Abortable Linearizable Modules

Rachid Guerraoui      Viktor Kuncak      Giuliano Losa

September 19, 2015

Abstract

We define the Abortable Linearizable Module automaton (ALM for short) and prove its key composition property using the IOA theory of HOLCF. The ALM is at the heart of the Speculative Linearizability framework. This framework simplifies devising correct speculative 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 typical example of a refinement proof in the I/O-automata framework of Lynch and Tuttle.

Contents

1 Introduction 2

2 Definition and properties of the longest common postfix of a set of lists 3

3 The ALM Automata specification 3

4 Proof that the composition of two instances of the ALM automaton behaves like a single instance of the ALM automaton 7
  4.1 A case split useful in the proofs 7
  4.2 Invariants of a single ALM instance 8
  4.3 Invariants of the composition of two ALM instances 9
  4.4 Proofs of invariance 10
  4.5 The refinement proof 12

5 Conclusion 12
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 present an I/O-automaton [4] specification, called ALM (a shorthand for Abortable Linearizable Module), which can be used to build a speculative linearizable algorithm out of independent modules that implement the different modes of the speculative algorithm. The ALM is at the heart of the Speculative Linearizability framework [1].

The ALM automaton produces traces that are linearizable with respect to a generic type of object. Moreover, the composition of two instances of the ALM automaton behaves like a single instance. Hence it is guaranteed that the composition of any number of instances of the ALM automaton is linearizable.

The properties stated above greatly 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 ALM 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 ALM automaton and prove the Composition Theorem, which states that the composition of two instances of the ALM automaton behaves as a single instance of the ALM automaton. We use a refinement mapping to establish this fact.
2 Definition and properties of the longest common postfix of a set of lists

theory LCP
imports Main ~/src/HOL/Library/Sublist
begin

definition common-postfix-p :: ('a list) set => 'a list => bool
— Predicate that recognizes the common postfix of a set of lists
— The common postfix of the empty set is the empty list
where
common-postfix-p ≡ λ xss xs. if xss = {} then xs = [] else ALL xs'. xs' ∈ xss
→ suffixeq xs xs'

definition l-c-p-pred :: 'a list set ⇒ 'a list ⇒ bool
— Predicate that recognizes the longest common postfix of a set of lists
where
l-c-p-pred ≡ λ xss xs. common-postfix-p xss xs ∧ (ALL xs'. common-postfix-p xss xs' → suffixeq xs' xs)

definition l-c-p :: 'a list set ⇒ 'a list
— The longest common postfix of a set of lists
where
l-c-p ≡ λ xss. THE xs. l-c-p-pred xss xs

lemma l-c-p-ok: l-c-p-pred xss (l-c-p xss)
— Proof that the definition of the longest common postfix of a set of lists is consistent

lemma l-c-p-lemma:
— A useful lemma
(ls ≠ {} ∧ (∀ l ∈ ls . (∃ l'. l = l' @ xs))) → suffixeq xs (l-c-p ls)

lemma l-c-p-common-postfix: common-postfix-p xss (l-c-p xss)
{proof}

lemma l-c-p-longest: common-postfix-p xss xs → suffixeq xs (l-c-p xss)
{proof}

end

3 The ALM Automata specification

theory ALM
imports ~/src/HOL/HOLCF/IOA/meta-theory/IOA LCP
begin

typedecl client
— A non-empty set of clients

typeddecl data
— Data contained in requests
datatype request =
— A request is composed of a sender and data
  Req client data

definition request-snd :: request ⇒ client
  where request-snd ≡ λ r. case r of Req c - ⇒ c

type-synonym hist = request list
— Type of histories of requests.
datatype ALM-action =
— The actions of the ALM automaton
  Invoke client request
  | Commit client nat hist
  | Switch client nat hist request
  | Initialize nat hist
  | Linearize nat hist
  | Abort nat
datatype phase = Sleep | Pending | Ready | Aborted
— Executions phases of a client
definition linearizations :: request set ⇒ hist set
— The possible linearizations of a set of requests
  where
  linearizations ≡ λ reqs . {h . set h ⊆ reqs ∧ distinct h}
definition postfix-all :: hist ⇒ hist set ⇒ hist set
— appends to the right the first argument to every member of the history set
  where
  postfix-all ≡ λ h hs . {h′. ∃ h′′. h′ = h′′ @ h ∧ h′′ ∈ hs}

