varphi \rightarrow \psi)$. A temporal formula that is satisfied by all computations of $\mathcal{S}$ is $\mathcal{S}$*-valid.*

Deductive verification of $\mathcal{S}$-validity relies on a library of verification rules that reduce the verification of temporal formulas into first-order premises. For example, the *basic invariance* verification rule `B-INV` reduces the proof of $\Box p$ to the verification of the first-order premises $\Theta \rightarrow p$ and $\{p\}\mathcal{T}\{p\}$. A complete set of inference rules for standard fair transition systems is presented in [12].

2.2 Uniform Compassion

To model liveness in fault-tolerant systems we introduce the notion of *Uniform Compassion.* Uniform Compassion captures the assumption that random faults cannot happen selectively only in certain states. For example, we want to rule out the case where there is an infinite number of messages from stations i and j, messages from station i are always lost and messages from station j are not always lost.

When making a transition uniformly compassionate we assume that it is taken uniformly on all the states where it is enabled without special preference to some states. Next to the justice and compassion sets $\mathcal{J}$ and $\mathcal{C}$ in fair transition systems we add the set of uniformly compassionate transitions $\mathcal{UC}$ and restrict computations to satisfy the extra condition:

Uniform compassion For any uniformly compassionate transition $\tau \in \mathcal{UC}$ and assertion φ: If τ is enabled infinitely often at φ-states, then it is taken infinitely often at a φ-state.

The notion of Uniform Compassion is closely related to the *extreme-fairness* found in [17], which is used in a slightly different model. Under this model a transition may have a few branches. When the transition is taken, a branch is selected nondeterministically. Under extreme fairness either a transition is taken only finitely often from a φ-state or every branch of the transition is taken infinitely often from a φ-state.

Note that for finite domains Uniform Compassion can be simulated by splitting uniformly compassionate transitions into a set of transitions, each corresponding to one possible state of the entire state space. However, even in this case Uniform Compassion offers an exponentially more succinct (and a more natural) way to represent systems.

We will use the verification rule below for transitions in $\mathcal{UC}$.

RESP$-$UC (Response under Uniform Compassion)	
For $\tau_h \in \mathcal{UC}$, assertions p, q, ψ and φ	
$U1$	$p \rightarrow \varphi \vee \psi \vee q$
$U2$	$\{\varphi\}\mathcal{T}\{q \vee \varphi \vee \psi\}$
$U3$	$\psi \;\Rightarrow\; \diamond(\varphi \vee q)$
$U4$	$\{\varphi\}\tau_h\{q\}$
$U5$	$\varphi \rightarrow En(\tau_h)$
	$p \;\Rightarrow\; \diamond q$

Theorem 1. *Rule RESP-UC is sound.*

Proof: From $U1 - U3$ we get that $p \Rightarrow (\square\diamond\varphi \vee \diamond q)$. If $p \Rightarrow \diamond q$ we are done. Assume $p \Rightarrow \square\diamond\varphi$. From U5 and using $\tau_h \in \mathcal{U}_C$ we get that τ_h will be taken infinitely often from a φ-state. From $U4$ this implies $\square\diamond q$. ∎

2.3 STeP

The STeP system was used to formally check the proofs of the safety and liveness requirements of the leader-election algorithm. This helped reveal several misguided proof attempts and finally led to a formally verified proof.

As input STeP accepts a state-based encoding of a reactive system as a fair transition system and a temporal requirements specification. Fair transition systems are encoded using a UNITY-style [5] programming language. Each transition τ is labeled by its name, auxiliary parameters $[i]$, and a fairness constraint. The parameters describe a set of transitions, one for each different instance of the parameters. The body of a transition is a guarded command consisting of an enabling condition *guard* and a (simultaneous) state assignment $\overline{x} := e$. Variables not mentioned explicitly in the state assignment are unchanged. The transition relation associated with transition $\tau[i]$ is $guard \;\wedge\; \overline{x}' = e \;\wedge\; \overline{y}' = \overline{y}$, where $\overline{y}$ are the state variables not included in $\overline{x}$. An example of such an encoding is presented in Figure 2.

Verification rules, verification diagrams [14], and decision procedures for propositional temporal logic are used to reduce the verification of temporal formulas to first-order verification conditions. STeP then provides an interactive Gentzen-style environment for establishing first-order formulas. A tightly integrated suite of decision procedures combined with first-order reasoning [4] facilitates this part of the verification considerably. We will give system and specification descriptions in STeP syntax to show how the formalization is presented to our tool.

