

Abortable Linearizable Modules

Rachid Guerraoui Viktor Kuncak Giuliano Losa

February 6, 2026

Abstract

We define the SLin I/O-automaton and prove its composition property. The SLin I/O-automaton is at the heart of the Speculative Linearizability framework. This framework simplifies devising robust distributed algorithms by enabling their decomposition into independent modules that can be analyzed and proved correct in isolation. It is particularly useful when working in a distributed environment, where the need to tolerate faults and asynchrony has made current monolithic protocols so intricate that it is no longer tractable to check their correctness. Our theory contains a formalization of simulation proof techniques in the I/O-automata of Lynch and Tuttle and a typical example of a refinement proof.

Contents

1	Introduction	2
2	Sequences as Lists	3
3	I/O Automata with Finite-Trace Semantics	4
3.1	Signatures	4
3.2	I/O Automata	4
3.3	Composition of Families of I/O Automata	6
3.4	Executions and Traces	7
3.5	Operations on Executions	8
4	Recoverable Data Types	10
4.1	The pre-RDR locale contains definitions later used in the RDR locale to state the properties of RDRs	10
4.2	Useful Lemmas in the pre-RDR locale	10
4.3	The RDR locale	11
4.4	Some useful lemmas	11
5	The SLin Automata specification	12

6	Definition and Soundness of Refinement Mappings, Forward Simulations and Backward Simulations	15
6.1	A series of lemmas that will be useful in the soundness proofs	16
6.2	Soundness of Refinement Mappings	16
6.3	Soundness of Forward Simulations	17
6.4	Soundness of Backward Simulations	17
7	Idempotence of the SLin I/O automaton	17
7.1	A case rule for decomposing the transition relation of the composition of two SLins	18
7.2	Definition of the Refinement Mapping	19
7.3	Invariants	19
7.4	Proof of the Idempotence Theorem	23
8	The Consensus Data Type	23
9	Conclusion	24

1 Introduction

Linearizability [2] is a key design methodology for reasoning about implementations of concurrent abstract data types in both shared memory and message passing systems. It presents the illusion that operations execute sequentially and fault-free, despite the asynchrony and faults that are often present in a concurrent system, especially a distributed one.

However, devising complete linearizable objects is very difficult, especially in the presence of process crashes and asynchrony, requiring complex algorithms (such as Paxos [3]) to work correctly under general circumstances, and often resulting in bad average-case behavior. Concurrent algorithm designers therefore resort to speculation, i.e. to optimizing existing algorithms to handle common scenarios more efficiently. More precisely, a speculative systems has a fall-back mode that works in all situations and several optimization modes, each of which is very efficient in a particular situation but might not work at all in some other situation. By observing its execution, a speculative system speculates about which particular situation it will be subject to and chooses the most efficient mode for that situation. If speculation reveals wrong, a new speculation is made in light of newly available observations. Unfortunately, building speculative system ad-hoc results in protocols so complex that it is no longer tractable to prove their correctness.

We the specification of the SLin (a shorthand for Speculative Linearizability) I/O-automaton [5], which can be used to build a speculatively linearizable algorithm out of independent modules that each implement the different modes of the speculative algorithm. The SLin I/O-automaton is

at the heart of the Speculative Linearizability framework [4, 1]. The Speculative Linearizability framework first appeared in [1] and was later refined in [4]. This development is based on the later [4].

The SLin I/O-automaton produces traces that are linearizable with respect to a generic type of object. Moreover, the composition of two instances of the SLin I/O-automaton behaves like a single instance. Hence it is guaranteed that the composition of any number of instances of the SLin I/O-automaton is linearizable. In this formal development, we prove the idempotence theorem, i.e. that the composition of two instances of the SLin I/O-automaton is itself an implementation of the SLin I/O-automaton.

The properties stated above simplify the development and analysis of speculative systems: Instead of having to reason about an entanglement of complex protocols, one can devise several modules with the property that, when taken in isolation, each module refines the SLin I/O-automaton. Hence complex protocols can be divided into smaller modules that can be analyzed independently of each other. In particular, it allows to optimize an existing protocol by creating separate optimization modules, prove each optimization correct in isolation, and obtain the correctness of the overall protocol from the correctness of the existing one.

In this document we define the SLin I/O-automaton and prove the Composition Theorem, which states that the composition of two instances of the SLin I/O-automaton behaves as a single instance of the SLin I/O-automaton. We use a refinement mapping to establish this fact.

2 Sequences as Lists

theory *Sequences*

imports *Main*

begin

locale *Sequences*

begin

We reverse the order of application of (#) and (@) because it we think that it is easier to think of sequences as growing to the right.

no-notation *Cons* (**infixr** <#> 65)

abbreviation *Append* (**infixl** <#> 65)

where *Append* $xs\ x \equiv Cons\ x\ xs$

no-notation *append* (**infixr** <@> 65)

abbreviation *Concat* (**infixl** <@> 65)

where *Concat* $xs\ ys \equiv append\ ys\ xs$

end

end

3 I/O Automata with Finite-Trace Semantics

```
theory IOA
imports Main Sequences
begin
```

This theory is inspired and draws material from the IOA theory of Nipkow and Müller

```
locale IOA = Sequences
```

```
record 'a signature =
  inputs::'a set
  outputs::'a set
  internals::'a set
```

```
context IOA
begin
```

3.1 Signatures

```
definition actions :: 'a signature  $\Rightarrow$  'a set where
  actions asig  $\equiv$  inputs asig  $\cup$  outputs asig  $\cup$  internals asig
```

```
definition externals :: 'a signature  $\Rightarrow$  'a set where
  externals asig  $\equiv$  inputs asig  $\cup$  outputs asig
```

```
definition locals :: 'a signature  $\Rightarrow$  'a set where
  locals asig  $\equiv$  internals asig  $\cup$  outputs asig
```

```
definition is-asig :: 'a signature  $\Rightarrow$  bool where
  is-asig triple  $\equiv$ 
    inputs triple  $\cap$  outputs triple = {}  $\wedge$ 
    outputs triple  $\cap$  internals triple = {}  $\wedge$ 
    inputs triple  $\cap$  internals triple = {}
```

```
lemma internal-inter-external:
```

```
  assumes is-asig sig
  shows internals sig  $\cap$  externals sig = {}
  <proof>
```

```
definition hide-asig where
  hide-asig asig actns  $\equiv$ 
    (inputs = inputs asig - actns, outputs = outputs asig - actns,
     internals = internals asig  $\cup$  actns)
```

```
end
```