definition ALM-asig :: nat ⇒ nat ⇒ ALM-action signature
— The action signature of ALM automata
— Input actions, output actions, and internal actions
  where
  ALM-asig ≡ λ id1 id2 . (act . ∃ c r h .
    {act . ∃ c r h .
      act = Invoke c r | act = Switch c id1 h r},
    {act . ∃ c r id′.
      id1 < id′ ∧ id′ < id2 ∧ act = Commit c id′ h
        | act = Switch c id2 h r},
    {act . ∃ h .
      act = Abort id1
        | act = Linearize id1 h}
record ALM-state =
  — The state of the ALM automata
  pending :: client ⇒ request
  — Associates a pending request to a client process
  initHists :: hist set
  — The set of init histories submitted by clients
  phase :: client ⇒ phase
  — Associates a phase to a client process
  hist :: hist
  — Represents the chosen linearization of the concurrent history of the current instance only
  aborted :: bool
  initialized :: bool

definition pendingReqs :: ALM-state ⇒ request set
  — the set of requests that have been invoked but that are not yet in the hist parameter
  where
    pendingReqs ≡ λ s . \{ r . ∃ c .
      r = pending s c
      ∧ r /∈ set (hist s)
      ∧ phase s c ∈ \{Pending, Aborted\}\}

definition initValidReqs :: ALM-state ⇒ request set
  — any request that appears in an init hist after the longest common prefix or that is pending
  where
    initValidReqs ≡ λ s . \{ r .
      (r ∈ pendingReqs s ∨ (∃ h ∈ initHists s . r ∈ set h))
      ∧ r /∈ set (l-c-p (initHists s))\}

definition ALM-trans :: nat ⇒ nat ⇒ (ALM-action, ALM-state)transition set
  — the transitions of the ALM automaton
  where
    ALM-trans ≡ λ id1 id2 . \{ trans .
      let s = fst trans; s’ = snd (snd trans); a = fst (snd trans) in

      case a of Invoke c r ⇒
        if phase s c = Ready ∧ request-snd r = c ∧ r /∈ set (hist s)
        then s’ = s[pending := (pending s)(c := r),
              phase := (phase s)(c := Pending)]
        else s’ = s

      Linearize i h ⇒
        initialized s ∧ ¬ aborted s
        ∧ h ∈ postfix-all (hist s) (linearizations (pendingReqs s s))
\( \land s' = s[\text{hist} := h] \)

| Initialize i h ⇒
  (\( \exists \ c . \ \text{phase} \ s \ c \neq \text{Sleep} \) \( \land \ \neg \ \text{aborted} \ s \ \land \ \neg \ \text{initialized} \ s \)
  \( \land h \in \text{postfix-all} \ (l-c-p \ (\text{initHists} \ s)) \) (linearizations (\( \text{initValidReqs} \ s \))
  \( \land s' = s[\text{hist} := h, \ \text{initialized} := \text{True}] \) |

| Abort i ⇒
  \( \neg \ \text{aborted} \ s \ \land \ (\exists \ c . \ \text{phase} \ s \ c \neq \text{Sleep}) \)
  \( \land s' = s[\text{aborted} := \text{True}] \) |

| Commit c i h ⇒
  phase s c = Pending \( \land \ \text{pending} \ s \ c \in \text{set} \ (\text{hist} \ s) \)
  \( \land h = \text{dropWhile} \ (\lambda \ r . \ r \neq \text{pending} \ s \ c) \ (\text{hist} \ s) \)
  \( \land s' = s[\text{phase} := (\text{phase} \ s)(c := \text{Ready})] \) |

| Switch c i h r ⇒
  if i = id1
  then if phase s c = Sleep
    then s' = s (\( \{\text{initHists} := \{h\} \cup (\text{initHists} \ s)\}, \)
      \( \text{phase} := (\text{phase} \ s)(c := \text{Pending}), \)
      \( \text{pending} := (\text{pending} \ s)(c := r)\))
    else s' = s
  else if i = id2
    then aborted s
    \( \land \ \text{phase} \ s \ c = \text{Pending} \land \ r = \text{pending} \ s \ c \)
    \( \land (\text{if} \ \text{initialized} \ s \)
      \then (h \in \text{postfix-all} \ (\text{hist} \ s) \) (linearizations (\( \text{pendingReqs} \ s \))))
    \else (h \in \text{postfix-all} \ (l-c-p \ (\text{initHists} \ s)) \) (linearizations (\( \text{initValidReqs} \ s \))))
    \( \land s' = s[\text{phase} := (\text{phase} \ s)(c := \text{Aborted})] \)
  else False |

