Existence of Forward Simulations for Particular Data Structures

January 10, 2017

1 Preliminaries

Systems we consider are labeled transition systems (LTS):

Definition 1. *An LTS is defined over four-tuples $A = (Q, \Sigma, q_0, \delta)$ where Q is the set of states, Σ is the set of transition labels, $q_0 \in Q$ is the initial state and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.*

Executions generated by this system are alternating sequence of states and transition labels $\rho = s_0, e_0, s_1, ...s_k, e_k, ...$ where each $s_i \in Q$, each $e_i \in \Sigma$, $s_0 = q_0$ and each $(s_i, e_i s_{i+1}) \in \delta$. The projection of the sequence ρ over the set Π is denoted by $\rho|\Pi$, and it is the maximum subsequence of ρ consisting of elements of Π. Traces of the LTS are obtained from executions by projecting them over Σ. For the rest of the paper and in all of the proofs, we consider only finite executions (denoted as $E(A)$) and/or traces (denoted as $Tr(A)$ of the LTSs in focus.

Libraries are LTSs that provide methods. Let $\mathcal{M}$ be the set of method names and $\mathcal{D}$ be the domain of values as input/output parameters for the methods. Then, this library contains transition labels of the form $inv(m, d, i)$ representing the invocation of method $m \in \mathcal{M}$ with input value $d \in \mathcal{D}$. The third field is the operation identifier for differentiating the different calls of the same method from the set $\mathcal{O}$. For simplicity, we take $\mathcal{O} = \mathbb{N}$ for the rest of the paper. We also assume that methods could have at most one input parameter. If they do not have any input arguments (like pop method of a stack), we can omit the second field from the action. They also provide actions of the form $ret(m, d, i)$ representing the return of method $m \in M$ with value $d \in D$ which has been invoked previously with action $inv(m, d', i)$. Again, we assume that the methods can return at most one parameter and we may omit the second field from the action if they have none (like enqueue method of a queue). Before starting to reason about any set of libraries, we first fix the sets $\mathcal{M}$ and $\mathcal{D}$ and libraries in our focus agree on this sets. For any transition label $e = inv(m, d, i)$ or $e = ret(m, d, i)$, we have the function $oid(e) = i$.

Since libraries are LTSs, they produce traces. A trace $e = e_1, e_2, ..., e_n$ of library L is *well-formed* iff (i) every return matches an earlier invocation:

$e_j = ret(m, d, k)$ implies that there exists $i < j$ such that $e_i = inv(m, d', k)$ and (ii) every operation identifier is used at most one invocation/return pair: $oid(e_i) = oid(e_j) = k$ and $i < j$ implies $e_i = inv(m, d, k)$ and $e_j = ret(m, d', k)$. From now on, we assume that libraries produce well-formed traces. Let $f : \mathbb{N} \to (N)$ be a bijection. Then, traces e and e' are equivalent if e' is obtained from e by replacing every action $inv(m, d, k)$ with $inv(m, d, f(k))$ and every action $ret(m, d, k)$ with $ret(m, d, f(k))$.

Based on these definitions on the traces of the libraries, we can define refinement between libraries:

Definition 2. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. We define the set $A\Sigma = \{inv(m, d, i) | m \in \mathcal{M} \wedge d \in \mathcal{D} \wedge i \in \mathbb{N}\} \cup \{ret(m, d, i) | m \in \mathcal{M} \wedge d \in \mathcal{D} \wedge i \in \mathbb{N}\}$ as abstract transition labels. Note that $A\Sigma \subseteq \Sigma_{L_1}$ and $A\Sigma \subseteq \Sigma_{L_2}$. Then, we say L_1 refines L_2 iff for every trace $e \in Tr(L_1)$, there exists a trace $e' \in Tr(L_2)$ such that $e|A\Sigma$ is equivalent to $e'|A\Sigma$.*

Linearizability is also a relation between two libraries and it is stricter than refinement. It requires e' in Definition 2 to be a sequential one. A trace e is sequential iff following two conditions hold for its projection to abstract transition labels $e|A\Sigma = e_1, ...e_n$: (i) $e_1 = inv(m, d, k)$ for some $m \in \mathcal{M}, d \in \mathcal{D} and k \in \mathbb{N}$ and (ii) for all $i \in [1, n)$, either $e_i = inv(m, d, k)$ and $e_{i+1} = ret(m, d', k)$ or $e_i = ret(m, d, k)$ and $e_{i+1} = inv(m', d', k')$ for some $m, m' \in \mathcal{M}$, $d, d' \in \mathcal{D}$ and $k, k' \in \mathbb{N}$.

We can extend the relations between libraries by introducing simulation relations. We will later show that simulation relations imply refinement.

Definition 3. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. Then, the relation $fs \subseteq Q_{L_1} \times Q_{L_2}$ is called a forward simulation iff the following holds:*

(i) $fs[q_{0_{L_1}}] = \{q_{0_{L_2}}\}$

(ii-a) *If $(s, inv(m, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that $a_1 = inv(m, d, k)$ and for all $i \in [2, n]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$. The expression $u \xrightarrow{a} u'$ means that there exists a sequence of states $u_1, u_2, ..., u_{n+1}$ such that $u_1 = u$, $u_{n+1} = u'$ and for all $i \in [1, n]$, $(u_i, a_i, u_{i+1}) \in \delta_{L_2}$.*

(ii-b) *If $(s, ret(m, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that $a_n = ret(m, d, k)$ and for all $i \in [1, n-1]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$.*

(ii-c) *If $(s, t, s') \in \delta_{L_1}$ for some $t \in \Sigma_{L_1} \backslash A\Sigma$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that for all $i \in [1, n]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$. Moreover, a could be the empty sequence.*

If $fs[s]$ is a unique state for all $s \in Q_{L_1}$ then it is called a refinement mapping/function. A dual notion of forward simulation is the backward simulation:

Definition 4. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. Then, the relation $bs \subseteq Q_{L_1} \times Q_{L_2}$ is called a backward simulation iff the following holds:*

(i) $bs[q_{0_{L_1}}] = \{q_{0_{L_2}}\}$

(ii-a) If $(s, inv(m,d,k), s') \in \delta_{L_1}$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that* $a_1 = inv(m,d,k)$ *and for all* $i \in [2,n]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*.*

(ii-b) If $(s, ret(m,d,k), s') \in \delta_{L_1}$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that* $a_n = ret(m,d,k)$ *and for all* $i \in [1, n-1]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*.*

(ii-c) If $(s, t, s') \in \delta_{L_1}$ *for some* $t \in \Sigma_{L_1} \backslash A\Sigma$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that for all* $i \in [1,n]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*. Moreover, a could be the empty sequence.*

Simulation relations are used to prove refinement relations among libraries. Following lemmas show their soundness:

Lemma 1. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. If fs (bs) is a forward (backward) simulation relating L_1 to L_2, then L_1 refines L_2.*

Proof. Looks trivial and follows Lynch paper. Can be completed later. □

2 Existence of Forward Simulations for Queue Implementations that have Fixed Dequeue Linearization Points

In this section, we will show that for any concurrent queue implementation library L_C for which we know the linearization points of the dequeue operation and that is linearizable with respect to the reference implementation library L_A, there exists a forward simulation fs relating L_C to L_I where L_I is an intermediate library equivalent to L_A i.e. L_I refines L_A and vice versa.

