Safety Proof of a Distributed Termination-Detection Algorithm

Giuliano Losa

November 1, 2022

Contents

theory *Termination*
  imports *Main HOL−Statespace.StateSpaceSyntax*
begin

1 Specification of the algorithm

statespace $'p$ *vars* =
  $s :: 'p \Rightarrow 'p \Rightarrow nat$
  $S :: 'p \Rightarrow 'p \Rightarrow nat$
  $r :: 'p \Rightarrow 'p \Rightarrow nat$
  $R :: 'p \Rightarrow 'p \Rightarrow nat$
  $visited :: 'p \Rightarrow bool$
  $terminated :: bool$

context *vars*
begin

definition *pending* **where**
  — Number of messages in flight from p to q
  $pending\ c\ p\ q \equiv ((c \cdot s)\ p\ q) - ((c \cdot r)\ q\ p)$

definition *receive-step* **where**
  — Process p receives a message from q and sends a few more messages in response.
  $receive\text{-}step\ c\ c'\ p \equiv \exists\ q\ .$
    $p \neq q$
    $\wedge\ pending\ c\ q\ p > 0$
    $\wedge$ ($\exists\ P\ .$ — We pick a set P of processes to send messages to.
      $\exists\ s\text{-}p'\ .\ s\text{-}p' = (\lambda\ q\ .$
        $let\ s\text{-}p\text{-}q = ((c \cdot s)\ p\ q)$
        $in\ if\ q \in P - \{p\}\ then\ s\text{-}p\text{-}q + 1\ else\ s\text{-}p\text{-}q)$

— Note, above, that a process doesn't send a message to itself.
— Now we describe the new state:
$\wedge\ c' = c<s := (c{\cdot}s)(p{:=}s\text{-}p'),\ r := (c{\cdot}r)(p{:=}((c{\cdot}r)\ p)(q := (c{\cdot}r)\ p\ q + 1))>)$

definition *daemon-step* **where**
$daemon\text{-}step\ c\ c'\ p \equiv \neg\ c{\cdot}terminated \wedge ($
$\quad if\ (\exists\ p\ .\ \neg\ (c{\cdot}visited)\ p) \vee (\exists\ p\ q\ .\ (c{\cdot}S)\ p\ q \neq (c{\cdot}R)\ q\ p)$
$\quad then\ c' = c<$
$\qquad visited := (\lambda\ q\ .\ (c{\cdot}visited)\ q \vee p = q),$
$\qquad S := (c{\cdot}S)(p := (c{\cdot}s)\ p),$
$\qquad R := (c{\cdot}R)(p := (c{\cdot}r)\ p)>$
$\quad else\ c' = c<terminated := True>\ )$

definition *step* **where**
$step\ c\ c' \equiv \exists\ p\ .\ receive\text{-}step\ c\ c'\ p \vee daemon\text{-}step\ c\ c'\ p$

definition *init* **where**
$init\ c \equiv \forall\ p\ q\ .$
$\quad (c{\cdot}s)\ p\ q \geq 0$
$\quad \wedge\ (c{\cdot}S)\ p\ q = 0$
$\quad \wedge\ (c{\cdot}r)\ p\ q = 0$
$\quad \wedge\ (c{\cdot}R)\ p\ q = 0$
$\quad \wedge\ \neg\ (c{\cdot}visited)\ p$
$\quad \wedge\ \neg\ (c{\cdot}terminated)$

2 Correctness proof

definition *inv1* **where**
$inv1\ c \equiv \forall\ p\ q\ .\ (c{\cdot}R)\ p\ q \leq (c{\cdot}r)\ p\ q$

lemma *inv1-init*:
assumes *init c*
shows *inv1 c*
using *assms*
unfolding *init-def inv1-def*
by *auto*