3 The fault-tolerant leader-election algorithm

3.1 The leader-election algorithm

We are given a ring-formed network of stations indexed $1..N$ that share a common resource R. The stations can send messages to each other via channels in a clockwise fashion to determine which station can access R. Figure 1 shows the network configuration for three stations with links L_1, L_2 and L_3 and the shared resource in the middle. Simple algorithms for this distributed leader-election problem have been proposed in [6], whereas [9] gives an algorithm that addresses the issues of faulty channels and dead stations.

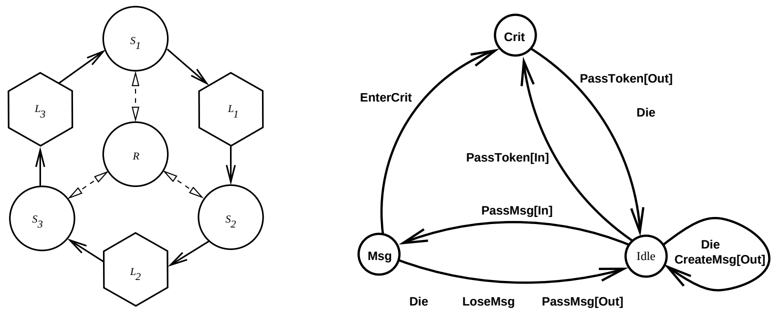

Fig. 1. Ring network and leader-election algorithm transitions

In the fault-tolerant leader-election algorithm each station can reside at three control locations: an idling location `Idle`, a message-passing location `Msg`, and a critical location `Crit`, where the station has exclusive access to the resource. We further assume that a station may die.

A station enters its critical section when it receives the token message from its predecessor. When it exits the critical section it sends the token message to its successor. We assume that the token message can get lost when it is sent (unreliable links). To overcome this we introduce a leader-election protocol. Every living station may send a claim message, meaning it assumes the token is lost and wants to create a new one. The claim message includes the station index and an extra bit, whose role will be discussed later.

When station i gets a claim message, it passes the message iff i is bigger than the index of station that created the claim. A dead station passes the messages it receives unless it created the message. This behaviour guarantees that only the lowest living station will get its own message back, so it cannot happen that two different stations will become the leader. Claim messages can get lost, just like token messages.

When a station gets its own message back it still cannot necessarily enter the critical section, because there may be a token in the network. As we prove in Section 5, in this case the token would have been received by the station exactly once between the time a message was sent and the time a message was received.

Therefore it is enough to keep one extra bit which changes whenever a station gets the token. When a station creates a claim message it adds the bit to the message. When a station gets its own message back it creates a new token iff the bit in the message is equal to the local bit in the station. Dead stations always delete their own messages.

The somewhat peculiar requirement that dead stations should be able to detect their own messages seems somewhat ad-hoc, but is necessary to guarantee liveness. Suppose that dead stations did not delete their messages, then we could have the following scenario: Station 1 sends $N-1$ messages and then dies. This implies that there is only one station in `Idle` mode throughout the network. Since no station can delete any one of the $N-1$ messages no two consecutive stations will ever be together in `Idle` mode, so a new claim message will never be created and liveness will be violated.

It is possible to change the algorithm by adding an extra field to the messages. The extra field is altered by live stations and is used to make sure that the same claim cannot go through the same live station twice. [2]

3.2 The formal model

In the original presentation of the algorithm, each control location has an analogous location in the case the station had died. We model these states by adding an extra bit `alive` to each station, indicating whether it is alive or dead.

In addition to `alive` each station saves another three variables. The integer `J` and the flag `b0` are the station index and extra bit fields in the message the station has received. The flag `b` is the local bit of the station. We represent these four local variables as four arrays of size N, where station i may access only entry i in the arrays.

The transition diagram on the right of Figure 1 summarizes the transitions that a station can take between its locations. The algorithm is encoded as a uniformly fair transition system in Figure 2. The encoding assumes that the links have zero capacity. The communication between the stations therefore proceeds by a synchronous handshake. Section 4 demonstrates that this choice is not a limitation.

The `PassToken` and `PassMsg` transitions send a token (claim message) to the next station that receives it (that is, the synchronous version combines `PassToken[Out]` and `PassToken[In]` into a single transition). `LoseToken` and `LoseMsg` are similar, but the token (claim message) gets lost due to the unreliable channels. `LoseMsg` is also used when a station deletes its own message (because it has died or the bit is in the wrong state). `CreateMsg` is used to create a claim message. `EnterCrit` is used to create a new token when a station gets its own message back. Finally, `Die` simulates the death of a station by moving to `Idle` and setting `alive` to *false*.