For all the queue libraries, we fix $\mathcal{M} = \{enq, deq\}$ and $\mathcal{D} = \mathbb{N} \cup \{\texttt{EMPTY}\}$. Since we know the linearization point of dequeues, we extend the definitions of the previous sections adding this information. First, we extend the set $A\Sigma$ introduced in Definition 2 for queues in our focus as $AQ\Sigma = A\Sigma \cup \{lin(deq,d,k) | d \in \mathcal{D}, k \in \mathbb{N}\}$. For any library L we consider in this section, $AQ\Sigma \subseteq \Sigma_L$. Then, we define q-refinement by replacing $A\Sigma$ with $AQ\Sigma$ in Definition 2. We also define q-linearizability, by enforcing $lin(deq,d,k)$ to appear immediately after $inv(deq,k)$ and immediately before $ret(deq,d,k)$ in a sequential execution. We also change definition of forward and backward simulation relations by replacing $A\Sigma$ with $AQ\Sigma$ in the original definitions and adding a new condition $(ii-d)$ to each of them:

Forward Simulation: (ii-d) If $(s, lin(deq, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $(u, lin(deq, d, k), u') \in \delta_{L_2}$

Backward Simulation: (ii-d) If $(s, lin(deq, d, k), s') \in \delta_{L_1}$ and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $(u, lin(deq, d, k), u') \in \delta_{L_2}$

Lemma 1 of the previous section still holds if we replace refinement with q-refinement and use new definitions of backward and forward simulations.

Next, we define the abstract library L_A as follows:

- A queue state consists of a finite sequence of natural numbers representing the queue content and a program counter for each operation. More formally: $Q_A \subseteq \mathbb{N}^* \times (\mathbb{N} \to Lbl_A)$ where $Lbl_A = \{N, E_0, E_1, E_2, D_0, D_1, D_2\}$ is the set of transition labels of the operations. Operations that have not started yet are mapped to N. E_i (D_i) denotes particular transitions in enqueue (dequeue) operations that will be clear when we define δ_A. For a state q, we denote the contents of the queue (first field) with s_q and the function that maps operations to labels (the second field) with f_q.

- Transition labels consists of invocations, returns and linearizations of enqueue and dequeue operations: $\Sigma_A = AQ\Sigma \cup \{lin(enq, d, k) | d \in \mathcal{D}, k \in \mathbb{N}\}$. Invocation of *deq* operation does not have any input and return of *enq* operation does not have any output value. We omit second fields from these labels.

- Initial state is the empty queue: $q_{0A} = (\langle\rangle, f_{q_{0A}})$ where $f_{q_{0A}}(i) = N$ for all $i \in \mathbb{N}$.

- Each operation consists of invocation, linearization and return steps. These are reflected in the transition relation. For the below definitions, unchanged parts of the state are omitted from the formulae:
  - $(q, inv(enq, d, k), q') \in \delta_A$ iff $d \neq \texttt{EMPTY} \wedge f_q(k) = N \wedge f_{q'}(k) = E_0$,
  - $(q, lin(enq, d, k), q') \in \delta_A$ iff $f_q(k) = E_0 \wedge s_{q'} = s_q \circ \langle d \rangle \wedge f_{q'}(k) = E_1$ where $\circ$ is the operation that concatenates two finite sequences,
  - $(q, ret(enq, k), q') \in \delta_A$ iff $f_q(k) = E_1 \wedge f_{q'}(k) = E_2$,
  - $(q, inv(deq, k), q') \in \delta_A$ iff $f_q(k) = N \wedge f_{q'}(k) = D_0$
  - $(q, lin(deq, d, k), q') \in \delta_A$ iff $f_q(k) = D_0 \wedge (d \neq \texttt{EMPTY} \wedge s_q = \langle d \rangle \circ s_{q'} \vee d = \texttt{EMPTY} \wedge s_q = s_{q'} = \langle\rangle) \wedge f_{q'}(k) = D_1$,
  - $(q, ret(deq, d, k), q') \in \delta_A$ iff $f_q(k) = D_1 \wedge f_{q'}(k) = D_2$.

We restrict traces generated by this LTS by neglecting the invalid traces. If we project a trace to some operation identifier k and obtain a sequence $\langle ..., inv(enq, d, k), lin(enq, d', k), ... \rangle$ or $\langle ..., lin(deq, d, k), ret(deq, d', k), ... \rangle$ where $d \neq d'$ we say that this trace is invalid.

We want to introduce L_I as the next step. States of L_I consists of strict orders of enqueue operations. Nodes of the strict order come from the set

$ND = \mathbb{N} \times \mathcal{D} \times \{\texttt{PENDING}, \texttt{CLOSED}\}$. Basically each node is a tuple keeping the operation ID, the value to be enqueued and if this enqueue is pending or closed. Strict order also keeps a set of directed edges $ed \subseteq ED = ND \times ND$. Then, a strict order is a tuple (nd, ed) where the set ed obeys the assumption of strict order i.e. irreflexivity, asymmetry and transitivity. Then, the LTS of L_I is defined as follows:

- A state consists of a partial order and a function keeping the program counter. More formally $Q_I \subseteq ND \times ED \times (\mathbb{N} \to Lbl_I)$ where $Lbl_I = \{N, E_0, E_1, D_0, D_1, D_2\}$ is the set of transition labels. nd_q ed_q and f_q denotes the nodes of the strict order in state q, edges of the strict order in state q and the function that maps operation to labels in state q, respectively.

- The transition label set consists of invocation and return action of both methods and linearization action of only the dequeue method: $\Sigma_I = AQ\Sigma$. Invocation of *deq* operation does not have any input and return of *enq* operation does not have any output value. We omit second fields from these labels.

- Initial state consists of an empty strict order and a function mapping every operation to N: $q_{0I} = (so_{q_{0I}}, f_{q_{0I}}$ where $so_{q_{0I}} = (\emptyset, \emptyset)$ and $f_{q_{0I}}(i) = N$ for all $i \in \mathbb{N}$.