lemma *inv1-step*:
assumes *step c c′* **and** *inv1 c*
shows *inv1 c′*
proof −
have *inv1 c′* **if** *receive-step c c′ p* **and** *inv1 c* **for** *p*
using *that* **unfolding** *receive-step-def inv1-def*
using *nat-le-linear not-less-eq-eq* **by** *fastforce*
moreover
have *inv1 c′* **if** *daemon-step c c′ p* **and** *inv1 c* **for** *p*
proof −
have $(c'{\cdot}R) = (c{\cdot}R)(p := (c{\cdot}r)\ p) \vee (c'{\cdot}R) = (c{\cdot}R)$
using ‹*daemon-step c c′ p*› **unfolding** *daemon-step-def* **by** (*auto split*: *if-splits*)

hence $(c' \cdot R) = (c \cdot R)(p := (c \cdot r)\ p) \vee (c' \cdot R) = (c \cdot R)$ **by** *blast*
moreover have $c' \cdot r = c \cdot r$ **using** ‹*daemon-step c c' p*› **unfolding** *daemon-step-def*
by (*auto split*:*if-splits*)
ultimately show *?thesis* **using** ‹*inv1 c*› **unfolding** *inv1-def* **by** *auto*
qed
ultimately show *?thesis*
using *assms step-def* **by** *blast*
qed

definition *inv2* **where**
— A process can only receive what has been sent.
$inv2\ c \equiv \forall\ p\ q\ .\ (c \cdot r)\ p\ q \leq (c \cdot s)\ q\ p$

lemma *inv2-init*:
assumes *init c*
shows *inv2 c*
using *assms*
unfolding *init-def inv2-def*
by *auto*

lemma *inv2-step*:
assumes *step c c'* **and** *inv2 c*
shows *inv2 c'*
proof −
have *inv2 c'* **if** *receive-step c c' p* **and** *inv2 c* **for** *p*
using *that* **unfolding** *receive-step-def pending-def inv2-def*
by (*auto*; *smt* (*verit*, *best*) *trans-le-add1*)
moreover
have *inv2 c'* **if** *daemon-step c c' p* **and** *inv2 c* **for** *p*
using *that* **unfolding** *daemon-step-def inv2-def*
by (*auto split*:*if-splits*)
ultimately show *?thesis*
using *assms step-def* **by** *blast*
qed

definition *consistent* **where**
$consistent\ c\ Q \equiv \forall\ p \in Q\ .\ (c \cdot visited)\ p \wedge (\forall\ q \in Q\ .\ (c \cdot S)\ p\ q = (c \cdot R)\ q\ p)$

definition *inv3* **where**
$inv3\ c \equiv \forall\ Q\ .\ consistent\ c\ Q \wedge (\exists\ p \in Q\ .\ \exists\ q\ .\ (c \cdot R)\ p\ q \neq (c \cdot r)\ p\ q \vee (c \cdot S)\ p\ q \neq (c \cdot s)\ p\ q)$
$\longrightarrow (\exists\ p \in Q\ .\ \exists\ q \in -Q\ .\ (c \cdot r)\ p\ q > (c \cdot R)\ p\ q)$

lemma *inv3-init*:
assumes *init c*
shows *inv3 c*
using *assms*
unfolding *init-def inv3-def consistent-def*