[2] Ironically, liveness can be guaranteed without a dead station deleting its own message if we let the `LoseMsg` transition be compassionate. This represents a network that is less reliable than our model, since it guarantees that if we pass an infinite number of messages an infinite number of messages will be lost.

```
type loc = { lCrit, lIdle, lMsg }
in    N : int where N > 1
local l : array [1..N] of loc where Forall i : [1..N] . l[i] = lIdle
local J : array [1..N] of [1..N]
local b,b0 : array [1..N] of bool where Forall i : [1..N] . b[i]
local alive : array [1..N] of bool where Forall i : [1..N] . alive[i]

macro Crit(i:[1..N]) = l[i] = lCrit
macro Idle(i:[1..N]) = l[i] = lIdle
macro Msg(i:[1..N]) = l[i] = lMsg
macro next(i:[1..N]) = if i = N then 1 else i + 1 (* i mod N + 1 *)

Transition LoseToken [i : [1..N]] :
  enable Crit(i) /\ Idle(next(i))
  assign (l[i],b[i]) := (lIdle,!b[i])

Transition PassToken [i : [1..N]] Compassionate :
  enable Crit(i) /\ Idle(next(i))
  assign (l[i],b[i],l[next(i)]) := (lIdle,!b[i],lCrit)

Transition CreateMsg [i : [1..N]] Compassionate :
  enable Idle(i) /\ Idle(next(i)) /\ alive[i]
  assign (l[next(i)],b0[next(i)],J[next(i)]) := (lMsg,b[i],i)

Transition EnterCrit [i : [1..N]] Just :
  enable Msg(i) /\ b0[i] = b[i] /\ J[i] = i /\ alive[i]
  assign l[i] := lCrit

Transition LoseMsg [i : [1..N]] Just :
  enable Msg(i) /\ !(b0[i] = b[i] /\ J[i] = i /\ alive[i]) /\
        (alive[i] /\ i < J[i] \/ i = J[i] \/ Idle(next(i)))
  assign l[i] := lIdle

Transition PassMsg [i : [1..N]] Uniformly Compassionate :
  enable Msg(i) /\ (alive[i] --> i > J[i]) /\ Idle(next(i)) /\ i != J[i]
  assign (l[i],l[next(i)],b0[next(i)],J[next(i)]) := (lIdle,lMsg,b0[i],J[i])

Transition Die [i : [1..N]] :
  enable alive[i]
  assign (alive[i],l[i]) := (false,lIdle)
```

Fig. 2. The synchronous leader-election algorithm encoded as a STeP transition system

The key requirements for the algorithm are:

Safety (Mutual Exclusion): At most one station will be at the critical section at any given time:

$$\texttt{Crit}(i) \wedge \texttt{Crit}(j) \Rightarrow i = j$$

Liveness: Every living station gets access to its critical section infinitely often:

$$\Box\Diamond(\neg\texttt{alive}[i] \vee \texttt{Crit}(i))$$

4 Modeling channels

The algorithm in Figure 2 is formulated for networks where message-passing proceeds synchronously. The links between two consecutive stations have capacity 0. We refer to the N-station instance of this algorithm as $\mathcal{S}[0, N]$. In general, let $\mathcal{S}[k, N]$ and $\mathcal{S}[\infty, N]$ be the versions where the links have buffering capacities k and ∞ respectively. The version model checked in [9] is $\mathcal{S}[1, 3]$.

In this section we demonstrate that our analysis of $\mathcal{S}[0, N]$ can be reused to verify correctness for the other versions of the algorithm, thereby producing correctness proofs "for free".

4.1 Simulating $\mathcal{S}[k, N]$ using $\mathcal{S}[0, N(k+1)]$

We now establish that we can simulate $\mathcal{S}[k, N]$ using $\mathcal{S}[0, N(k+1)]$ preserving certain properties. The idea is to simulate an asynchronous link between two stations by adding k dead stations between them. Dead stations behave as links, by passing on messages from other stations.

Not every execution sequence of $\mathcal{S}[0, N(k+1)]$ corresponds to an execution of $\mathcal{S}[k, N]$ with the dead-station-as-link interpretation. For example, consider the case where four dead stations simulate a four-message capacity link, and there are two messages in the link. When a capacity four buffer holds two messages, the actions of placing a new message on the buffer and taking a message from the buffer should both be enabled. However, in the configuration presented in Figure 3 a new message cannot be sent or received.

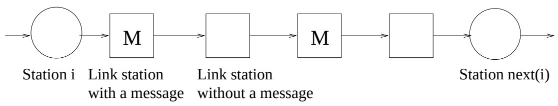

Fig. 3. Illegal link behavior

Fortunately, all that is needed for our purposes is a mapping from any $\mathcal{S}[k, N]$-computation σ to some $\mathcal{S}[0, N(k+1)]$-computation σ' that preserves the temporal properties of interest. The way a single state in $\mathcal{S}[k, N]$ is associated with a state in $\mathcal{S}[0, N(k+1)]$ is illustrated in Figure 4.