3.2 I/O Automata

```
type-synonym
```

(s, a) transition = $s \times a \times s$

record (s, a) ioa =
 asig :: a signature
 start :: s set
 trans :: (s, a) transition set

context IOA
begin

abbreviation $act A \equiv actions (asig A)$
abbreviation $ext A \equiv externals (asig A)$
abbreviation int **where** $int A \equiv internals (asig A)$
abbreviation $inp A \equiv inputs (asig A)$
abbreviation $out A \equiv outputs (asig A)$
abbreviation $local A \equiv locals (asig A)$

definition $is-ioa :: (s, a) ioa \Rightarrow bool$ **where**
 $is-ioa A \equiv is-asig (asig A)$
 $\wedge (\forall triple \in trans A . (fst \ o \ snd) \ triple \in act A)$

definition $hide$ **where**
 $hide A \ actns \equiv A(\lambda asig := hide-asig (asig A) \ actns)$

definition $is-trans :: s \Rightarrow a \Rightarrow (s, a) ioa \Rightarrow s \Rightarrow bool$ **where**
 $is-trans \ s1 \ a \ A \ s2 \equiv (s1, a, s2) \in trans A$

notation
 $is-trans \ (\leftarrow \ \text{-----} \ \rightarrow) [81, 81, 81, 81] \ 100$

definition $rename-set$ **where**
 $rename-set A \ ren \equiv \{b. \exists x \in A . ren \ b = Some \ x\}$

definition $rename$ **where**
 $rename A \ ren \equiv$
 $(\lambda asig = (\lambda inputs = rename-set (inp A) \ ren,$
 $outputs = rename-set (out A) \ ren,$
 $internals = rename-set (int A) \ ren),$
 $start = start A,$
 $trans = \{tr. \exists x . ren (fst (snd tr)) = Some \ x \wedge (fst \ tr) \ -x-A \longrightarrow (snd (snd$
 $tr))\})$

Reachable states and invariants

inductive
 $reachable :: (s, a) ioa \Rightarrow s \Rightarrow bool$
for $A :: (s, a) ioa$
where
 $reachable-0: s \in start A \implies reachable A \ s$
 $| \ reachable-n: [\ reachable A \ s; s \ -a-A \longrightarrow t] \implies reachable A \ t$

definition *invariant* **where**

invariant $A P \equiv (\forall s . \text{reachable } A s \longrightarrow P(s))$

theorem *invariantI*:

fixes $A P$

assumes $\bigwedge s . s \in \text{start } A \implies P s$

and $\bigwedge s t a . \llbracket \text{reachable } A s ; P s ; s -a-A \longrightarrow t \rrbracket \implies P t$

shows *invariant* $A P$

<proof>

end

3.3 Composition of Families of I/O Automata

record *'id, 'a* *family* =

ids :: *'id* *set*

memb :: *'id* \Rightarrow *'a*

context *IOA*

begin

definition *is-ioa-fam* **where**

is-ioa-fam $fam \equiv \forall i \in \text{ids } fam . \text{is-ioa } (\text{memb } fam i)$

definition *compatible2* **where**

compatible2 $A B \equiv$

$\text{out } A \cap \text{out } B = \{\}$ \wedge

$\text{int } A \cap \text{act } B = \{\}$ \wedge

$\text{int } B \cap \text{act } A = \{\}$

definition *compatible::('id, ('s, 'a)ioa) family* \Rightarrow *bool* **where**

compatible $fam \equiv \text{finite } (\text{ids } fam) \wedge$

$(\forall i \in \text{ids } fam . \forall j \in \text{ids } fam . i \neq j \longrightarrow$

$\text{compatible2 } (\text{memb } fam i) (\text{memb } fam j))$

definition *asig-comp2* **where**

asig-comp2 $A B \equiv$

$(\text{inputs} = (\text{inputs } A \cup \text{inputs } B) - (\text{outputs } A \cup \text{outputs } B),$

$\text{outputs} = \text{outputs } A \cup \text{outputs } B,$

$\text{internals} = \text{internals } A \cup \text{internals } B)$

definition *asig-comp::('id, ('s, 'a)ioa) family* \Rightarrow *'a* *signature* **where**

asig-comp $fam \equiv$

$(\text{inputs} = \bigcup i \in (\text{ids } fam) . \text{inp } (\text{memb } fam i)$

$- (\bigcup i \in (\text{ids } fam) . \text{out } (\text{memb } fam i)),$

$\text{outputs} = \bigcup i \in (\text{ids } fam) . \text{out } (\text{memb } fam i),$

$\text{internals} = \bigcup i \in (\text{ids } fam) . \text{int } (\text{memb } fam i) \text{)}$

definition *par2* (infixr <||> 10) **where**

$A \parallel B \equiv$
 $(\text{asig} = \text{asig-comp2 } (\text{asig } A) (\text{asig } B),$
 $\text{start} = \{pr. \text{fst } pr \in \text{start } A \wedge \text{snd } pr \in \text{start } B\},$
 $\text{trans} = \{tr.$
 $\text{let } s = \text{fst } tr; a = \text{fst } (\text{snd } tr); t = \text{snd } (\text{snd } tr)$
 $\text{in } (a \in \text{act } A \vee a \in \text{act } B)$
 $\wedge (\text{if } a \in \text{act } A$
 $\text{then } \text{fst } s -a-A \longrightarrow \text{fst } t$
 $\text{else } \text{fst } s = \text{fst } t)$
 $\wedge (\text{if } a \in \text{act } B$
 $\text{then } \text{snd } s -a-B \longrightarrow \text{snd } t$
 $\text{else } \text{snd } s = \text{snd } t) \})$

definition *par*::('id, ('s,'a)ioa) family \Rightarrow ('id \Rightarrow 's,'a)ioa **where**

par fam \equiv let *ids* = *ids* fam; *memb* = *memb* fam in
 $(\text{asig} = \text{asig-comp } \text{fam},$
 $\text{start} = \{s. \forall i \in \text{ids}. s \ i \in \text{start } (\text{memb } i)\},$
 $\text{trans} = \{ (s, a, s').$
 $(\exists i \in \text{ids}. a \in \text{act } (\text{memb } i))$
 $\wedge (\forall i \in \text{ids}.$
 $\text{if } a \in \text{act } (\text{memb } i)$
 $\text{then } s \ i -a-(\text{memb } i) \longrightarrow s' \ i$
 $\text{else } s \ i = (s' \ i) \} \})$

lemmas *asig-simps* = *hide-asig-def is-asig-def locals-def externals-def actions-def*
hide-def compatible-def asig-comp-def