by *auto*

lemma *inv3-step*:
assumes *step* c c' **and** *inv1* c **and** *inv2* c' **and** *inv3* c
shows *inv3* c'
proof −
define *stale* **where** *stale* $c\ Q \equiv \exists\ p \in Q\ .\ \exists\ q\ .\ (c{\cdot}R)\ p\ q \neq (c{\cdot}r)\ p\ q \vee (c{\cdot}S)\ p\ q \neq (c{\cdot}s)\ p\ q$ **for** $c\ Q$
have *inv3* c' **if** *receive-step* $c\ c'\ p$ **for** p
proof −
{ **fix** Q
assume *consistent* $c'\ Q$ **and** *stale* $c'\ Q$
have $\exists\ p \in Q\ .\ \exists\ q \in -Q\ .\ (c'{\cdot}r)\ p\ q > (c'{\cdot}R)\ p\ q$
proof −
have *consistent* $c\ Q$ **using** ‹*consistent* $c'\ Q$› **and** ‹*receive-step* $c\ c'\ p$›
unfolding *consistent-def receive-step-def* **by** *auto*
{ **assume** *stale* $c\ Q$
— If Q is stale in c, then already in c there is a process that has received a message from outside Q that the daemon has not seen. This remains true.
hence $\exists\ p \in Q\ .\ \exists\ q \in -Q\ .\ (c{\cdot}r)\ p\ q > (c{\cdot}R)\ p\ q$ **using** ‹*inv3* c› ‹*consistent* $c\ Q$› *inv3-def stale-def* **by** *auto*
hence *?thesis* **using** ‹*receive-step* $c\ c'\ p$› **unfolding** *receive-step-def*
using ‹*inv1* c› *inv1-def less-Suc-eq-le* **by** *fastforce* }
moreover
{ **assume** $\neg$ (*stale* $c\ Q$)
— If Q is not stale in c, then no process in Q can receive a message from another process in Q (because all counts match). So, because we assume that the count of at least one process in Q changes, it must be by receiving a message from outside Q.
obtain q **where** $p \in Q$ **and** $(c'{\cdot}r)\ p\ q \neq (c{\cdot}r)\ p\ q$
using ‹*stale* $c'\ Q$› **and** ‹*receive-step* $c\ c'\ p$› **and** ‹$\neg$ (*stale* $c\ Q$)›
unfolding *receive-step-def stale-def pending-def*
apply *auto*
apply (*smt* (*verit*, *best*) *n-not-Suc-n*)+
done
moreover
have $q \notin Q$
proof −
have $\forall\ p \in Q\ .\ \forall\ q \in Q\ .\ (c{\cdot}r)\ p\ q = (c'{\cdot}r)\ p\ q$
proof −
from ‹$\neg$ (*stale* $c\ Q$)› **have** $\forall\ p \in Q\ .\ \forall\ q \in Q\ .\ (c{\cdot}s)\ p\ q = (c{\cdot}r)\ q\ p$
using ‹*consistent* $c\ Q$› *consistent-def stale-def* **by** *force*
thus *?thesis*
using ‹*receive-step* $c\ c'\ p$› *pending-def* **unfolding** *receive-step-def* **by** *auto*
qed
thus *?thesis*
using ‹$(c'{\cdot}r)\ p\ q \neq (c{\cdot}r)\ p\ q$› ‹$p \in Q$› **by** *auto*
qed

```
        moreover
        have (c′·r) p q > (c·r) p q using ‹(c′·r) p q ≠ (c·r) p q› and ‹receive-step
c c′ p›
          unfolding receive-step-def by (auto split:if-splits)
        moreover
         have (c′·R) p q = (c·r) p q using ‹receive-step c c′ p› and ‹¬ (stale c
Q)› and ‹p ∈ Q›
          unfolding receive-step-def stale-def by auto
        ultimately
        have ?thesis by force }
      ultimately show ?thesis by auto
    qed }
  thus ?thesis unfolding inv3-def stale-def by blast
qed
moreover
have inv3 c′ if daemon-step c c′ p for p
proof –
  { fix Q
    assume consistent c′ Q and stale c′ Q
    have ∃ p ∈ Q . ∃ q ∈ −Q . (c′·r) p q > (c′·R) p q
    proof (cases (∃ p . ¬ (c·visited) p) ∨ (∃ p q . (c·S) p q ≠ (c·R) q p))
      case True
      then show ?thesis
      proof –
        { assume p ∉ Q
          have ∃ p ∈ Q . ∃ q ∈ −Q . (c·r) p q > (c·R) p q
          proof –
            from ‹p∉Q› have consistent c Q and stale c Q
              using ‹daemon-step c c′ p› and ‹consistent c′ Q›
                and ‹stale c′ Q›
              unfolding daemon-step-def consistent-def stale-def
              by (force split:if-splits)+
            thus ?thesis using ‹inv3 c› unfolding inv3-def stale-def by auto
          qed
          hence ?thesis using ‹daemon-step c c′ p› and ‹p ∉ Q›
            unfolding daemon-step-def by (auto split:if-splits) }
        moreover
        { assume p ∈ Q
          define Q′ where Q′ ≡ Q − {p}
          obtain p′ q where p′ ∈ Q′ and q ∈ −Q′ and (c′·r) p′ q > (c′·R) p′ q
          proof –
            have ∃ p ∈ Q′ . ∃ q ∈ −Q′ . (c′·r) p q > (c′·R) p q
            proof –
              have ∃ p ∈ Q′ . ∃ q ∈ −Q′ . (c·r) p q > (c·R) p q
              proof –
                have consistent c Q′
                  using ‹daemon-step c c′ p› True ‹consistent c′ Q›
                  unfolding daemon-step-def consistent-def Q′-def
                  by (auto; (smt (verit)))
```