- While defining δ_I, we again omit mentioning about the parts that has not changed:

  - $(q, inv(enq, d, k), q') \in \delta_I$ iff $f_q(k) = N \wedge f_{q'}(k) = E_0 \wedge d \neq \texttt{EMPTY} \wedge (k, \_, \_) \notin nd_q \wedge nd_{q'} = nd_q \cup \{(k, d, \texttt{PENDING}\} \wedge AddNode(ed_{q'}, ed_q, k)$ where $AddNode(ed_{q'}, ed_q, k)$ is true iff $ed_{q'}$ is obtained from ed_q by adding edges from every closed node of ed_q to the node with identifier k.
  - $(q, ret(enq, k), q') \in \delta_I$ iff $f_q(k) = E_0 \wedge f_{q'}(k) = E_1 \wedge ((k, \_, \texttt{PENDING}) \notin nd_q \wedge so_q = so_{q'} \vee UpdateNode(so_{q'}, so_q, k))$ where $UpdateNode(so_{q'}, so_q, k)$ is true iff $(k, d, \texttt{PENDING}) \in nd_q$ for some $d \in \mathcal{D}$, it is replaced with node $(k, d, \texttt{CLOSED}$ in the state q' and all the edges adjacent to $(k, d, \texttt{PENDING})$ in state q are replaced with edges adjacent to $(k, d, \texttt{CLOSED})$ in the state q'.
  - $(q, inv(deq, k), q') \in \delta_I$ iff $f_q(k) = N \wedge f_{q'}(k) = D_0$.
  - $(q, lin(deq, d, k), q') \in \delta_I$ iff $f_q(k) = D_0 \wedge f_{q'}(k) = D_1 \wedge (d = \texttt{EMPTY} \wedge nd_q = ed_q = nd_{q'} = ed_{q'}\emptyset \vee d \neq \texttt{EMPTY} \wedge RemoveNode(so_q, so_{q'}, d))$ where $RemoveNode(so_q, so_{q'}, d)$ is true iff there is a minimal node $n \in so_q$ of which data value (second field) is d and $so_{q'}$ is obtained from so_q by removing n and all the edges adjacent to it. Note that linearization of dequeue could remove a PENDING node.
  - $(q, ret(deq, d, k)q') \in \delta_I$ iff $f_q(k) = D_1 \wedge f_{q'}(k) = D_2$.

Again, we omit the inconsistent traces from L_I as in L_A. Note that the library L_I is deterministic with respect to alphabe $AQ\Sigma$.

Next, we show equivalence of L_A and L_I in terms of q-refinement.

Lemma 2. *L_I is a q-refinement of L_A.*

Proof. We will provide a backward simulation relation btw L_I and L_A. Given a state $q = (so_q, f_q) \in Q_I$, $q' \in bs[q]$ iff (i) $s_{q'}$ is a linearization of so_q projected to d fields (second field). This linearization may omit the pending elements (the ones of which third field is `PENDING`), but it must contain all the `CLOSED` elements. Note that pending elements of so_q are maximal and they can appear after the closed elements, towards the end of the sequence. (ii) If $f_q(k) \in \{N, D_0, D_1, D_2\}$, then $f_{q'}(k) = f_q(k)$. If $f_q(k) = E_0$, $(k, d, \texttt{PENDING}) \in nd_q$ and this node participates in the linearization $s_{q'}$. Then $f_{q'}(k) = E_1$. If it does not participate in the linearization, then $f_{q'}(k) = E_0$. If $f_q(k) = E_0$ but there is no node with operation identifier k in nd_q, then $f_{q'} = E_1$. Lastly, if $f_q(k) = E_1$ then $f_{q'}(k) = E_2$. Next, we will show that bs is a backward simulation relation.

$\langle i \rangle$ $bs[q_{0\,I}] = \{q_{0\,A}\}$.

$\langle ii - a - enq \rangle$ Let $(q, inv(enq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We know that $(k, d, \texttt{PENDING}$ is a maximal element in $so_{q'}$ and either $s_{t'}$ does not linearize it or $s_{t'} = \rho \circ \langle d \rangle \circ \pi$ where π consists of only linearization of `PENDING` elements. For the first case $s_{t'} = s_t$ for some $t \in bs[q]$ and $(t, inv(enq, d, k), t') \in \delta_A$. For the latter case, $s_t = \rho$ for some $t \in bs[q]$ and $t \xrightarrow{a}_{L_A} t'$ where $a = inv(enq, d, k), lin(enq, d, k), lin(enq, d_1, k_1), ..., lin(enq, d_j, k_j)$ where $\pi = d_1, ...d_j$. The f_t could be obtained easily by observing the sequence a and it can be checked that such $t \in bs[q]$ exists.

$\langle ii - a - deq \rangle$ Let $(q, inv(deq, k), q') \in \delta_I$ and $t' \in bs[q']$. We know that $f_{t'}(k) = D_0$. We know that there is a $t \in bs[q]$ such that $s_t = s_{t'}$ (since $so_q = so_{q'}$), $f_t(k) = N$ and $(t, inv(deq, k), t') \in \delta_A$.

$\langle ii - d \rangle$ Let $(q, lin(deq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. First consider the case $d \neq$ `EMPTY`. We have two cases: Either the node with operation id k in the so_q was `PENDING` or `CLOSED`. For both of the cases, we can find $t \in bs[q]$ such that $s_t = \langle d \rangle \circ s_{t'}$ and $f_t(k) = D_0$. Hence $(t, lin(deq, d, k), t') \in \delta_I$. Next, consider the case $d =$ `EMPTY`. Then, we know that $s_{t'} = \langle\rangle$ and there exists $t \in bs[q]$ such that $s_t = \langle\rangle$ and $f_t = D_0$. Hence $(t, lin(deq, d, k), t') \in \delta_A$.

$\langle ii - b - enq \rangle$ Let $(q, ret(enq, k), q') \in \delta_I$ and $t' \in bs[q']$. There are two possible cases: There exists a node $(k, d, \texttt{PENDING}) \in nd_q$ or for all nodes $n = (k, \_, \_)$, $n \notin nd_q$. For the former case, $s_{t'} = \rho \circ \langle d \rangle \circ \pi$ where ρ is linearization of closed nodes in $nd_{q'}$ and π is linearization of some open nodes in $node_{q'}$. We also know that $f_{t'}(k) = E_2$. Then, there exists a node $t \in bs[q]$ such that $s_t = s_{t'}$ (since nodes that generate ρ are closed in q and that generate π are open in q) and $f_t(k) = E_1$. Hence, $(t, ret(deq, k), t') \in \delta_A$. For the latter case, we know that $so_q = so_{q'}$. Therefore, there exists $t \in bs[q]$ such that $s_t = s_{t'}$. Moreover $f_t(k) = E_1$. Hence, $(t, ret(enq, k), t') \in \delta_A$.

$\langle ii - b - deq \rangle$ Let $(q, ret(deq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. Then, we know that $f_{t'}(k) = D_2$ and there exists a $t \in bs[q]$ such that $s_t = s_{t'}$ (since $so_q = so_{q'}$) and $f_t(k) = D_1$. Then, $(t, ret(deq, d, k) \in \delta_A$.