**definition** ALM-start :: nat ⇒ ALM-state set
— the set of start states

**where**
ALM-start ≡ λ id . \{s .
  \( \forall \ c . \text{phase} \ s \ c = (\text{if} \ id \neq 0 \ \text{then} \text{Sleep} \ \text{else} \ \text{Ready}) \)
  \( \land \ \text{hist} \ s = [] \)
  \( \land \ \neg \ \text{aborted} \ s \)
  \( \land (\text{if} \ id \neq 0 \ \text{then} \ \neg \ \text{initialized} \ s \ \text{else} \ \text{initialized} \ s) \)
  \( \land \ \text{initHists} \ s = \{\} \})

**definition** ALM-ioa :: nat ⇒ nat ⇒ (ALM-action, ALM-state)ioa
— The ALM automaton

**where**
ALM-ioa ≡ λ (id1::nat) id2 .
   (ALM-asig id1 id2,
    ALM-start id1,
type-synonym com-p-o-state = ALM-state × ALM-state

definition composeALMs :: nat ⇒ nat ⇒ (ALM-action, com-p-o-state) ioa
— the composition of two ALMs

where
composeALMs ≡ λ id1 id2.

hide (ALM-ioa 0 id1 || ALM-ioa id1 id2)
{act . EX c tr r . act = Switch c id1 tr r}

end

4 Proof that the composition of two instances of the ALM automaton behaves like a single instance of the ALM automaton

theory CompositionCorrectness
imports ALM

begin

declare split-if-asm [split]
declare Let-def [simp]

4.1 A case split useful in the proofs

definition in-trans-cases-fun :: nat ⇒ nat ⇒ (ALM-state * ALM-state) ⇒ (ALM-state * ALM-state) ⇒ bool
— Helper function used to decompose proofs

where
in-trans-cases-fun ≡ % id1 id2 s ta tr .

(EX ca tr. (fst s, Invoke ca tr, fst t) : ALM-trans 0 id1 & (snd s, Invoke ca tr, snd t) : ALM-trans id1 id2)

| (EX ca tr. (fst s, Switch ca id1 tr, fst t) : ALM-trans 0 id1 & (snd s, Switch ca id1 tr, snd t) : ALM-trans id1 id2)

| (EX c id' tr. fst t = fst s & (snd s, Commit c id', snd t) : ALM-trans id1 id2 & id1 <= id' & id' < id2)

| (EX c tr. fst t = fst s & (snd s, Switch c id2 tr, snd t) : ALM-trans id1 id2)

| (EX h. fst t = fst s & (snd s, Linearize id1 h, snd t) : ALM-trans id1 id2)

| (fst t = fst s & (snd s, Abort id1, snd t) : ALM-trans id1 id2)

| (EX h. fst t = fst s & (snd s, Initialize id1 h, snd t) : ALM-trans id1 id2)

| (EX ca tr. (fst s, Switch ca 0 tr, fst t) : ALM-trans 0 id1 & snd t = snd s)

| (EX ca id' tr. (fst s, Commit ca id' tr, fst t) : ALM-trans 0 id1 & snd t = snd s & id' < id1)

| (EX h. (fst s, Linearize 0 h, fst t) : ALM-trans 0 id1 & snd t = snd s)

| (EX h. (fst s, Initialize 0 h, fst t) : ALM-trans 0 id1 & snd t = snd s)
lemma composeALMsE:
  — A rule for decomposing proofs
assumes id1 \sim 0 \text{ and } id1 < id2 \text{ and } \text{in-trans-comp} : s \Rightarrow (a::ALM\text{-action}) \Rightarrow \text{composeALMs id1 id2} \Rightarrow t
  shows \text{decomp} : \text{in-trans-cases-fun id1 id2 s t} 

(\text{proof})

lemma my-rule:
| id1 \neq 0; id1 < id2; s \Rightarrow \text{composeALMs id1 id2} \Rightarrow t; |

(\text{proof})

lemma my-rule2:
| 0 < id1; id1 < id2; s \Rightarrow \text{composeALMs id1 id2} \Rightarrow t; |

(\text{proof})

4.2 Invariants of a single ALM instance

definition P1a :: (ALM-state * ALM-state) \Rightarrow bool where
  — In ALM 1, a pending request of client c has client c as sender
P1a == \% s . let s1 = fst s; s2 = snd s in ALL c . phase s1 c \in \{Pending, Aborted\} \Rightarrow request-snd (pending s1 c) = c

definition P1b :: (ALM-state * ALM-state) \Rightarrow bool where
  — In ALM 2, a pending request of client c has client c as sender
P1b == \% s . let s1 = fst s; s2 = snd s in ALL c . phase s2 c \neq Sleep \Rightarrow request-snd (pending s2 c) = c

definition P2 :: (ALM-state * ALM-state) \Rightarrow bool where
  P2 == \% s . let s1 = fst s; s2 = snd s in (
  \forall c . phase s2 c = Sleep) \Rightarrow (\neg initialized s2 \land \text{hist s2} = [])