The mapping makes sure that for every state s in the run of $\mathcal{S}[k, N]$ there is a matching state $m(s)$ in the run in $\mathcal{S}[0, N(k+1)]$. We call the $m(s)$ states *genuine states*. The rest of the states in $\mathcal{S}[0, N(k+1)]$ are the *artificial states*.

In a genuine state, station i in the asynchronous version corresponds closely to station $f(i)$ in the synchronous version. The two stations are in the same state, their local variables have the same values, and a transition is enabled in one iff it is enabled in the other.

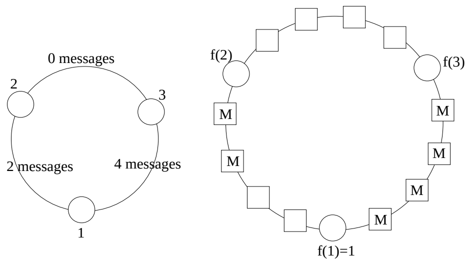

Fig. 4. Matching states in $\mathcal{S}[4, 3]$ and $\mathcal{S}[0, 15]$

A transition in the asynchronous system is simulated by a sequence of transitions in the synchronous system to maintain the state correspondence. The sequence of steps needed for this correspondence is illustrated in Figure 5. Initially, the synchronous version makes sure to kill the stations designated as links.

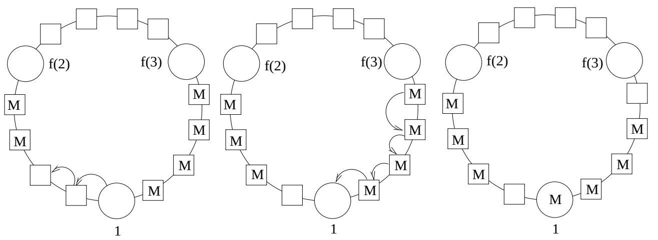

Fig. 5. Sending and receiving a message

4.2 Proving Mutual Exclusion for $\mathcal{S}[k, N]$

Theorem 2. *If Mutual Exclusion holds in $\mathcal{S}[0, N]$ for every N then it must hold in $\mathcal{S}[k, M]$ for every k, M.*

Proof: Assume Mutual Exclusion does not hold for $\mathcal{S}[k, M]$ for certain k, M. This implies the existence of a computation that leads to a state s where two different stations have a token. We simulate this computation with a $\mathcal{S}[0, N(k+1)]$ system. In state $m(s)$ in $\mathcal{S}[0, N]$ two different stations will have a token, therefore Mutual Exclusion is violated for the synchronous case. ∎

It should be obvious that we can modify this theorem to many other safety properties, assuming we can formalize them both in $\mathcal{S}[k, M]$ and in $\mathcal{S}[0, N]$.

4.3 Liveness

Establishing that the simulation using $\mathcal{S}[0, N(k+1)]$ preserves the Liveness property of $\mathcal{S}[k, N]$ is more complicated than Mutual Exclusion, since we have to take the fairness constraints of transitions into account. Uniform Compassion is especially troublesome as the set of assertions that hold for the different systems are incompatible. For example, we can compose a formula φ stating : "There is a link in illegal mode and a `CreateMsg` transition is enabled for station k". This formula may hold infinitely often, but the transition would not be taken according to the simulation definition, violating Uniform Compassion for the $\mathcal{S}[0, N(k+1)]$ system.

A key observation in overcoming this difficulty is that the deductive verification that will be presented in Section 6 only appeals to a restricted set $\mathcal{U} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ of assertions when using the verification rule for Uniform Compassion. Liveness not only holds for all computations, but also for all runs that satisfy Justice, Compassion and Uniform Compassion with respect to $\mathcal{U}$. Therefore, it is enough to show that if the Liveness requirements fail for $\mathcal{S}[k, N]$ then they fail for a run that satisfies the Uniform Compassion requirements with respect to $\mathcal{U}$.

Theorem 3. *If Liveness holds in $\mathcal{S}[0, N]$ for every N then it must hold in $\mathcal{S}[k, M]$ for every k, M.*

Proof: Define $\mathcal{U}$ to be the set of state formulas used in the Liveness proof. As we shall see in Section 6 we have:

$\mathcal{U} = \{\texttt{SmallestAlive}(k) \wedge \texttt{HasMsg}(k, j) \wedge \texttt{Idle}(\texttt{next}(j)) \mid 1 \leq k, j \leq N\}$

where `SmallestAlive`(k) is *true* iff station k is alive and no lower ranking station is alive.