□

Lemma 3. *L_A is a q-refinement of L_I.*

Proof. Since L_I is deterministic with respect to the alphabet $AQ\Sigma$, we should be able to find a forward simulation between L_A and L_I if our claim is correct. We propose a forward simulation by adding some auxiliary varibles to the state of L_A. In the new augmented state of L_A, the sequence does not only keep the elements of the queue but also the operation identifiers that enqueued them. Hence, the sequence consists of pairs of the form $s_q(i) = (k, d)$ where $q \in Q_A$, $i \in \mathbb{N}$ is an index, $d \in \mathcal{D} \backslash \{\texttt{EMPTY}\}$ is a value and $k \in \mathbb{N}$ is an operation identifier. We also add a set $InvEnq_q$ component to state that keeps the enqueue operations at the point E_0 i.e. enqueues that are invoked bu have not been linearized yet. Elements of $InvEnq_q$ are pairs of the form (d, k) where $d \in \mathcal{D} \backslash \{\texttt{EMPTY}\}$ is a value and $k \in \mathbb{N}$ is an operation identifier. Then our forward simulation $fs \subseteq Q_A \times Q_I$ relates the state $q = (s_q, f_q) \in Q_A$ to a state $q' = (so_{q'}, f_{q'}) \in Q_I$ iff (i) $f_{q'}(k) = f_q(k)$ for all $f_q(k) \in \{N, D_0, D_1, D_2\}$, (ii) $f_q(k) = E_0$ or $f_q(k) = E_1$ implies $f_{q'}(k) = E_0$, (iii) $f_q(k) = E_2$ implies $f_{q'}(k) = E_1$, (iv) If $(d, k) \in InvEnq_q$ or there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ and $f_q(k) = E_1$, then $(k, d, \texttt{PENDING})$ is a node in $so_{q'}$, (v) if there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ and $f_q(k) = E_2$ then $(k, d, \texttt{CLOSED})$ is a node in $so_{q'}$, (vi) open nodes in $so_{q'}$ are maximal and (vii) there exists a linearization (total order) $lo_{q'}$ of $so_{q'}$ that may omit some open nodes of $so_{q'}$ such that if we project nodes of $lo_{q'}$ to the first two fields, this linear order is equal to the sequence s_q.

Next, we will show that fs is a forward simulation relation:

$\langle i \rangle$ $fs[(q_0)_A] = \{(q_0)_I\}$

$\langle ii - a - enq \rangle$ Let $(q, inv(enq, d, k), q') \in \delta_A$ and $t \in fs[q]$. We can obtain t' such that $(t, inv(enq, d, k), t') \in \delta_I$. We show that $t' \in fs[q']$ by checking seven conditions of our fs relation. We omit the node (k, d) while linearizing $so_{t'}$ and obtain $s_{q'}$.

$\langle ii - a - deq \rangle$ Let $(q, inv(deq, k), q') \in \delta_A$ and $t \in fs[q]$. We obtain t' such that $(t, inv(deq, k), t') \in \delta_I$. The only difference between t' and t is that $f_{t'}(k) = D_0$. We can again see that $t' \in fs[q']$.

$\langle ii - c \rangle$ Let $(q, lin(enq, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick $t' = t$. We can see that $t \in fs[q']$. While obtaining linearization of $so_{t'}$, we do not neglect the node $(k, d, \texttt{PENDING})$ this time, although we neglect it in the linearization of so_t.

$\langle ii - d \rangle$ Let $(q, lin(deq, d, k), q') \in \delta_A$ and $t \in fs[q]$. Obtain t' such that $(t, lin(deq, d, k), t') \in \delta_I$. In s_q, (k, d) must be the minimum element. Hence, $(k, d, \_)$ is a minimal node in so_t and $lin(deq, d, k)$ is an enabled action in L_I. This action removes the node $(k, d, \_)$ from so_t and changes $f_t(k) = D_0$ to $f_{t'}(k) = D_1$. We can see that $t' \in fs[q']$ by checking the seven conditions.

$\langle ii - b - enq \rangle$ Let $(q, ret(enq, k), q') \in \delta_A$ and $t \in fs[q]$. There are two cases: Either there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ or there is no such i. For the former case, there is no node of the form $(k, d, \_)$ in so_t. We obtain t' such that $(t, ret(enq, k), t') \in \delta_I$. Then, there is no node of the form $(k, d, \_)$ in $so_{t'}$ neither and $so_{t'} = so_t$. since $s_q = s_{q'}$ also holds for this case, $t' \in fs[q']$ holds. For the latter case, we again pick t' such that $(t, ret(enq, k), t') \in \delta_I$. Again, $so_t = so_{t'}$ and $s_q = s_{q'}$ holds and $t' \in fs[q']$ can be observed.

$\langle ii - b - deq \rangle$ Let $(q, ret(deq, d, k), q') \in \delta_A$ and $t \in fs[q]$. We pick t' such that $(t, ret(deq, d, k), t') \in \delta_I$. We know that $so_t = so_{t'}$ and only change is $f_{t'}(k) = D_2$. Since we know that $s_q = s_{q'}$ and $f_{q'}(k) = D_2$, $t' \in fs[q']$ holds.

□

3 Existence of Forward Simulations for Stack Implementations that have Fixed Pop Linearization Points

4 Relaxation for the Data Structures Without Fixed Remove Linearization Points

We have observed that some implementations (like time-stamped stack) do not have fixed remove (pop) linearization points that will correspond to $lin(pop, e, k)$ where e could be a data value or `EMPTY`. However, we observe that, these implementations contain some points in their pop methods that logically removes the element from the pool. We call them commit points. If a method of a library has a fixed linearization point, it is also a commit point. In this sense, commit points are weaker versions of fixed linearization points. Fixed linearization point of a pop preserves the following properties:

- If a *ret* comes before a linearization point in the concrete execution, this order is preserved in the linearization of this execution.
- If a *lin* pop comes before another *lin* pop in the concrete execution, this order is preserved in the linearization of this execution.
- If a *lin* pop comes before an *inv* in the concrete execution, this order is preserved in the linearization of this execution.

A commit point is weaker than a fixed linearization point in the sense that it does not need to satisfy the first and the second conditions.

To our intuition, if a pop (remove) method has multiple finite linearization points, commit point is the latest element in this set. We have never observed an implementation in which a pop is linearized after it logically removes the element from the data structure.

From now on, we will restrict ourselves to stacks, since our example implementation that we will show linearizability of is the time-stamped (TS) stack. However, the notions and the machines we will introduce can be extended to queues easily.

We fix $\mathcal{M} = \{push, pop\}$ and $\mathcal{D} = \mathbb{N} \cup \{\texttt{EMPTY}\}$. We extend the alphabet $A\Sigma$ for stacks with commit points as $ACS\Sigma = A\Sigma \cup \{com(pop, d, k) | d \in \mathcal{D}, k \in \mathcal{M}\}$. We define cs-refinement and cs-linearizability as we defined q-refinement, q-linearizability in the previous sections. We also change definitions of backward and forward simulation relations for stacks with commit points as we do in the previous sections by replacing linearization points with commit points in this extensions. Lemma 1 of the first section still holds with new simulation relation definitions and cs-linearizability and cs-linearizability implies the original linearizability definition.

Our road map for this section is as follows: We will first introduce an intermediate stack machine L_I that will be deterministic with respect to the alphabet $ACS\Sigma$. We will show that L_I is equivalent to the standard abstract stack L_A defined in the previous section with respect to the language $A\Sigma$. We show this by first showing that L_I is a refinement of L_A wrt alphabet $A\Sigma$ by finding a backward simulation relation between them and then L_A is a refinement of L_I with respect to alphabet $A\Sigma$ by finding a forward simulation relation between them. Since L_I is deterministic wrt $ACS\Sigma$, if we have an implementation L_C that is a cs-refinement of L_I, we can find a forward simulation between them. As an example, we will pick L_C as time-stamped stack and establish a forward simulation relation between it and our L_I machine.