definition P3 :: (ALM-state * ALM-state) \Rightarrow bool where
  P3 == \% s . let s1 = fst s; s2 = snd s in
  \forall c . (phase s2 c = Ready \Rightarrow initialized s2)

definition P4 :: (ALM-state * ALM-state) \Rightarrow bool where
  — The set of init histories of ALM 2 is empty when no client ever invoked anything
P4 == \% s . let s1 = fst s; s2 = snd s in
  (\forall c . phase s2 c = Sleep) = (\text{initHists s2} = \{\})

definition P5 :: (ALM-state * ALM-state) \Rightarrow bool where
  — In ALM 1 a client never sleeps
P5 == \% s . let s1 = fst s; s2 = snd s in
  \forall c . phase s1 c \neq Sleep
4.3 Invariants of the composition of two ALM instances

definition P6 :: (ALM-state × ALM-state) ⇒ bool
— Non-interference accross instances
where
P6 == % s . let s1 = fst s; s2 = snd s in
  (∼ aborted s1 ---> (ALL c . phase s2 c = Sleep)) & (ALL c . phase
  s1 c = Aborted = (phase s2 c = Sleep))

definition P7 :: (ALM-state × ALM-state) ⇒ bool
— Before initialization of the ALM 2, pending requests are the same as in ALM
1 and no new requests may be accepted (phase is not Ready)
where
P7 == % s . let s1 = fst s; s2 = snd s in
  ALL c . phase s1 c = Aborted ∧ ¬ initialized s2 ---> (pending s2 c =
  pending s1 c ∧ phase s2 c ∈ {Pending, Aborted})

definition P8 :: (ALM-state × ALM-state) ⇒ bool
— Init histories of ALM 2 are built from the history of ALM 1 plus pending
requests of ALM 1
where
P8 == % s , let s1 = fst s; s2 = snd s in
  ∀ h ∈ initHists s2 . h ∈ postfix-all (hist s1) (linearizations (pendingReqs
  s1))

definition P9 :: (ALM-state × ALM-state) ⇒ bool
— ALM 2 does not abort before ALM 1 aborts
where
P9 == % s . let s1 = fst s; s2 = snd s in
  aborted s2 ---> aborted s1

definition P10 :: (ALM-state × ALM-state) ⇒ bool
— ALM 1 is always initialized and when ALM 2 is not initialized its history is
empty
where
P10 == % s . let s1 = fst s; s2 = snd s in
  initialized s1 ∧ (∼ initialized s2 ---> (hist s2 = []))

definition P11 :: (ALM-state × ALM-state) ⇒ bool
where
— After ALM 2 has been invoked and before it is initialized, any request found
in init histories after their longest common prefix is pending in ALM 1
P11 == % s . let s1 = fst s; s2 = snd s in
  ((∃ c . phase s2 c ≠ Sleep) ∧ ¬ initialized s2) ---> initValidReqs s2 ⊆
  pendingReqs s1

definition P12 :: (ALM-state × ALM-state) ⇒ bool
where
— After ALM 2 has been invoked and before it is initialized, the longest common
prefix of the init histories of ALM 2 is buit from appending a set of request pending
in ALM 1 to the history of ALM 1

\[ P12 \quad \text{let } s1 = \text{fst } s; s2 = \text{snd } s \text{ in } \\
(\exists \ c. \ \text{phase } s2 \ c \neq \text{Sleep}) \rightarrow (\exists \ rs. \ \text{l-c-p} (\text{initHists} \ s2) = \text{rs} @ (\text{hist} \ s1) \\
\land \ \text{set } \text{rs} \subseteq \text{pendingReqs} \ s1 \land \text{distinct } \text{rs}) \]

definition \( P13 \quad \text{let } s1 = \text{fst } s; s2 = \text{snd } s \text{ in } \\
(\exists \ c. \ \text{phase } s2 \ c \neq \text{Sleep}) \rightarrow \ \text{postfix-all} (\text{initHists} \ s2) = \text{rs} @ (\text{hist} \ s1) \\
\land \ \text{set } \text{rs} \subseteq \text{pendingReqs} \ s1 \land \text{distinct } \text{rs}) \)

\[ P14 \quad \text{let } s1 = \text{fst } s; s2 = \text{snd } s \text{ in } \\
(\text{hist} \ s2 \neq [] \lor \text{initialized} \ s2) \rightarrow (\exists \ rs. \ \text{hist} \ s2 = \text{rs} @ (\text{hist} \ s1) \\
\land \ \text{set } \text{rs} \cap \text{set} (\text{hist} \ s1) = \{\}) \]

definition \( P15 \quad \text{let } s1 = \text{fst } s; s2 = \text{snd } s \text{ in } \\
(\forall \ r. \ \text{let } c = \text{request-snd } r \text{ in } \text{phase} \ s2 \ c = \text{Sleep} \land r \in \text{set} (\text{hist} \ s2) \rightarrow (r \in \text{set} (\text{hist} \ s1) \lor r \in \text{pendingReqs} \ s1) \)