Assume there is a computation of $A = \mathcal{S}[k, M]$ where Liveness does not hold. We simulate this computation by a run on $B = \mathcal{S}[0, M(k+1)]$ (where Liveness is violated). It is enough to show that this run satisfies Justice, Compassion and Uniform Compassion with respect to $\mathcal{U}$ (it does not necessarily satisfy Uniform Compassion with respect to every state formula). To establish this we consider the fair transitions in our system.

- Just and Compassionate Transitions
  We distinguish between links and original stations. It is fairly easy to show that in our simulation fairness is not violated by link stations. We concentrate on transitions for original stations. If such a transition is enabled continuously (infinitely often) in B it is also enabled continuously (infinitely often) in the genuine states of B. Therefore, if Justice (Compassion) is not violated in A it is not violated in B.
- Uniformly Compassionate Transitions
  The only relevant transition is `PassMsg`. This transition is always taken for the dead station representing a link, so again we restrict ourselves to original stations.

If $\varphi \in \mathcal{U}$ holds infinitely often, then it holds infinitely often in the genuine states, so we can restrict ourselves to these states (note that this is not true in general for every assertion φ). If $\varphi \in \mathcal{U}$ holds in $m(s) \in B$ then a similar formula must hold in $s \in A$ where we replace the predicate $\texttt{Idle}(\texttt{next}(k))$ by a predicate $\neg\texttt{FullLink}(k)$. Therefore if Uniform Compassion is not violated in A it is not violated in B with respect to $\mathcal{U}$. ∎

4.4 Unbounded links

Our reduction considered only bounded asynchronous links, but we can also use it for unbounded links.

Corollary 4. *If Mutual Exclusion holds for $\mathcal{S}[0, N]$ for every N, then it must hold for $\mathcal{S}[\infty, M]$ for every M.*

Proof: Assume Mutual Exclusion does not hold in $\mathcal{S}[\infty, M]$. Then there must exist a computation σ with a finite prefix σ' in which mutual exclusion is violated. Let k be the (finite) number of messages sent in σ'. Then σ' must also be a legal prefix of a computation in $\mathcal{S}[k, M]$. From Theorem 2 we get that Mutual Exclusion must also be violated for some N in $\mathcal{S}[0, N]$. ∎

We cannot use a similar argument for the Liveness property, since the computation segment in which the property may be violated is not finite. Indeed, we can have an infinite sequence of states in $\mathcal{S}[\infty, N]$ in which Liveness is violated. The only transitions taken are sending claim messages from stations on the network. We cannot give a similar sequence of states in $\mathcal{S}[0, N]$.

It is possible, however, to reuse the Liveness proof for the unbounded case with some local modifications. We actually can drop Part-1 of the proof (lemmass **liveness-A**, **liveness-B** and **liveness-C**). Lemmas **liveness-D** and **liveness-F** will be modified by the use of ranking functions that represent the number of messages waiting in the link. See Section 6 for more details.

5 Safety proof

The verification of the safety requirement proceeds in two steps, by establishing two invariants. The invariants are stated in Figure 6.

First we establish the auxiliary invariant `safety-A`: if station m has a message from station k, where $k \geq m$, then all stations with rank lower than m must have died before. This property holds because if station t with a rank lower than m is alive, it will not pass a message from station k, therefore m cannot receive the message.

The main specification is stated as the assertion `Mux`, appearing as the first conjunct of invariant `safety-B`. `Mux` states that at no time there is more than one token in the system. However, this conjunct by itself is not inductive since it does not prevent arbitrary stations to create tokens. We use the second conjunct of `safety-B` to obtain an inductive assertion.

We focus on stations m and k, where m has a message from k, and on the location of the token (if one exists). We examine two basic cases. The first case is where a token resides between k and m (including m, not including k). Note that if the the message can get from m to k a new token may be created and mutual exclusion will be violated. The reason this cannot happen is that in this case we would have $b[k] \neq b0[m]$. Therefore, we state that if there is a token between k and m then $b[k] \neq b0[m]$.

The second case is where there is a token between m and k (including k, not including m). We also include in this case the possibility of a token being created between m and k (this is possible if a living station gets its own message back and creates a new token). In this case we must have $b[k] = b0[m]$, so when the token passes through k we go to the previous case. We can state this as follows: if we are not in the first case, and still $b[k] \neq b0[m]$, then there is no token between m and k and none can be created.