lemmas *ioa-simps* = *rename-def rename-set-def is-trans-def is-ioa-def par-def*

end

3.4 Executions and Traces

type-synonym

('s,'a)pairs = ('a \times 's) list

type-synonym

('s,'a)execution = 's \times ('s,'a)pairs

type-synonym

'a trace = 'a list

record ('s,'a)execution-module =

execs::('s,'a)execution set
asig::'a signature

record 'a trace-module =

traces::'a trace set
asig::'a signature

context IOA

begin

fun *is-exec-frag-of*::('s,'a)ioa \Rightarrow ('s,'a)execution \Rightarrow bool **where**
 is-exec-frag-of A (s,(ps#p')#p) =
 (snd p' -fst p-A \longrightarrow snd p \wedge *is-exec-frag-of* A (s, (ps#p')))
| *is-exec-frag-of* A (s, [p]) = s -fst p-A \longrightarrow snd p
| *is-exec-frag-of* A (s, []) = True

definition *is-exec-of*::('s,'a)ioa \Rightarrow ('s,'a)execution \Rightarrow bool **where**
 is-exec-of A e \equiv fst e \in start A \wedge *is-exec-frag-of* A e

definition *filter-act* **where**
 filter-act \equiv map fst

definition *schedule* **where**
 schedule \equiv *filter-act* o snd

definition *trace* **where**
 trace sig \equiv *filter* (λ a . a \in externals sig) o *schedule*

definition *is-schedule-of* **where**
 is-schedule-of A sch \equiv
 (\exists e . *is-exec-of* A e \wedge sch = *filter-act* (snd e))

definition *is-trace-of* **where**
 is-trace-of A tr \equiv
 (\exists sch . *is-schedule-of* A sch \wedge tr = *filter* (λ a . a \in ext A) sch)

definition *traces* **where**
 traces A \equiv {tr . *is-trace-of* A tr}

lemma *traces-alt*:
 shows *traces* A = {tr . \exists e . *is-exec-of* A e
 \wedge tr = *trace* (ioa.asig A) e}
 <proof>

lemmas *trace-simps* = *traces-def is-trace-of-def is-schedule-of-def filter-act-def is-exec-of-def*
 trace-def schedule-def

definition *proj-trace*::'a trace \Rightarrow ('a signature) \Rightarrow 'a trace (**infixr** <|> 12) **where**
 proj-trace t sig \equiv *filter* (λ a . a \in actions sig) t

definition *ioa-implements* :: ('s1,'a)ioa \Rightarrow ('s2,'a)ioa \Rightarrow bool (**infixr** <=<|> 12)
where
 A =<| B \equiv inp A = inp B \wedge out A = out B \wedge *traces* A \subseteq *traces* B

3.5 Operations on Executions

definition *cons-exec* **where**

$cons-exec\ e\ p \equiv (fst\ e, (snd\ e)\#p)$

definition *append-exec* **where**

$append-exec\ e\ e' \equiv (fst\ e, (snd\ e)\@(snd\ e'))$

fun *last-state* **where**

$last-state\ (s, []) = s$

| $last-state\ (s, ps\#p) = snd\ p$

lemma *last-state-reachable*:

fixes $A\ e$

assumes *is-exec-of* $A\ e$

shows *reachable* $A\ (last-state\ e)$ $\langle proof \rangle$

lemma *trans-from-last-state*:

assumes *is-exec-frag-of* $A\ e$ **and** $(last-state\ e) - a - A \longrightarrow s'$

shows *is-exec-frag-of* $A\ (cons-exec\ e\ (a, s'))$

$\langle proof \rangle$

lemma *exec-frag-prefix*:

fixes $A\ p\ ps$

assumes *is-exec-frag-of* $A\ (cons-exec\ e\ p)$

shows *is-exec-frag-of* $A\ e$

$\langle proof \rangle$

lemma *trace-same-ext*:

fixes $A\ B\ e$

assumes *ext* $A = ext\ B$

shows *trace* $(ioa.asig\ A)\ e = trace\ (ioa.asig\ B)\ e$

$\langle proof \rangle$

lemma *trace-append-is-append-trace*:

fixes $e\ e'\ sig$

shows *trace* $sig\ (append-exec\ e'\ e) = trace\ sig\ e' @ trace\ sig\ e$

$\langle proof \rangle$

lemma *append-exec-frags-is-exec-frag*:

fixes $e\ e'\ A\ as$

assumes *is-exec-frag-of* $A\ e$ **and** *last-state* $e = fst\ e'$

and *is-exec-frag-of* $A\ e'$

shows *is-exec-frag-of* $A\ (append-exec\ e\ e')$

$\langle proof \rangle$

lemma *last-state-of-append*:

fixes $e\ e'$

assumes *fst* $e' = last-state\ e$

shows *last-state* $(append-exec\ e\ e') = last-state\ e'$

$\langle proof \rangle$

end

end

4 Recoverable Data Types

theory *RDR*
imports *Main Sequences*
begin

4.1 The pre-RDR locale contains definitions later used in the RDR locale to state the properties of RDRs

locale *pre-RDR* = *Sequences* +
 fixes $\delta::'a \Rightarrow ('b \times 'c) \Rightarrow 'a$ (infix $\langle \cdot \rangle$ 65)
 and $\gamma::'a \Rightarrow ('b \times 'c) \Rightarrow 'd$
 and *bot*::'a ($\langle \perp \rangle$)
begin

fun *exec*::'a $\Rightarrow ('b \times 'c)list \Rightarrow 'a$ (infix $\langle \star \rangle$ 65) where
 exec *s Nil* = *s*
| *exec* *s (rs#r)* = (*exec* *s rs*) \cdot *r*

definition *less-eq* (infix $\langle \preceq \rangle$ 50) where
 less-eq *s s'* $\equiv \exists rs . s' = (s \star rs)$

definition *less* (infix $\langle \prec \rangle$ 50) where
 less *s s'* $\equiv less\text{-eq } s s' \wedge s \neq s'$

definition *is-lb* where
 is-lb *s s1 s2* $\equiv s \preceq s2 \wedge s \preceq s1$

definition *is-glb* where
 is-glb *s s1 s2* $\equiv is\text{-lb } s s1 s2 \wedge (\forall s' . is\text{-lb } s' s1 s2 \longrightarrow s' \preceq s)$

definition *contains* where
 contains *s r* $\equiv \exists rs . r \in set\ rs \wedge s = (\perp \star rs)$

definition *inf* (infix $\langle \sqcap \rangle$ 65) where
 inf *s1 s2* $\equiv THE\ s . is\text{-glb } s s1 s2$