moreover
have *stale c Q′*
using ‹*daemon-step c c′ p*› *True* ‹*stale c′ Q*›
unfolding *daemon-step-def Q′-def stale-def*
by (*auto split*:*if-splits*)
ultimately
show *?thesis*
using ‹*inv3 c*› **unfolding** *inv3-def stale-def* **by** *auto*
qed
thus *?thesis* **using** ‹*daemon-step c c′ p*› **unfolding** *daemon-step-def*
Q′-def
by (*auto split*:*if-splits*)
qed
thus *?thesis* **using** *that* **by** *auto*
qed
moreover
have $q \neq p$
— Now we have to show that q is not the p visited by the daemon.
proof −
have $(c' \cdot R)\ p'\ p = (c' \cdot s)\ p\ p'$
proof −
have $(c' \cdot r)\ p = (c' \cdot R)\ p$ **and** $(c' \cdot s)\ p = (c' \cdot S)\ p$
using ‹*daemon-step c c′ p*› *True* **unfolding** *daemon-step-def*
by *auto*
moreover
have $(c' \cdot R)\ p'\ p = (c' \cdot S)\ p\ p'$ **using** ‹*consistent c′ Q*› ‹$p \in Q$› ‹$p' \in$
Q'›
unfolding *consistent-def Q′-def*
by *auto*
ultimately show *?thesis* **by** *auto*
qed
{ assume $p = q$
hence $(c' \cdot r)\ p'\ p > (c' \cdot R)\ p'\ p$ **using** ‹$(c' \cdot r)\ p'\ q > (c' \cdot R)\ p'\ q$›
by *auto*
hence $(c' \cdot r)\ p'\ p > (c' \cdot s)\ p\ p'$ **using** ‹$(c' \cdot R)\ p'\ p = (c' \cdot s)\ p\ p'$›
by *auto*
hence *False* **using** ‹*inv2 c′*› **unfolding** *inv2-def*
by (*simp add*: *leD*) **}**
thus *?thesis* **by** *blast*
qed
ultimately
have *?thesis* **using** *Q′-def* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed
next
case *False*
then show *?thesis*
using ‹*daemon-step c c′ p*› **and** ‹*consistent c′ Q*›

```
          and ‹stale c′ Q›
          and ‹inv3 c›
        unfolding daemon-step-def consistent-def inv3-def stale-def
        by auto
    qed }
  thus ?thesis
    using inv3-def stale-def by blast
 qed
 ultimately show ?thesis
  using assms(1) unfolding step-def by blast
qed

definition inv4 where
 inv4 c ≡ c·terminated ⟶ consistent c UNIV

lemma inv4-init:
 assumes init c
 shows inv4 c
 using assms
 unfolding init-def inv4-def
 by auto

lemma inv4-step:
 assumes step c c′ and inv4 c
 shows inv4 c′
 using assms
 unfolding step-def daemon-step-def receive-step-def inv4-def consistent-def
 by (force split:if-splits)

definition safety where
 safety c ≡ c·terminated ⟶ (∀ p q . pending c p q = 0)

lemma safe:
 assumes inv3 c and inv4 c
 shows safety c
 using assms unfolding inv3-def safety-def pending-def inv4-def consistent-def
 by (simp; metis ComplD iso-tuple-UNIV-I le-refl)

end

end
```