The formalization of the safety property is given in Figure 6. Due to space constraints we do not show the actual proof of the invariance in this paper.

```
macro Mux = Forall x,y : [1..N] . Crit(x) /\ Crit(y) --> x=y
macro NoToken = Forall z : [1..N] . !Crit(z)
macro Between(x:[1..N],i:[1..N],j:[1..N]) =
    if i < j then (i<x /\ x<=j) else (i<x \/ x<=j)
macro TokenBetween(k:[1..N],m:[1..N]) =
    Exists j : [1..N] . Between(j,k,m) /\ Crit(j)
macro HasMsg(k:[1..N],m:[1..N]) = Msg(m) /\ J[m]=k
macro NoPreviousToken(k:[1..N],m:[1..N]) =
    Forall r : [1..k], n : [1..N] .
          HasMsg(r,n) /\ alive[r] /\ b[r]=b0[n] --> Between(n,r,m)

PROPERTY safety-A :
  [] Forall m,k,t : [1..N] .
        HasMsg(k,m) /\ t < m /\ m <= k --> !alive[t]

PROPERTY safety-B :
  [] (Mux /\
        Forall k, m : [1..N] . alive[k] /\ HasMsg(k,m) -->
              if TokenBetween(k,m) then
                     b[k]!=b0[m]
              else
                     (b[k]!=b0[m] --> NoPreviousToken(k,m) /\ NoToken))
```

Fig. 6. Safety specification in STeP

6 Liveness proof

In this Section we present the proof of the liveness requirement. We present the proof in an hierarchical form going from general high-level lemmas to the concrete formalization. Due to space constraints we only present the detailed proof of selected lemmas of expository interest.

6.1 Proof overview

We now give an informal overview of the proof. We decompose the proof into smaller parts and give an intuitive explanation why each part is correct.

In the proof we concentrate on the lowest living station. If there is no such station, then all the stations are dead and no progress will be made. In general the proof has three major parts:

Part–1 Show that the lowest living station will be able to send a claim message. A key lemma in this part is that messages from a dead station will eventually disappear from the network (established using **liveness-A,B,C** which are presented below).

Part–2 Show that if there is no token in the network then the lowest living station will create one. We do this by showing that the lowest living station will get its own message back (**liveness-D,E**).

Part–3 Show that every station will eventually get the token, assuming that there is at least one living station (**liveness-F,G**).

6.2 The Main Lemmas

The lemmas used in this proof are as follows:

liveness-A Every station gets to `Idle`: $\Box\Diamond\texttt{Idle}(k)$.
To prove this property we first show that at least one station is in `Idle`. We then claim that if station i is in `Idle` then eventually station $\texttt{pred}(i)$ will get to `Idle`. By induction we get that every station will eventually get to `Idle`.

liveness-B Messages of dead stations disappear: $\neg\texttt{alive}[k] \Rightarrow \Diamond\neg\texttt{HasMsg}(j,k)$.
Assume station k is dead and cannot create new claim messages. We want to show that eventually there will not be any messages from k in the network. This is true since every message from k will be lost along the network or will eventually get to k and then be deleted by the dead station.

liveness-C Smallest alive can send a claim or gets the token:
$\texttt{SmallestAlive}(k) \;\Rightarrow\; \Diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k) \;\vee\; En(\texttt{CreateMsg}[k]))$.
This property is implied by the last two properties. Let k be the lowest living station. By **liveness-B** eventually all the messages from lower ranking stations will disappear. After that, by **liveness-A** its successor will eventually get to `Idle`. If k is in `Msg` then it is either a message from itself or a message from a higher ranking station. In any case k will either enter `Crit` or delete the message and get to `Idle`. Since $\texttt{next}(k)$ must stay in `Idle` we will have infinitely often that both k and $\texttt{next}(k)$ will be in `Idle`.

liveness-D Smallest alive gets its claim:
$\texttt{SmallestAlive}(k) \;\Rightarrow\; \Diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k) \;\vee\; En(\texttt{EnterCrit}[k]))$.
We show by induction that every station will eventually get the claim from the smallest alive. The base case was established in **liveness-C**. For the induction step we use Uniform Compassion, with the assertion φ stating

that a station has a message from the lowest alive station. Since this happens infinitely often, the message will be passed to the next station.

liveness-E Smallest alive creates a token:
$\texttt{SmallestAlive}(k) \;\Rightarrow\; \diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k))$.
This is implied directly by **liveness-D**, assuming there is no token in the network that can change the bit in the station.

liveness-F Smallest alive sends the token to other stations:
$\texttt{SmallestAlive}(k) \;\Rightarrow\; \diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(j))$.
This lemma is established by induction much like **liveness-D** (although here we use only standard Compassion and not Uniform Compassion). The base case is established by **liveness-E**, and the induction step is a straightforward application of the compassion rule.

liveness-G Every alive station gets the token: $\square\diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k))$.
This is implied directly by **liveness-F**. If there is a living station then there is a smallest alive station, so either all the living stations will die or they will get the token.

In the next Sections we illustrate how the formally checked proof of lemmas **liveness-A** and **liveness-D** is carried out with STeP. The other cases are verified in a similar way. We have formally checked all lemmas. In this process we found bugs in preliminary informal proofs, and used the formal tool to improve the proof-structure.