Let us continue with defining L_I first:

- A state of L_I again consists of a partial strict order and a program counter: $Q_I \subseteq ND \times ED \times (\mathbb{N} \to Lbl_I)$ where $Lbl_I = \{N, A_0, A_1, R_0, R_1, R_2\}$ is the set of transition labels for the operations. Different from the fixed linearization point case, this time nodes are not triples but 5-tuples of the form (k, d, st, mc, con). First three fields are the same as previous intermediate machines: $k \in \mathbb{N}$ is operation identifier of a push operation, $d \in \mathcal{D}$ is the data value of that push and $st \in \{\texttt{PENDING}, \texttt{CLOSED}\}$ is the current status of the push operation. The fourth field $mc \subset \mathbb{N}$ keeps the operation identifiers of pop operations such that this push was maximally closed when the pop began. A node n is maximally closed in a state s iff $n.st = \texttt{CLOSED}$ and if there is an edge $n \to n' \in ed_s$, then $n'.st = \texttt{OPEN}$. The fifth field $con \subset \mathbb{N}$ is the set of operation identifiers of the pop methods that are concurrent with this push i.e. the either this node was open when the pop started or this node is created when the pop was pending.

- The transition labels consist of invocation and return actions of both methods and commit action for only pop method. Hence $\Sigma_I = ACS\Sigma$. Number of parameters for all actions common in previous intermediate stack machine and this intermediate stack machine are the same. Commit actions contain the second data field. They are of the form: $com(pop, d, k)$ where $d \in \mathcal{D}$ and $k \in \mathbb{N}$.

- Initial state consists of an empty strict partial order and a function mapping ever operation to N: $q_{0I} = (\emptyset, \emptyset, f_{q_{0I}})$ where $f_{q_{0I}}(i) = N$ for all $i \in \mathbb{N}$.

- We define δ_I less formally, by giving verbal explanations to the transitions, omitting the obvious updates on the f part and not mentioning about the parts of the nodes that does not change:

  - $(q, inv(push, d, k), q') \in \delta_I$ iff a new node $n = (k, d, \texttt{PENDING}, \emptyset, con_n)$ is added to $nd_{q'}$, where $con_n = \{i \in \mathbb{N} | f_q(i) = R_0\}$; $n' \to n$ will be added to $ed_{q'}$ if n' is a closed node at the state q.
  - $(q, ret(push, k), q') \in \delta_I$ iff either there is a `PENDING` node n in state q and this node becomes `CLOSED` in state q' or there is no node with identifier k in q and nothing else than f field changes in q'.
  - $(q, inv(pop, k), q') \in \delta_I$ iff for every open node $n \in nd_q$, k is added to $n.con$ in q' and for every maximally closed node $m \in nd_q$, k is added to $m.mc$ in state q'.
  - $(q, com(pop, k, d), q') \in \delta_I$ iff there exists a node n in state q such that $n.d = d$ and either $k \in n.con$ or $k \in n.mc$, this node n and all the nodes adjacent to it are removed in the state q', k is removed from all con and mc fields of all nodes in state q' and for all other pop operations k' in $n.mc$ or in $n.con$ and for all states $n' \in nd_q$ such that $n' \to n \in ed_q$ and for all $n'' \in nd_q$, $n' \to n'' \in ed_q$ implies $k' \in n''.con$ (implies $k \notin n''.mc$ but it is stronger than this condition: if there are three closed states p, q, r s.t. $p \to q$, $q \to r$, $p \to r$, k' is only in $r.mc$ and we delete r, former condition only allows $k' \in q.mc$ whereas the latter one allows $k' \in p.mc$ in addition) we have $k' \in n'.mc$ in the state q'. Note that we need to assume data independence to make this action deterministic.
  - $(q, ret(pop, k, d), q') \in \delta_I$ iff $q = q'$ ignoring the f fields.

L_A is the same machine defined in the previous section. The common alphabet between L_A and L_I is $A\Sigma$. We will show that they are equivalent in terms of this alphabet.

Lemma 4. *L_I is a refinement of L_A.*

Proof. We will provide a backward simulation relation bs between states of L_I and states of L_A. Our relation $bs \subseteq Q_I \times Q_A$ relates state $q = (so_q, f_q)$ to $q' = (s_{q'}, f_{q'}$ iff (i) for all operation identifiers $k \in \mathbb{N}$, if $f_q(k) \in \{N, R_1, R_2\}$, then

$f_q(k) = f_{q'}(k)$; if $f_q(k) = A_0$, then $f_{q'}(k) = A_0$ and the data value d associated with this add operation is not inserted to $s_{q'}$ or $f_{q'}(k) = A_1$; if $f_q(k) = A_1$, then $f_{q'}(k) = A_2$; if $f_q(k) = R_0$, then $f_{q'}(k) = R_0$ or $f_{q'}(k) = R_1$; (ii) Let us call a pop operation pending if $f_q(k) = R_0$ and PP_q be set of pending pops. There there exists a function $g : PP_q \to ND_q \cup \{\texttt{NONE}\}$ such that $k \in g(k).mc$ or $k \in g(k).con$ for all pop operation identifiers k such that $g(k) \neq \texttt{NONE}$ and g is one-to-one if we neglect NONE; $s_{q'}$ is obtained by extending so_q to a total order in which pending nodes may not take place and nodes (open or closed) n such that there exists a pop operation with identifier k so that $g(k) = n$ surely do not take place, (iii) $g(k) = \texttt{NONE}$ implies $f_{q'}(k) = R_0$ and $g(k) \neq \texttt{NONE}$ implies $f_{q'}(k) = R_1$, (iv) if $g_{q'}(k) = n$ and n is a pending node, then $f_{q'}(k') = A_1$ where k' is the operation identifier part of n, (v) if there is a pending node with identifier k and it takes part in the linearization, then $f_{q'}(k) = A_1$.

Now, we will show that bs is a backward simulation relation:

$\langle i \rangle$ $bs[q_{0I}] = \{q_{0A}\}$

$\langle ii - a - push \rangle$ Let $(q, inv(push, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We consider two cases: Either the newly added node in q' takes place in the linearization and there exists an index i such that $s_{t'}(i) = d$ or this new node does not exist in the linearization. For the former case construct $s_t = \langle s_{t'}(1), s_{t'}(2), ..., s_{t'}(i-1)\rangle$ and $f_t = f_{t'}$ for all the operations except the ones of which data values are linearized as $s_{t'}(j)$, for $j \geq i$. For those nodes, $f_{t'}(k') = A_1$ whereas we assign $f_t(k') = A_0$ for $j > i$ and $f_t(k) = N$. Let operation identifiers of these nodes be k_j for $j > i$. Then, $t \xrightarrow{\alpha} t'$ holds where $\alpha = inv(push, d, k), lin(push, d, k), lin(push, d_{i+1}, k_{i+1}), ...$ Moreover, $t \in bs[q]$ since s_t is a valid linearization of so_q using the same g function and omitting more open nodes and f_t obeys the conditions. For the latter case, we pick $s_t = s_{t'}$. We have two subcases: There exists a pending pop with identifier k' such that $g(k')$ is the new node with identifier k or not. For the first subcase, we pick $f_t = f_{t'}$ except $f_t(k) = N$ and $f_t(k') = R_0$. Then $txrightarrowinv(push, d, k), lin(pop, d, k')t'$ is a path in L_A and $t \in bs[q]$. For the second subcase, we just pick $f_t = f_{t'}$ except $f_t(k) = N$. Then, it is easy to see that $t \xrightarrow{inv(push,d,k)} t'$ holds and $t \in bs[q]$.