### 4.4 Proofs of invariance

\[ \text{lemma } \text{invariant-imp: } [\text{invariant } \text{ioa} \ P; \forall \ s. \ P \ s \rightarrow Q \ s] \implies \text{invariant } \text{ioa} \ Q \]

\[ \langle \text{proof} \rangle \]

\[ \text{declare } \text{phase.split} [\text{split}] \]
\[ \text{declare } \text{phase.split asm} [\text{split}] \]
\[ \text{declare } \text{ALM-action.split} [\text{split}] \]
\[ \text{declare } \text{ALM-action.split asm} [\text{split}] \]

\[ \text{lemma } \text{dropWhile-lemma: } \forall \ ys, zs = ys @ zs \land \text{hd } zs = x \land zs \neq [] \land x \notin \text{set } ys \rightarrow \text{dropWhile} (\lambda x'. x' \neq x) \ zs = zs \]

\[ \langle \text{proof} \rangle \]

\[ \text{lemma } P2\text{-invariant: } [[id1 < id2; id1 \neq 0]] \implies \text{invariant } (\text{composeALMs} \ id1 \ id2) \ P2 \]
lemma P5-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P5
(proof)

lemma P6-invariant: \([| \text{id1} \neq 0 ; \text{id1} < \text{id2} |] \implies \text{invariant (composeALMs id1 id2)} \) P6
(proof)

lemma P9-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P9
(proof)

lemma P10-invariant: \([| \text{id1} < \text{id2}; \text{id1} \sim= 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P10
(proof)

lemma P3-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P3
(proof)

lemma P7-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P7
(proof)

lemma P4-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P4
(proof)

lemma P8-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P8
(proof)

lemma P12-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P12
(proof)

lemma P11-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P11
(proof)

lemma P1a-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P1a
(proof)

lemma P1b-invariant: \([| \text{id1} < \text{id2}; \text{id1} \neq 0 |] \implies \text{invariant (composeALMs id1 id2)} \) P1b
(proof)
lemma \( P13\text{-invariant} : \left[ |id1 < id2; id1 \neq 0| \right] \implies \text{invariant} \ (\text{composeALMs id1 id2}) \ P13 \)
\( \langle \text{proof} \rangle \)

lemma \( P14\text{-invariant} : \left[ |id1 < id2; id1 \neq 0| \right] \implies \text{invariant} \ (\text{composeALMs id1 id2}) \ P14 \)
\( \langle \text{proof} \rangle \)

lemma \( P15\text{-invariant} : \left[ |id1 < id2; id1 \neq 0| \right] \implies \text{invariant} \ (\text{composeALMs id1 id2}) \ P15 \)
\( \langle \text{proof} \rangle \)

4.5 The refinement proof

definition ref-mapping :: \((\text{ALM-state} \ast \text{ALM-state}) \rightarrow \text{ALM-state}\)
— The refinement mapping between the composition of two ALMs and a single ALM

where
ref-mapping \( \equiv \lambda \left( s1, s2 \right) . \)
\( \{\text{pending} = \lambda c. \ (\text{if phase s1 c } \neq \text{Aborted then pending s1 c else pending s2 c}), \)
\( \text{initHists} = \{\}, \)
\( \text{phase} = \lambda c. \ (\text{if phase s1 c } \neq \text{Aborted then phase s1 c else phase s2 c}), \)
\( \text{hist} = \ (\text{if hist s2 } = [] \ then \ \text{hist s1 else hist s2}), \)
\( \text{aborted} = \text{aborted s2}, \)
\( \text{initialized} = \text{True}\} \)

theorem composition: \( [|id1 \neq 0; id1 < id2|] \implies ((\text{composeALMs id1 id2}) =<|\)
\( (\text{ALM-ioa 0 id2})) \)
— The composition theorem
\( \langle \text{proof} \rangle \)
end

5 Conclusion

In this document we have defined the ALM automaton (a shorthand for Aboratable Linearizable Modules) and we have proved that the composition of two instances of the ALM automaton behaves like a single instance of the ALM automaton. This theorem justifies the compositional proof technique presented in [1].

References

able at http://lara.epfl.ch/w/slin.