4.2 Useful Lemmas in the pre-RDR locale

lemma *exec-cons*:
 $s \star (rs \# r) = (s \star rs) \cdot r$ *<proof>*

lemma *exec-append*:
 $(s \star rs) \star rs' = s \star (rs @ rs')$
<proof>

lemma *trans*:
assumes $s1 \preceq s2$ **and** $s2 \preceq s3$
shows $s1 \preceq s3$ *<proof>*

lemma *contains-star*:
fixes $s r rs$
assumes *contains* $s r$
shows *contains* $(s \star rs) r$
<proof>

lemma *preceq-star*: $s \star (rs \# r) \preceq s' \implies s \star rs \preceq s'$
<proof>

end

4.3 The RDR locale

locale *RDR* = *pre-RDR* +
assumes *idem1*: *contains* $s r \implies s \cdot r = s$
and *idem2*: $\bigwedge s r r' . \text{fst } r \neq \text{fst } r' \implies \gamma s r = \gamma ((s \cdot r) \cdot r')$
and *antisym*: $\bigwedge s1 s2 . s1 \preceq s2 \wedge s2 \preceq s1 \implies s1 = s2$
and *glb-exists*: $\bigwedge s1 s2 . \exists s . \text{is-glb } s s1 s2$
and *consistency*: $\bigwedge s1 s2 s3 rs . s1 \preceq s2 \implies s2 \preceq s3 \implies s3 = s1 \star rs$
 $\implies \exists rs' rs'' . s2 = s1 \star rs' \wedge s3 = s2 \star rs''$
 $\wedge \text{set } rs' \subseteq \text{set } rs \wedge \text{set } rs'' \subseteq \text{set } rs$
and *bot*: $\bigwedge s . \perp \preceq s$
begin

lemma *inf-glb*: *is-glb* $(s1 \sqcap s2) s1 s2$
<proof>

sublocale *ordering less-eq less*
<proof>

sublocale *semilattice-set inf*
<proof>

sublocale *semilattice-order-set inf less-eq less*
<proof>

notation $F (\lrcorner \sqcap \rightarrow [99])$

4.4 Some useful lemmas

lemma *idem-star*:
fixes $r s rs$
assumes *contains* $s r$
shows $s \star rs = s \star (\text{filter } (\lambda x . x \neq r) rs)$
<proof>

lemma *idem-star2*:

fixes $s\ rs'$

shows $\exists\ rs' . s \star rs = s \star rs' \wedge \text{set } rs' \subseteq \text{set } rs$

$\wedge (\forall\ r \in \text{set } rs' . \neg \text{contains } s\ r)$

<proof>

lemma *idem2-star*:

assumes *contains* $s\ r$

and $\bigwedge\ r' . r' \in \text{set } rs \implies \text{fst } r' \neq \text{fst } r$

shows $\gamma\ s\ r = \gamma\ (s \star rs)\ r$ *<proof>*

lemma *glb-common*:

fixes $s1\ s2\ s\ rs1\ rs2$

assumes $s1 = s \star rs1$ **and** $s2 = s \star rs2$

shows $\exists\ rs . s1 \sqcap s2 = s \star rs \wedge \text{set } rs \subseteq \text{set } rs1 \cup \text{set } rs2$

<proof>

lemma *glb-common-set*:

fixes $ss\ s0\ rset$

assumes *finite* ss **and** $ss \neq \{\}$

and $\bigwedge\ s . s \in ss \implies \exists\ rs . s = s0 \star rs \wedge \text{set } rs \subseteq rset$

shows $\exists\ rs . \bigsqcap\ ss = s0 \star rs \wedge \text{set } rs \subseteq rset$

<proof>

end

end

5 The SLin Automata specification

theory *SLin*

imports *IOA RDR*

begin

datatype (a,b,c,d) *SLin-action* =

— The nat component is the instance number

Invoke $\text{nat } b\ c$

| *Response* $\text{nat } b\ d$

| *Switch* $\text{nat } b\ c\ a$

| *Recover* nat

| *Linearize* nat

datatype *SLin-status* = *Sleep* | *Pending* | *Ready* | *Aborted*

record (a,b,c) *SLin-state* =

pending :: $b \Rightarrow b \times c$

initVals :: $a\ \text{set}$

abortVals :: $a\ \text{set}$

$status :: 'b \Rightarrow SLin\text{-}status$
 $dstate :: 'a$
 $initialized :: bool$

locale $SLin = RDR + IOA$
begin

definition

$asig :: nat \Rightarrow nat \Rightarrow ('a, 'b, 'c, 'd)SLin\text{-}action\ signature$

— The first instance has number 0

where

$asig\ i\ j \equiv ()$

$inputs = \{act . \exists p\ c\ iv\ i' .$

$(i \leq i' \wedge i' < j \wedge act = Invoke\ i'\ p\ c) \vee (i > 0 \wedge act = Switch\ i\ p\ c\ iv)\}$,

$outputs = \{act . \exists p\ c\ av\ i'\ outp .$

$(i \leq i' \wedge i' < j \wedge act = Response\ i'\ p\ outp) \vee act = Switch\ j\ p\ c\ av\}$,

$internals = \{act . \exists i' . i \leq i' \wedge i' < j$

$\wedge (act = Linearize\ i' \vee act = Recover\ i')\}$ $\})$

definition $pendingReqs :: ('a, 'b, 'c)SLin\text{-}state \Rightarrow ('b \times 'c)\ set$

where

$pendingReqs\ s \equiv \{r . \exists p .$

$r = pending\ s\ p$

$\wedge status\ s\ p \in \{Pending, Aborted\}\}$

definition $Inv :: 'b \Rightarrow 'c$

$\Rightarrow ('a, 'b, 'c)SLin\text{-}state \Rightarrow ('a, 'b, 'c)SLin\text{-}state \Rightarrow bool$

where

$Inv\ p\ c\ s\ s' \equiv$

$status\ s\ p = Ready$

$\wedge s' = s(pending := (pending\ s)(p := (p, c)),$

$status := (status\ s)(p := Pending))$

definition $pendingSeqs$ **where**

$pendingSeqs\ s \equiv \{rs . set\ rs \subseteq pendingReqs\ s\}$

definition $Lin :: ('a, 'b, 'c)SLin\text{-}state \Rightarrow ('a, 'b, 'c)SLin\text{-}state \Rightarrow bool$