$\langle ii - a - pop \rangle$ Let $(q, inv(pop, k), q') \in \delta_A$ and $t' \in bs[q']$. We will again consider two cases: When relating t' to q', either $g(k) = \texttt{NONE}$ or $g(k)$ is a node in q'. In other words, either the newly invoked pop operation k did not linearize yet or it linearizes and removes an element inserted by a linearized push. The second case also splits into two cases: The element removed by pop k is inserted by a push k' that is still pending or the push has returned. We will look at all three cases separately. The easiest one is the first case. Construct $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = N$ whereas $f_{t'}(k) = R_0$. One can see that $t \xrightarrow{inv(pop,k)} t'$ is a step in L_A and $t \in bs[q]$. For the first case of the second case, we construct $s_t = s_{t'} \circ \langle d \rangle$ where d is

the data of node identifier k' and $f_t = f_{t'}$ except that $f_t(k) = N$ whereas $f_{t'}(k) = R_1$. One can see that $t \xrightarrow{inv(pop,k),lin(pop,d,k)}$ is a valid path in L_A. Moreover, $t \in bs[q]$ since s_t is a valid linearization of so_q. This is true because the node with identifier k' is a maximal node in so_q and we can add it to the end of linearization of $so_{q'}$. For the second subcase, we obtain s_t from $s_{t'}$ by the following procedure: Let n be the node with identifier k' and k'' be the node such that $k' \to k''$ is an edge in $so_{q'}$, k'' takes part in the linearization of $so_{q'}$ to $s_{t'}$ and it has the minimum index i in the $s_{t'}$ among all such nodes. Then, $s_t = \langle s_{t'}(1), s_{t'}(2), ..., s_{t'}(i-1), d\rangle$ where d is the data value of node with identifier k. Let k_j and d_j be the identifiers and data values of nodes that constitute $s_{q'}(j)$ for $j > i$. Clearly, $t \xrightarrow{\alpha} t'$ is a path in L_A where $\alpha = inv(pop,k), lin(pop,d,k), lin(push,d_i,k_i), ...$ Moreover, $t \in bs[q]$ because s_t is a valid linearization of so_q. It is true because $so_q = so_{q'}$, k' is a maximally closed node in so_q and all the nodes with identifier k_j $(j > i)$ are pending nodes.

$\langle ii - c\rangle$ Let $(q, com(pop,d,k), q') \in \delta_I$ and $t' \in bs[q']$. We will consider two cases. The first case is commit action removes a maximally closed or an open node. For these cases, we can construct t as in the second case of the previous item (invoke pop case). The same arguments apply for constructing a path between t and t' in L_A and showing that $t \in bs[q]$. The second case is that commit action removes a non-maximal closed node. This time, pick $t = t'$. Then, $t \xrightarrow{\epsilon} t'$ is the path in L_A and one can show that $t \in bs[q]$ by choosing $g(k)$ as the node that is removed.

$\langle ii - b - push\rangle$ Let $(q, ret(push,k), q') \in \delta_I$ and $t' \in bs[q']$. We consider two cases. Either there is a node n with identifier k and data value d in state q or not. For the first case, we consider two subcases. Either this node takes part in linearization or not (if there exists a pop k' such that $g(k') = n$). For the first subcase, we can pick $s_t = s_{t'}$ and $f_q = f_{q'}$ except that $f_q(k) = A_0$. Then, $t \xrightarrow{ret(push,k)} t'$ is a path in L_A. Moreover, $t \in bs[q]$ since n is a maximal node in q. For the second subcase, we pick $s_t = s_{t'} \circ \langle d\rangle$ and $f_t = f_{t'}$ except that $f_t(k) = A_1$ and $f_t(k') = R_0$. Then, $t \xrightarrow{lin(pop,d,k'),ret(push,k)} t'$ is a path L_A and $t \in bs[q]$ since n is a maximal open node in q. For the second case, we pick $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = A_1$. Then, $t \xrightarrow{ret(push,k)} t'$ is a path in L_A and $t \in bs[q]$.

$\langle ii - b - pop\rangle$ Let $(q, ret(pop,d,k), q') \in \delta_I$ and $t' \in bs[q']$. We pick $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = R_1$. One can see that $t \xrightarrow{ret(pop,d,k)} t'$ is a valid action in L_A and $t \in bs[q]$.

□

Lemma 5. *L_A is a refinement of L_I.*

Proof. We will construct a forward simulation relation fs between L_A and L_I. Our relation $fs \subseteq Q_A \times Q_I$ relates state $q = (s_q, f_q) \in Q_A$ to a state $q' = (so_{q'}, f_{q'}) \in Q_I$ iff (i) for all operation identifiers $k \in \mathbb{N}$, if $f_q(k) \in \{N, A_0, R_0, R_1, R_2\}$ then $f_{q'}(k) = f_q(k)$; if $f_q(k) = A_1$, then $f_{q'}(k) = A_0$; if $f_q(k) = A_2$, then $f_{q'}(k) = A_1$; (ii) We form nodes of $so_{q'}$ ($ND_{q'}$ as follows: If k is a push operation adding data value d and either $f_q(k) = R_0$ or $f_q(k) = R_1$ and data added by this push exists in s_q, then there is a `PENDING` node in $ND_{q'}$ with identifier k and data value d. If $f_q(k) = R_2$, then there is a `CLOSED` node in nd_q with identifier k and data value d. If there is an operation identifier k such that $f_q(k) = R_0$, we call this a pending pop and this pop takes place mc or con fields of nodes of q'. If $n \in ND_{q'}$ is a `PENDING` node, then $k \in n.con$. If n is maximally closed, then $k \in n.con$ or $n.mc$. If $k \in n.mc$ or $k \in n.con$ and there exists another node $n' \in ND_{q'}$ such that $n \to n' \in ED_{q'}$, then $k \in n'.con$. If $k \in n.mc$ and there exists another node $n' \in ND_{q'}$ such that $n' \to n \in ED_{q'}$, then neither $k \in n'.mc$ nor $k \in n'.con$. For any node $n \in ND_{q'}$, either $k \in n.mc$ or $k \in n.con$. (iii) We form edges of $so_{q'}$ ($ED_{q'}$) as follows: `PENDING` nodes are maximal. Edges obey the strict partial order conditions. (iv) We can find a linearization $so_{q'}$ that is equal to s_q. `PENDING` nodes may not participate in the linearization. Note that we do not need g function for keeping track of linearized pops unlike the previous proof.

Next, we will show that fs is a forward simulation relation.

$\langle i \rangle$ $fs[q_{0A}] = \{q_{0I}\}$

$\langle ii - a - push \rangle$ Let $(q, inv(push, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, inv(push, d, k), t') \in \delta_I$. Since $s_q = s_{q'}$ and $so_{t'}$ contains a maximal open new node as the only difference from so_t, we can linearize $so_{t'}$ so that linearization is equal to $s_{q'}$. By checking the other conditions, one can observe that $t' \in fs[q']$.