6.3 The STeP proof of liveness-A

To establish that every station reaches its `Idle` state infinitely often we use the auxiliary lemmas in Figure 7. Property `A.1`, proved using rule B-INV, establishes that at any time some station resides at `Idle`.

```
macro dist(k1,k2) = (k1-k2) mod N
macro pred(i) = if i = 1 then N else i - 1

PROPERTY A.1 : [] Exists k : [1..N] . Idle(k)
PROPERTY A.2 : Idle(k) ==> <> Idle(pred(k))
PROPERTY A.3 : Forall m : [0..N-1], k1, k2 : [1..N] .
    m=dist(k1,k2) --> (Idle(k1) ==> <>Idle(k2))
PROPERTY A : []<> Idle(k)
```

Fig. 7. liveness-A

The proof of `A.2` is done using a verification diagram, presented in Figure 8. The diagram contains the different cases for process A and process $B = \texttt{pred}(A)$. Nodes labeled by φ_3 and φ_2 treat the cases where B is in the `Msg` state. It can then either go directly to the `Idle` state or enter the critical section in node φ_1. From φ_1 the transition `PassToken` is enabled since A is in the `Idle` state. Thus, taking the transition will lead to the node labeled by φ_0. STeP checks this argument formally by assigning the corresponding first-order verification conditions to each node and departing transition. Verification diagrams are described in more detail in [14].

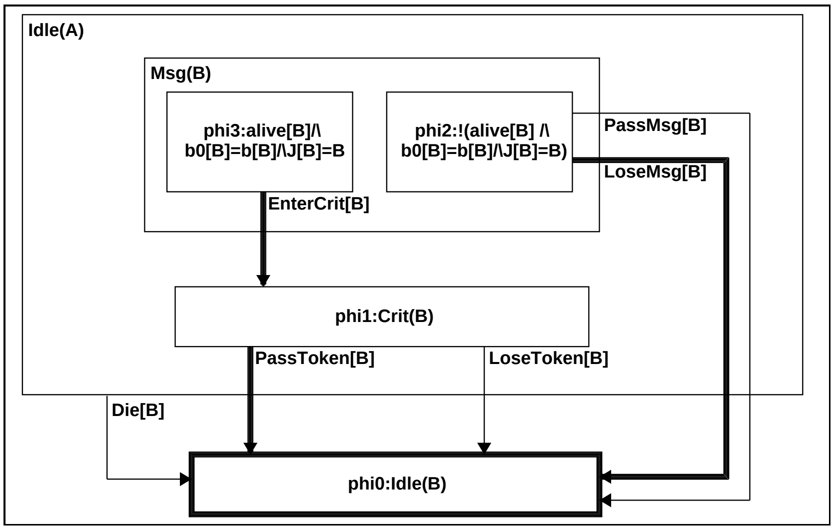

Fig. 8. Verification diagram for `A.2`

To prove `A.3` we apply the mathematical induction rule, where `A.2` is used for the induction step.

Finally, `A` follows from `A.1` and `A.3`. We use STeP's interactive Gentzen-style prover to split temporal operators, skolemize and instantiate `A.3` so the goal sequents can be decided using decision procedures. STeP combines propositional temporal decision procedure with ground decision procedures for linear arithmetic and other ground-decidable theories. This is helpful since instantiations of quantifiers give subgoals that are not purely propositionally valid. For instance $\Box\Diamond(k < 0 \wedge a \leq k) \rightarrow \Box\Diamond(a < 0)$ can be proved automatically, since all counter-models must contain the unsatisfiable state assignment $\langle k < 0 : \mathit{true},\ a \leq k : \mathit{true},\ a < 0 : \mathit{false}\rangle$.

6.4 The STeP proof of liveness-D

The first step in establishing that a message eventually circles the entire network, is the verification of property `D.1`. It states that the neighboring station to k, where `SmallestAlive(`k`)`, must obtain a claim from k. Property `D.1` follows as **liveness-C** ensures that the compassionate transition `CreateMsg`$[k]$ is enabled infinitely often if `SmallestAlive`(k) $\wedge\neg$ `Crit`(k) holds continuously.