where

$Lin\ s\ s' \equiv \exists rs \in pendingSeqs\ s .$

$initialized\ s$

$\wedge (\forall av \in abortVals\ s . (dstate\ s) \star rs \preceq av)$

$\wedge s' = s(dstate := (dstate\ s) \star rs)$

definition $initSets$ **where**

$initSets\ s \equiv \{ivs . ivs \neq \{\}\ \wedge ivs \subseteq initVals\ s\}$

definition $safeInits$ **where**

$safeInits\ s \equiv if\ initVals\ s = \{\}\ then\ \{\}$

$else\ \{d . \exists ivs \in initSets\ s . \exists rs \in pendingSeqs\ s .$

$$d = \prod ivs \star rs \wedge (\forall av \in abortVals\ s . d \preceq av)\}$$

definition *initAborts* **where**

$$\begin{aligned} initAborts\ s \equiv & \{ d . dstate\ s \preceq d \\ & \wedge ((\exists rs \in pendingSeqs\ s . d = dstate\ s \star rs) \\ & \vee (\exists ivs \in initSets\ s . dstate\ s \preceq \prod ivs \\ & \wedge (\exists rs \in pendingSeqs\ s . d = \prod ivs \star rs))) \} \end{aligned}$$

definition *uninitAborts* **where**

$$\begin{aligned} uninitAborts\ s \equiv & \{ d . \\ & \exists ivs \in initSets\ s . \exists rs \in pendingSeqs\ s . \\ & d = \prod ivs \star rs \} \end{aligned}$$

definition *safeAborts*::('a,'b,'c)SLin-state \Rightarrow 'a set **where**

$$\begin{aligned} safeAborts\ s \equiv & \text{if initialized } s \text{ then } initAborts\ s \\ & \text{else } uninitAborts\ s \end{aligned}$$

definition *Reco* :: ('a,'b,'c)SLin-state \Rightarrow ('a,'b,'c)SLin-state \Rightarrow bool

where

$$\begin{aligned} Reco\ s\ s' \equiv & \\ & (\exists p . status\ s\ p \neq Sleep) \\ & \wedge \neg initialized\ s \\ & \wedge (\exists d \in safeInits\ s . \\ & s' = s(dstate := d, initialized := True)) \end{aligned}$$

definition *Resp* :: 'b \Rightarrow 'd \Rightarrow ('a,'b,'c)SLin-state \Rightarrow ('a,'b,'c)SLin-state \Rightarrow bool

where

$$\begin{aligned} Resp\ p\ ou\ s\ s' \equiv & \\ & status\ s\ p = Pending \\ & \wedge initialized\ s \\ & \wedge contains\ (dstate\ s)\ (pending\ s\ p) \\ & \wedge ou = \gamma\ (dstate\ s)\ (pending\ s\ p) \\ & \wedge s' = s(status := (status\ s)(p := Ready)) \end{aligned}$$

definition *Init* :: 'b \Rightarrow 'c \Rightarrow 'a

$$\Rightarrow ('a,'b,'c)SLin-state \Rightarrow ('a,'b,'c)SLin-state \Rightarrow bool$$

where

$$\begin{aligned} Init\ p\ c\ iv\ s\ s' \equiv & \\ & status\ s\ p = Sleep \\ & \wedge s' = s(\{initVals := \{iv\} \cup (initVals\ s), \\ & status := (status\ s)(p := Pending), \\ & pending := (pending\ s)(p := (p,c)) \}) \end{aligned}$$

definition *Abort* :: 'b \Rightarrow 'c \Rightarrow 'a

$$\Rightarrow ('a,'b,'c)SLin-state \Rightarrow ('a,'b,'c)SLin-state \Rightarrow bool$$

where

$$\begin{aligned} Abort\ p\ c\ av\ s\ s' \equiv & \\ & status\ s\ p = Pending \wedge pending\ s\ p = (p,c) \\ & \wedge av \in safeAborts\ s \\ & \wedge s' = s(status := (status\ s)(p := Aborted), \end{aligned}$$

$abortVals := (abortVals\ s \cup \{av\})$

definition *trans* where

$trans\ i\ j \equiv \{ (s, a, s') . \text{case } a \text{ of}$
 $\quad Invoke\ i'\ p\ c \Rightarrow i \leq i' \wedge i < j \wedge Inv\ p\ c\ s\ s'$
 $\quad | Response\ i'\ p\ ou \Rightarrow i \leq i' \wedge i < j \wedge Resp\ p\ ou\ s\ s'$
 $\quad | Switch\ i'\ p\ c\ v \Rightarrow (i > 0 \wedge i' = i \wedge Init\ p\ c\ v\ s\ s')$
 $\quad \quad \vee (i' = j \wedge Abort\ p\ c\ v\ s\ s')$
 $\quad | Linearize\ i' \Rightarrow i' = i \wedge Lin\ s\ s'$
 $\quad | Recover\ i' \Rightarrow i > 0 \wedge i' = i \wedge Reco\ s\ s' \}$

definition *start* where

$start\ i \equiv \{ s .$
 $\quad \forall p . \text{status } s\ p = (\text{if } i > 0 \text{ then Sleep else Ready})$
 $\quad \wedge \text{dstate } s = \perp$
 $\quad \wedge (\text{if } i > 0 \text{ then } \neg \text{initialized } s \text{ else initialized } s)$
 $\quad \wedge \text{initVals } s = \{\}$
 $\quad \wedge \text{abortVals } s = \{\}\}$

definition *ioa* where

$ioa\ i\ j \equiv$
 $\quad (ioa.asig = asig\ i\ j ,$
 $\quad \text{start} = \text{start } i,$
 $\quad \text{trans} = \text{trans } i\ j)$

end

end

6 Definition and Soundness of Refinement Mappings, Forward Simulations and Backward Simulations

theory *Simulations*

imports *IOA*

begin

context *IOA*

begin

definition *refines* where

$refines\ e\ s\ a\ t\ A\ f \equiv \text{fst } e = f\ s \wedge \text{last-state } e = f\ t \wedge \text{is-exec-frag-of } A\ e$
 $\quad \wedge (\text{let } tr = \text{trace } (ioa.asig\ A)\ e \text{ in}$
 $\quad \quad \text{if } a \in \text{ext } A \text{ then } tr = [a] \text{ else } tr = [])$

definition

$is-ref-map :: ('s1 \Rightarrow 's2) \Rightarrow ('s1, 'a)ioa \Rightarrow ('s2, 'a)ioa \Rightarrow \text{bool}$ where
 $is-ref-map\ f\ B\ A \equiv$
 $(\forall s \in \text{start } B . f\ s \in \text{start } A) \wedge (\forall s\ t\ a . \text{reachable } B\ s \wedge s -a-B \longrightarrow t$