$\langle ii - a - pop \rangle$ Let $(q, inv(pop, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, inv(pop, k), t') \in \delta_I$. Only difference between t and t' is that maximally closed and open nodes in t' contain k in their mc or con fields. Since this new addition obeys our forward simulation definition, $t' \in fs[q']$.

$\langle ii - c - push \rangle$ Let $(q, lin(push, d, k)q') \in \delta_A$ and $t \in fs[q]$. Pick $t' = t$. $s_{q'}$ is still a linearization of so_t since the node with identifier k is a maximal node in ND_t and we can linearize it at the end. So, $t' \in fs[q']$ holds.

$\langle ii - c - pop \rangle$ Let $(q, lin(pop, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, com(pop, d, k)t') \in \delta_I$. The action $com(pop, d, k)$ is a valid action in L_I because d is the last element in s_q. Hence, the node n with identifier k and data value d is a maximal element in so_q and either $k \in n.mc$ or $k \in n.con$ by the properties of fs. Hence, the node with identifier k can be removed by a com action. In addition, $s_{q'}$ is a linearization of $so_{t'}$ because removed node is a maximal node and $s_{q'}$ is obtained from s_q by deleting the maximum node. Hence, $t' \in fs[q']$ holds.

$\langle ii - b - push \rangle$ Let $(q, ret(push, k), q') \in \delta_A$ and $t \in fs[q]$. We will consider two cases: either the element inserted by the push with identifier k is removed by a concurrent pop or the element is still in s_q. For the former case, there is no node with identifier k in state t and we can pick t' such that $(t, ret(push, k), t') \in \delta_I$. We have $so_t = so_{t'}$ for this case. Since $s_q = s_{q'}$ also holds, $s_{t'}$ is a linearization of $so_{q'}$ and $t' \in fs[q']$. For the latter case, we again pick t' such that $(t, ret(push, k), t') \in \delta_I$ holds. This time, only difference between so_t and $so_{t'}$ is that the node with identifier k is `CLOSED` in $so_{t'}$. Since the edges are the same, $s_{q'}$ is a valid linearization of $s_{t'}$ and $t' \in fs[q']$ holds.

$langleii - b - pop\rangle$ Let $(q, ret(pop, d, k), q') \in \delta_A$ and $t \in fs[q]$. We pick t' such that $(t, ret(pop, d, k)t') \in \delta_I$. Only difference between t and t' is that $f_{t'}(k) = R_2$ whereas $f_t(k) = R_1$. Since $s_q = s_{q'}$ and $so_t = so_{t'}$, $s_{q'}$ is a valid linearization of $so_{t'}$. By checking the other conditions, we see that $t' \in fs[q']$.

□

Now, we show that TS-Stack is linearizable by showing the concrete TS-Stack implementation L_C is a cs-refinement of L_I. As L_C, we pick the simplest version that omits the `EMPTY` returns of the pop methods, does not allow unlinking and elimination.

5 Existence of Forward Simulations for Set Implementations That Have Fixed Remove Linearization Points

For all the set libraries we fix $\mathcal{M} = \{add, rmv, cnt\}$ where rmv is short for remove and cnt is short for contains methods and $\mathcal{D} = \{1, \texttt{TRUE}, \texttt{FALSE}\}$. We assume that only single element can be inserted into our list. Our results for this domain extends to other domains such as when $\mathcal{D} = \mathbb{N}$. We extend the definition of $A\Sigma$ introduced in Definition 2 for the set in our focus as $AS\Sigma = A\Sigma \cup \{lin(rmv, d, k) | d \in \{1\}, k \in \mathbb{N}\}$. We define s-refinement, s-linearizability and change the definitions of forward and backward simulations as in the beginning of Section 2.

We define L_A as follows:

- $Q_A = 2^{\{1\}} \times (\mathbb{N} \to Lbl_A)$ where $Lbl_A = \{N, A_0, A_{1T}, A_{1F}A_2, R_0, R_{1T}, R_{1F}, R_2, C_0, C_{1T}, C_{1F}, C_2\}$. For each $q \in Q_A$, s_q represents the set component (first component), and f_q represents the program counter (second component).
- Transition labels consists of invocations, returns and linearizations of methods in $\mathcal{M}$: $\Sigma_A : AS\Sigma \cup \{lin(m, d, k) | m \in \{add, cnt\}, d \in \{1\}, k \in \mathbb{N}\}$. All methods return `TRUE` or `FALSE` and all take an element in $\{1\}$ as the input value.
- $q_{0A} = (\emptyset, f_{q_{0A}}$ where $f_{q_{0A}}(k) = N$ for all $k \in \mathbb{N}$.

- State transitions:
  - $(q, inv(add, d, k), q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = N \wedge f_{q'}(k) = A_0$
  - $(q, lin(add, d, k), q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = A_0 \wedge (d \in s_q \wedge s_q = s_{q'} \wedge f_{q'}(k) = A_{1F} \vee d \notin s_q \wedge s_{q'} = s_q \cup \{d\} \wedge f_{q'}(k) = A_{1T})$
  - $(q, ret(add, d, k)q') \in delta_A)$ iff $(f_q(k) = A_{1T} \wedge d = \texttt{TRUE} \vee f_q(k) = A_{1F} \wedge d = \texttt{FALSE}) \wedge f_{q'}(k) = A_2$
  - $(q, inv(rmv, d, k)q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = N \wedge f_{q'}(k) = R_0$
  - $(q, lin(rmv, d, k), q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = R_0 \wedge (d \in s_q \wedge s_q = s_{q'} \cup \{d\} \wedge f_{q'}(k) = R_{1T} \vee d \notin s_q \wedge s_q = s_{q'} \wedge f_{q'}(k) = R_{1F})$
  - $(q, ret(rmv, d, k)q') \in delta_A)$ iff $(f_q(k) = R_{1T} \wedge d = \texttt{TRUE} \vee f_q(k) = R_{1F} \wedge d = \texttt{FALSE}) \wedge f_{q'}(k) = R_2$
  - $(q, inv(cnt, d, k)q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = N \wedge f_{q'}(k) = C_0$
  - $(q, lin(cnt, d, k), q') \in \delta_A$ iff $d \in \{1\} \wedge f_q(k) = C_0 \wedge s_q = s_{q'} \wedge (d \in s_q \wedge f_{q'}(k) = C_{1T} \vee d \notin s_q \wedge f_{q'}(k) = C_{1F})$
  - $(q, ret(cnt, d, k)q') \in delta_A)$ iff $(f_q(k) = C_{1T} \wedge d = \texttt{TRUE} \vee f_q(k) = C_{1F} \wedge d = \texttt{FALSE}) \wedge f_{q'}(k) = C_2$

We define L_I as follows:

- A state $q \in Q_I$ is a tuple of the form $(sac_q, src_q, UBSA_q, LBSA_q, CC_q, ubi, lbi, f_q)$ where $sac_q \in \mathbb{N}$ keeps the number of adds that return true so far, $src_q \in \mathbb{N}$ keeps the number of successful linearizations of remove (ones that move program counter from R_0 to R_{1T}), $f : \mathbb{N} \to Lbl_I$ is program counter that maps every operation to a label in $Lbl_I = \{N, A_0, A_1, R_0, R_{1T}, R_{1F}, R_2, C_0, C_1\}$. Let $PA_q = \{k \in \mathbb{N} | f_q(k) = A_0\}$ be the set of pending adds at state q. Then, $UBSA_q, LBSA_q : \mathbb{N} \to 2^{PE_q}$ are functions such that $UBSA_q(i)$ is a set of pending adds at most i of which may return true and $LBSA_q(i)$ is a set of pending adds at least i of which may return true. $ubi, lbi \in \mathbb{N}$ keeps the upper bound (lower bound) set indices that a new pending enqueue will be inserted to. Let $IC_q = \{k \in \mathbb{N} | f_q(k) = C_0 \vee f_q(k) = C_1$ be the set of contains operation identifiers. Then, $CC_q : IC_q \to 2^{PA}$ is a map that keeps a set of pending adds of which at least one should return true due to a true return of a contains operation.