Property `D.2` is proved by induction on `dist`(k, j). It establishes that j must obtain a claim from k infinitely often. The base of the induction is `D.1`. For the induction step, we assume

```
macro HasBMsg(k,j) = HasMsg(k,j) /\ b0[j] = b[k]

PROPERTY D.1 :
  SmallestAlive(k) ==>
      <>(!alive[k] \/ Crit(k) \/ SmallestAlive(k)/\ HasBMsg(k,next(k)))
PROPERTY D.2 :
  SmallestAlive(k) ==>
      <>(!alive[k] \/ Crit(k) \/ SmallestAlive(k)/\ HasBMsg(k,j))
PROPERTY D :
  SmallestAlive(k) ==> <>(!alive[k] \/ Crit(k) \/ Enabled(EnterCrit[k]))
```

Fig. 9. Proof of property `Liveness-D`

$$\texttt{SmallestAlive}(k) \;\Rightarrow\; \Diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k) \vee \texttt{SmallestAlive}(k) \wedge \texttt{HasBMsg}(k, j))$$

and show

$$\texttt{SmallestAlive}(k) \;\Rightarrow\; \Diamond(\neg\texttt{alive}[k] \vee \texttt{Crit}(k) \vee \texttt{SmallestAlive}(k) \wedge \texttt{HasBMsg}(k, \texttt{next}(j)))$$

using rule RESP-UC with intermediary assertions ψ : $\texttt{SmallestAlive}(k)$, φ : $\texttt{SmallestAlive}(k) \wedge \texttt{HasBMsg}(k, j) \wedge \texttt{Idle}(\texttt{next}(j))$, and the helpful transition τ_h : $\texttt{PassMsg}[k]$.

The first-order verification conditions generated by rule RESP-UC are verified using STeP's Gentzen prover using standard first-order reasoning aided by STeP's decision procedures. The verification rule also produces the temporal premise $U3$, which follows from the induction hypothesis, together with **liveness-A** and some simple safety properties that constrain j's movements from the `Msg` location.

7 Conclusion

We presented a formal proof of a generalized version of a fault-tolerant leader-election algorithm recently proposed in [9]. Safety was verified using standard verification rules for invariants. To analyze liveness under fault-tolerance, we developed a notion of Uniform Compassion to get a realistic model. Our approach allowed us to use existing tools already present in STeP.

This paper did not investigate fairness under fault-tolerancy from a model checking and proof-theoretical perspective in full depth. We feel there is still room to explore these issues.

Acknowledgements

We thank Xavier Boyen, Bernd Finkbeiner, Henny B. Sipma and Tomás E. Uribe for their most valuable comments and for insightful discussions.

References

1. Apt, K. R., and Kozen, D. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters 22*, 6 (1986).

2. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., and Uribe, T. STeP: The Stanford Temporal Prover, User's Manual. Tech. Rep. STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Nov. 1995.
3. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., and Uribe, T. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8th Intl. Conference on Computer Aided Verification* (July 1996), vol. 1102 of *LNCS*, Springer-Verlag, pp. 415–418.
4. Bjørner, N., Stickel, M., and Uribe, T. A practical combination of first-order reasoning and decision procedures. In *14th Intl. Conference on Automated Deduction* (July 1997), LNCS, Springer-Verlag. To appear.
5. Chandy, K., and Misra, J. *Parallel Program Design.* Addison-Wesley, 1988.
6. Chang, E., and Roberts, R. An improved algorithm for decentralized extrama-finding in circular configurations of processes. *Communications of the ACM 5*, 22 (May 1979), 281–283.
7. Clarke, E., Grumberg, O., and Jha, S. Verifying parameterized networks using abstraction and regular languages. In *CONCUR '95: Concurrency Theory. 6th Intl. Conference* (1995), I. Lee and S. Smolka, Eds., LNCS, Springer-Verlag.
8. Emerson, E., and Namjoshi, K. Automatic verification of parameterized synchronous systems. In *Proc. 8th Intl. Conference on Computer Aided Verification* (1996), R. Alur and T. Henzinger, Eds., vol. 1102 of *LNCS*, pp. 87–98.
9. Garavel, H., and Mounier, L. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. Rapport de recherche 2986, INRIA, Rhone-Alpes, France, Sept. 1996.
10. Ip, C., and Dill, D. Verifying systems with replicated components in Murφ. In *Proc. 8th Intl. Conference on Computer Aided Verification* (1996), R. Alur and T. Henzinger, Eds., vol. 1102 of *LNCS*, pp. 147–158.
11. Lesens, D., Halbwachs, N., and Raymond, P. Automatic verification of parameterized linear networks of processes. pp. 346–357.
12. Manna, Z., and Pnueli, A. Completing the temporal picture. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.* (1989), G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, Eds., vol. 372 of *LNCS*, Springer-Verlag, pp. 534–558. Also in Theoretical Computer Science.
13. Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.
14. Manna, Z., and Pnueli, A. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software* (1994), vol. 789 of *LNCS*, Springer-Verlag, pp. 726–765.
15. Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
16. McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.
17. Pnueli, A. On the extremely fair treatment of probabilistic algorithms. In *Proc. 15th ACM Symp. Theory of Comp.* (1983), pp. 278–290.