$\longrightarrow (\exists e . \text{refines } e \text{ s a t } A \text{ f })$

definition

is-forward-sim :: ($'s1 \Rightarrow ('s2 \text{ set})$) $\Rightarrow ('s1, 'a)ioa \Rightarrow ('s2, 'a)ioa \Rightarrow \text{bool}$ **where**
is-forward-sim $f B A \equiv$
 $(\forall s \in \text{start } B . f s \cap \text{start } A \neq \{\})$
 $\wedge (\forall s s' t a . s' \in f s \wedge s -a-B \longrightarrow t \wedge \text{reachable } B s$
 $\longrightarrow (\exists e . \text{fst } e = s' \wedge \text{last-state } e \in f t \wedge \text{is-exec-frag-of } A e$
 $\wedge (\text{let } tr = \text{trace } (ioa.asig A) e \text{ in}$
 $\text{if } a \in \text{ext } A \text{ then } tr = [a] \text{ else } tr = []))$

definition

is-backward-sim :: ($'s1 \Rightarrow ('s2 \text{ set})$) $\Rightarrow ('s1, 'a)ioa \Rightarrow ('s2, 'a)ioa \Rightarrow \text{bool}$ **where**
is-backward-sim $f B A \equiv$
 $(\forall s . f s \neq \{\})$ — Quantifying over reachable states would suffice
 $\wedge (\forall s \in \text{start } B . f s \subseteq \text{start } A)$
 $\wedge (\forall s t a t' . t' \in f t \wedge s -a-B \longrightarrow t \wedge \text{reachable } B s$
 $\longrightarrow (\exists e . \text{fst } e \in f s \wedge \text{last-state } e = t' \wedge \text{is-exec-frag-of } A e$
 $\wedge (\text{let } tr = \text{trace } (ioa.asig A) e \text{ in}$
 $\text{if } a \in \text{ext } A \text{ then } tr = [a] \text{ else } tr = []))$

6.1 A series of lemmas that will be useful in the soundness proofs

lemma *step-eq-traces*:

fixes $e-B' A e e-A' a t$

defines $e-A \equiv \text{append-exec } e-A' e$ **and** $e-B \equiv \text{cons-exec } e-B' (a, t)$

and $tr \equiv \text{trace } (ioa.asig A) e$

assumes $1:\text{trace } (ioa.asig A) e-A' = \text{trace } (ioa.asig A) e-B'$

and $2:\text{if } a \in \text{ext } A \text{ then } tr = [a] \text{ else } tr = []$

shows $\text{trace } (ioa.asig A) e-A = \text{trace } (ioa.asig A) e-B$

<proof>

lemma *exec-inc-imp-trace-inc*:

fixes $A B$

assumes $\text{ext } B = \text{ext } A$

and $\bigwedge e-B . \text{is-exec-of } B e-B$

$\implies \exists e-A . \text{is-exec-of } A e-A \wedge \text{trace } (ioa.asig A) e-A = \text{trace } (ioa.asig A) e-B$

shows $\text{traces } B \subseteq \text{traces } A$

<proof>

6.2 Soundness of Refinement Mappings

lemma *ref-map-execs*:

fixes $A::('sA, 'a)ioa$ **and** $B::('sB, 'a)ioa$ **and** $f::'sB \Rightarrow 'sA$ **and** $e-B$

assumes *is-ref-map* $f B A$ **and** *is-exec-of* $B e-B$

shows $\exists e-A . \text{is-exec-of } A e-A$

$\wedge \text{trace } (ioa.asig A) e-A = \text{trace } (ioa.asig A) e-B$

<proof>

theorem *ref-map-soundness*:
fixes $A::('sA,'a)ioa$ **and** $B::('sB,'a)ioa$ **and** $f::'sB \Rightarrow 'sA$
assumes *is-ref-map* $f B A$ **and** $ext A = ext B$
shows $traces B \subseteq traces A$
 $\langle proof \rangle$

6.3 Soundness of Forward Simulations

lemma *forward-sim-execs*:
fixes $A::('sA,'a)ioa$ **and** $B::('sB,'a)ioa$ **and** $f::'sB \Rightarrow 'sA$ *set* **and** $e-B$
assumes *is-forward-sim* $f B A$ **and** *is-exec-of* $B e-B$
shows $\exists e-A . is-exec-of A e-A$
 $\wedge trace (ioa.asig A) e-A = trace (ioa.asig A) e-B$
 $\langle proof \rangle$

theorem *forward-sim-soundness*:
fixes $A::('sA,'a)ioa$ **and** $B::('sB,'a)ioa$ **and** $f::'sB \Rightarrow 'sA$ *set*
assumes *is-forward-sim* $f B A$ **and** $ext A = ext B$
shows $traces B \subseteq traces A$
 $\langle proof \rangle$

6.4 Soundness of Backward Simulations

lemma *backward-sim-execs*:
fixes $A::('sA,'a)ioa$ **and** $B::('sB,'a)ioa$ **and** $f::'sB \Rightarrow 'sA$ *set* **and** $e-B$
assumes *is-backward-sim* $f B A$ **and** *is-exec-of* $B e-B$
shows $\exists e-A . is-exec-of A e-A$
 $\wedge trace (ioa.asig A) e-A = trace (ioa.asig A) e-B$
 $\langle proof \rangle$

theorem *backward-sim-soundness*:
fixes $A::('sA,'a)ioa$ **and** $B::('sB,'a)ioa$ **and** $f::'sB \Rightarrow 'sA$ *set*
assumes *is-backward-sim* $f B A$ **and** $ext A = ext B$
shows $traces B \subseteq traces A$
 $\langle proof \rangle$

end

end

7 Idempotence of the SLin I/O automaton

theory *Idempotence*
imports *SLin Simulations*
begin

locale *Idempotence* = *SLin* +
fixes $id1 id2 :: nat$

assumes $id1:0 < id1$ **and** $id2:id1 < id2$
begin

lemmas $ids = id1 id2$

definition *composition* **where**

composition \equiv
 $hide ((ioa\ 0\ id1) \parallel (ioa\ id1\ id2))$
 $\{act . \exists p\ c\ av . act = Switch\ id1\ p\ c\ av\ \}$

lemmas $comp-simps = hide-def\ composition-def\ ioa-def\ par2-def\ is-trans-def$
 $start-def\ actions-def\ asig-def\ trans-def$

lemmas $trans-defs = Inv-def\ Lin-def\ Resp-def\ Init-def$
 $Abort-def\ Reco-def$

declare *if-split-asm* [*split*]