- Transition labels are exactly the abstract transition labels we have defined earlier: $\Sigma_I = AS\Sigma$

- $q_{0_I} = (sac_{q_{0_I}}, src_{q_{0_I}}, UBSA_{q_{0_I}}, LBSA_{q_{0_I}}, CC_{q_{0_I}}, ubi_{q_{0_I}}, lbi_{q_{0_I}}, f_{q_{0_I}})$ where $sac_{q_{0_I}} = src_{q_{0_I}} = 0$, $lbi_{q_{0_I}} = ubi_{q_{0_I}} = 1$ and $f_{q_{0_I}}(k) = N$ for all $k \in \mathbb{N}$. Hence $PA_{q_{0_I}} = IC_{q_{0_I}} = \emptyset$ and $UBSA_{q_{0_I}}$, $LBSA_{q_{0_I}}$ and $CC_{q_{0_I}}$ are empty mappings.

- Instead of giving state transition relation formally, I would like to explain how $UBSA$, $LBSA$ and CC works by considering the possible state transformations (consider q as pre-state and q' as the post state as convention for the following):

$UBSA$ Initially, $ubi_{q_{0_I}} = 1$. Hence, a newly invoked add operation a will be inserted into $UBSA_{q_{0_I}}(1)$. All the newly invoked operations are inserted to $UBSA_q(1)$ until a linearization of remove comes or one of the adds return successful. If the first case happens, we increment the index of every set by 1 i.e. $UBSA_{q'}(i) = UBSA_q(i-1)$ for all $i > 0$ and $UBSA_{q'}(0) = \emptyset$. We also set $ubi_{q'} = 1$, if it was set to 0 somehow before (Consider the trace $inv(add, a, 1), ret(add, true, 1), inv(add, a, 2), lin(rmv, a, 3)$. $ubi = 0$ after second event and it needs to be incremented to 1 after the fourth event). Next, consider the case of successful add. Let the operation identifier of the add that will return successful be k and i be the index such that $k \in UBSA_q(i)$. Then, for all $j > i$, $UBSA_{q'}(j-1) = UBSA_q(j)$, $UBSA_{q'}(i-1) = UBSA_q(i-1) \cup UBSA_q(i) \backslash \{k\}$ and for all $j < i-1$, $UBSA_{q'}(j) = UBSA_q(j)$. We do not allow elements in $UBSA_q(0)$ to return true. So, $i = 0$ cannot be true. If $i = 1$, we change $ubi_{q'} = 0$ to denote that no new coming add can return true from now on (consider the case: $inv(add, a, 1), lin(rmv, a, 2) : true, inv(add, a, 3), inv(add, a, 4), lin(rmv, a, 5) : true, ret(add, true, 3), ret(add, true, 4), inv(add, a,$ This last invocation should be added to set with index 0. So, we need to change ubi to 0 after last return. Lastly, a failing return of add simply removes this add from its set, without changing its index. Let i be the index of the failing add operation k. Then, $UBSA_{q'}(i) = UBSA_q(i) \backslash \{k\}$. Observing a failing contains operation may change the upper bound. Consider the following trace: $inv(add, a, 1), inv(add, a, 2), inv(cnt, a, 3), ret(cnt, false, 3)$. Although $UBSA_{q'}(1) = \{1, 2\}$ after the last event, none of 1 or 2 can return true due to the restrictions we impose on contains operations that we will explain later on.

$LBSA$ Initially $lbi_{q_{0_I}} = 1$. When a new event of type invoke add comes, we modify $LBSA_{q'}(lbi_{q'}) = PE_{q'}$ where $lbi_{q'} = lbi_q$. Although it looks like that we build $LBSA_{q'}$ from scratch with every new add invocation, in practice, this is merely adding new pending add to the old set of the same index unless the lbi field are the same. If a new successful linearization of remove comes, first it updates $LBSA_{q'}(lbi_q) = PE_{q'}$. Then, it increments the lbi value by one if it was not 0 before. If it was 0 before, new lbi value becomes $src_{q'} - sac_{q'} + 1$. Note that a successful remove does not modify $LBSA$ field if a new add is invoked after the previous linearization of a successful remove. The first new add invocation that comes after a successful remove, copies the set $LBSA_q(lbi_q)$ and adds the new add's operation identifier as $LBSA_{q'}(lbi_{q'})$. However, if no new add comes between two successful remove linearizations, then $LBSA$ is modified by a sucsessful remove linearization. Next, consider the return events of add operations. First, consider the successful return. If an add operation with identifier k returns true, we remove this identifier from the $LBSA$ sets of which k is element of and decrement the index values of these sets

by 1. We can observe that if k is the element of the set with index i then it is element of every set with index $j > i$ (We need to prove this). Hence we decrement the index of every set bigger than some i value. For this case, we may end up with a situation that two sets fall into same index (at index $i-1$). In this case, one of the sets must be strictly subset of the other one (We need to prove this). In this case, we keep the subset as the set of this index and delete the larger set. If k is deleted from just no sets, we change $lbi_{q'} = 0$. To rationalize this behavior, consider the trace $inv(add, a, 1), lin(rmv, a, 2) : true, inv(add, a, 3), inv(add, a, 4), lin(rmv, a, 5) : true, ret(add, true, 3), ret(add, true, 4), inv(add, a,$ The last add invocation should be put to the set with index 0 and after the add with identifier 4 returns true, our procedure makes $lbi_{q'} = 0$. If the removed element is in $LBSA_q(lbi_q) \backslash LBSA_q(lbi_q - 1)$, then we also set $lbi_{q'} = 0$. The rationale behind this is the trace: $inv(add, a, 1), lin(rmv, a, 2) : true, inv(add, a, 3), ret(add, true, 3), inv(add, a, 4)$. The last operation with ID 4 should be able to return false. If an add operation with identifier k returns false, then we remove k from all $LBSA$ sets that k is element of. The constraint we check on $LBSA$ sets is that a return false event of an add operation keeps $|LBSA_{q'}(i)| \geq i$ for the indices i it modified.

CC We need to keep a cc flag to show that now this operation may return true. It should be a map from identifiers to boolean.