7.1 A case rule for decomposing the transition relation of the composition of two SLins

declare *comp-simps* [*simp*]

lemma *trans-elim*:

fixes $s\ t\ a\ s'\ t'\ P$

assumes $(s,t) \xrightarrow{-a-} composition \longrightarrow (s',t')$

obtains

$(Invoke1)\ i\ p\ c$

where $Inv\ p\ c\ s\ s' \wedge t = t'$

and $i < id1$ **and** $a = Invoke\ i\ p\ c$

| $(Invoke2)\ i\ p\ c$

where $Inv\ p\ c\ t\ t' \wedge s = s'$

and $id1 \leq i \wedge i < id2$ **and** $a = Invoke\ i\ p\ c$

| $(Switch1)\ p\ c\ av$

where $Abort\ p\ c\ av\ s\ s' \wedge Init\ p\ c\ av\ t\ t'$

and $a = Switch\ id1\ p\ c\ av$

| $(Switch2)\ p\ c\ av$

where $s = s' \wedge Abort\ p\ c\ av\ t\ t'$

and $a = Switch\ id2\ p\ c\ av$

| $(Response1)\ i\ p\ ou$

where $Resp\ p\ ou\ s\ s' \wedge t = t'$

and $i < id1$ **and** $a = Response\ i\ p\ ou$

| $(Response2)\ i\ p\ ou$

where $Resp\ p\ ou\ t\ t' \wedge s = s'$

and $id1 \leq i \wedge i < id2$ **and** $a = Response\ i\ p\ ou$

| $(Lin1)\ Lin\ s\ s' \wedge t = t'$ **and** $a = Linearize\ 0$

| $(Lin2)\ Lin\ t\ t' \wedge s = s'$ **and** $a = Linearize\ id1$

| $(Reco2)\ Reco\ t\ t' \wedge s = s'$ **and** $a = Recover\ id1$

declare *comp-simps* [*simp del*]

7.2 Definition of the Refinement Mapping

fun $f :: (('a,'b,'c)SLin\text{-state} * ('a,'b,'c)SLin\text{-state}) \Rightarrow ('a,'b,'c)SLin\text{-state}$
where
 $f (s1, s2) =$
 ($pending = \lambda p. (if\ status\ s1\ p \neq\ Aborted\ then\ pending\ s1\ p\ else\ pending\ s2\ p)$),
 $initVals = \{\}$,
 $abortVals = abortVals\ s2$,
 $status = \lambda p. (if\ status\ s1\ p \neq\ Aborted\ then\ status\ s1\ p\ else\ status\ s2\ p)$),
 $dstate = (if\ dstate\ s2 = \perp\ then\ dstate\ s1\ else\ dstate\ s2)$,
 $initialized = True$)

7.3 Invariants

declare

$trans\text{-}defs\ [simp]$

fun $P1$ **where**

$P1 (s1,s2) = (\forall p . status\ s1\ p \in \{Pending, Aborted\}$
 $\longrightarrow fst (pending\ s1\ p) = p)$

fun $P2$ **where**

$P2 (s1,s2) = (\forall p . status\ s2\ p \neq Sleep \longrightarrow fst (pending\ s2\ p) = p)$

fun $P3$ **where**

$P3 (s1,s2) = (\forall p . (status\ s2\ p = Ready \longrightarrow initialized\ s2))$

fun $P4$ **where**

$P4 (s1,s2) = ((\forall p . status\ s2\ p = Sleep) = (initVals\ s2 = \{\}))$

fun $P5$ **where**

$P5 (s1,s2) = (\forall p . status\ s1\ p \neq Sleep \wedge initialized\ s1 \wedge initVals\ s1 = \{\})$

fun $P6$ **where**

$P6 (s1,s2) = (\forall p . (status\ s1\ p \neq Aborted) = (status\ s2\ p = Sleep))$

fun $P7$ **where**

$P7 (s1,s2) = (\forall c . status\ s1\ c = Aborted \wedge \neg initialized\ s2$
 $\longrightarrow (pending\ s2\ c = pending\ s1\ c \wedge status\ s2\ c \in \{Pending, Aborted\}))$

fun $P8$ **where**

$P8 (s1,s2) = (\forall iv \in initVals\ s2 . \exists rs \in pendingSeqs\ s1 .$
 $iv = dstate\ s1 \star rs)$

fun $P8a$ **where**

$P8a (s1,s2) = (\forall ivs \in initSets\ s2 . \exists rs \in pendingSeqs\ s1 .$
 $\prod ivs = dstate\ s1 \star rs)$

fun P9 where

$$P9 (s1,s2) = (\text{initialized } s2 \longrightarrow \text{dstate } s1 \preceq \text{dstate } s2)$$

fun P10 where

$$P10 (s1,s2) = ((\neg \text{initialized } s2) \longrightarrow (\text{dstate } s2 = \perp))$$

fun P11 where

$$P11 (s1,s2) = (\text{initVals } s2 = \text{abortVals } s1)$$

fun P12 where

$$P12 (s1,s2) = (\text{initialized } s2 \longrightarrow \sqcap (\text{initVals } s2) \preceq \text{dstate } s2)$$

fun P13 where

$$P13 (s1,s2) = (\text{finite } (\text{initVals } s2) \\ \wedge \text{finite } (\text{abortVals } s1) \wedge \text{finite } (\text{abortVals } s2))$$

fun P14 where

$$P14 (s1,s2) = (\text{initialized } s2 \longrightarrow \text{initVals } s2 \neq \{\})$$

fun P15 where

$$P15 (s1,s2) = (\forall av \in \text{abortVals } s1 . \text{dstate } s1 \preceq av)$$

fun P16 where

$$P16 (s1,s2) = (\text{dstate } s2 \neq \perp \longrightarrow \text{initialized } s2)$$

fun P17 where

— For the Response1 case of the refinement proof, in case a response is produced in the first instance and the second instance is already initialized

$$P17 (s1,s2) = (\text{initialized } s2 \\ \longrightarrow (\forall p . \\ ((\text{status } s1 p = \text{Ready} \\ \vee (\text{status } s1 p = \text{Pending} \wedge \text{contains } (\text{dstate } s1) (\text{pending } s1 p))) \\ \longrightarrow (\exists rs . \text{dstate } s2 = \text{dstate } s1 \star rs \wedge (\forall r \in \text{set } rs . \text{fst } r \neq p))) \\ \wedge ((\text{status } s1 p = \text{Pending} \wedge \neg \text{contains } (\text{dstate } s1) (\text{pending } s1 p)) \\ \longrightarrow (\exists rs . \text{dstate } s2 = \text{dstate } s1 \star rs \wedge (\forall r \in \text{set } rs . \\ \text{fst } r = p \longrightarrow r = \text{pending } s1 p))))))$$

fun P18 where

$$P18 (s1,s2) = (\text{abortVals } s2 \neq \{\} \longrightarrow (\exists p . \text{status } s2 p \neq \text{Sleep}))$$

fun P19 where

$$P19 (s1,s2) = (\text{abortVals } s2 \neq \{\} \longrightarrow \text{abortVals } s1 \neq \{\})$$

fun P20 where

$$P20 (s1,s2) = (\forall av \in \text{abortVals } s2 . \text{dstate } s2 \preceq av)$$

fun P21 where

$P21 (s1,s2) = (\forall av \in abortVals s2 . \prod (abortVals s1) \preceq av)$

fun *P22* **where**

$P22 (s1,s2) = (initialized s2 \longrightarrow dstate (f (s1,s2)) = dstate s2)$

fun *P23* **where**

$P23 (s1,s2) = ((\neg initialized s2) \longrightarrow$
 $pendingSeqs s1 \subseteq pendingSeqs (f (s1,s2)))$

fun *P25* **where**

$P25 (s1,s2) = (\forall ivs . (ivs \in initSets s2 \wedge initialized s2$
 $\wedge dstate s2 \preceq \prod ivs)$
 $\longrightarrow (\exists rs' \in pendingSeqs (f (s1,s2)) . \prod ivs = dstate s2 \star rs'))$

fun *P26* **where**

$P26 (s1,s2) = (\forall p . (status s1 p = Aborted$
 $\wedge \neg contains (dstate s2) (pending s1 p))$
 $\longrightarrow (status s2 p \in \{Pending, Aborted\}$
 $\wedge pending s1 p = pending s2 p))$

lemma *P1-invariant*:

shows *invariant (composition) P1*
{proof}

lemma *P2-invariant*:

shows *invariant (composition) P2*
{proof}

lemma *P16-invariant*:

shows *invariant (composition) P16*
{proof}

lemma *P3-invariant*:

shows *invariant (composition) P3*
{proof}

lemma *P4-invariant*:

shows *invariant (composition) P4*
{proof}

lemma *P5-invariant*:

shows *invariant (composition) P5*
{proof}

lemma *P13-invariant*:

shows *invariant (composition) P13*
{proof}

lemma *P20-invariant:*
shows *invariant (composition) P20*
{*proof*}

lemma *P18-invariant:*
shows *invariant (composition) P18*
{*proof*}

lemma *P14-invariant:*
shows *invariant (composition) P14*
{*proof*}

lemma *P15-invariant:*
shows *invariant (composition) P15*
{*proof*}

lemma *P6-invariant:*
shows *invariant (composition) P6*
{*proof*}

lemma *P7-invariant:*
shows *invariant (composition) P7*
{*proof*}

lemma *P10-invariant:*
shows *invariant (composition) P10*
{*proof*}

lemma *P11-invariant:*
shows *invariant (composition) P11*
{*proof*}

lemma *P8-invariant:*
shows *invariant (composition) P8*
{*proof*}

lemma *P8a-invariant:*
shows *invariant (composition) P8a*
{*proof*}

lemma *P12-invariant:*
shows *invariant (composition) P12*
{*proof*}

lemma *P19-invariant:*
shows *invariant (composition) P19*
{*proof*}

lemma *P9-invariant:*

shows invariant (composition) P9
<proof>

lemma P17-invariant:
shows invariant (composition) P17
<proof>

lemma P21-invariant:
shows invariant (composition) P21
<proof>

lemma P22-invariant:
shows invariant (composition) P22
<proof>

lemma P23-invariant:
shows invariant (composition) P23
<proof>

lemma P26-invariant:
shows invariant (composition) P26
<proof>

lemma P25-invariant:
shows invariant (composition) P25
<proof>

7.4 Proof of the Idempotence Theorem

theorem idempotence:
shows ((composition) =<| (ioa 0 id2))
<proof>

end

end

8 The Consensus Data Type

theory Consensus
imports RDR
begin

This theory provides a model for the RDR locale, thus showing that the assumption of the RDR locale are consistent.

typedecl proc
typedecl val

```

locale Consensus
— To avoid name clashes
begin

fun  $\delta::val\ option \Rightarrow (proc \times val) \Rightarrow val\ option$  (infix  $\langle \cdot \rangle$  65) where
   $\delta\ None\ r = Some\ (snd\ r)$ 
|  $\delta\ (Some\ v)\ r = Some\ v$ 

fun  $\gamma::val\ option \Rightarrow (proc \times val) \Rightarrow val$  where
   $\gamma\ None\ r = snd\ r$ 
|  $\gamma\ (Some\ v)\ r = v$ 

interpretation pre-RDR  $\delta\ \gamma\ None$   $\langle proof \rangle$ 
notation exec (infix  $\langle \star \rangle$  65)
notation less-eq (infix  $\langle \preceq \rangle$  50 )
notation None ( $\langle \perp \rangle$ )

lemma single-use:
  fixes  $r\ rs$ 
  shows  $\perp \star ([r]@rs) = Some\ (snd\ r)$ 
 $\langle proof \rangle$ 

lemma bot:  $\exists\ rs . s = \perp \star rs$ 
 $\langle proof \rangle$ 

lemma prec-eq-None-or-equal:
fixes  $s1\ s2$ 
assumes  $s1 \preceq s2$ 
shows  $s1 = None \vee s1 = s2$   $\langle proof \rangle$ 

interpretation RDR  $\delta\ \gamma\ \perp$ 
 $\langle proof \rangle$ 

end

end

```

9 Conclusion

In this document we have defined the SLin I/O-automaton (a shorthand for Speculative Linearizability) and we have proved that the composition of two instances of the SLin I/O-automaton behaves like a single instance of the SLin I/O-automaton. This theorem justifies the compositional proof technique presented in [4].

References

- [1] R. Guerraoui, V. Kuncak, and G. Losa. Speculative linearizability. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 55–66. ACM, 2012.
- [2] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [3] L. Lamport and K. Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.
- [4] G. Losa. *Modularity in the design of robust distributed algorithms*. PhD thesis, EPFL, 2014.
- [5] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.