Refinement Proofs and Techniques

Suha Orhun Mutluergil

Koc University, Istanbul, Turkey

Refinement

- In general: A concrete and complex system S_1 refines the abstract system S_2 iff S_2 completely captures the behaviors of S_1.
- For automata/state machines/transition systems
  - Refinement is based on observable actions alphabet Σ.
  - Formally: A Labeled Transition System (LTS) L_1 $\Sigma-$refines the LTS L_2 iff for every trace τ of L_1, there exists a trace τ' of L_2 such that $\tau|_\Sigma = \tau'|_\Sigma$.
- Example: $\mathrm{L}_1 = \langle S_1 = \{s_0, s_1, \dots\}, \Sigma_1 = \{\alpha, \beta, \gamma\}, \delta_1 \rangle$ s.t. $\delta_1 \subseteq S_1 \times \Sigma_1 \times \mathrm{S}_1$

  $L_2 = \langle S_2 = \{q_0, q_1, \dots\}, \Sigma_2 = \{\alpha, \beta, \theta\}, \delta_2 \rangle$ s.t. $\delta_2 \subseteq S_2 \times \Sigma_2 \times S_2$

  $\Sigma = \{\alpha, \beta\}$

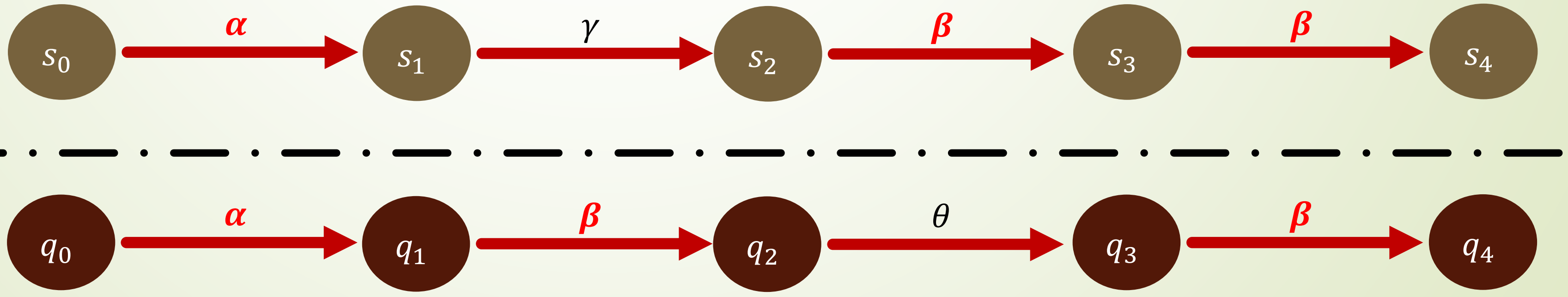
$$\begin{tikzcd}
s_0 \arrow[r, "\alpha"] & s_1 \arrow[r, "\gamma"] & s_2 \arrow[r, "\beta"] & s_3 \arrow[r, "\beta"] & s_4 \\
q_0 \arrow[r, "\alpha"] & q_1 \arrow[r, "\beta"] & q_2 \arrow[r, "\theta"] & q_3 \arrow[r, "\beta"] & q_4
\end{tikzcd}$$

Refinement

- In general: A concrete and complex system S_1 refines the abstract system S_2 iff S_2 completely captures the behaviors of S_1.
- For automata/state machines/transition systems
  - Refinement is based on observable actions alphabet Σ.
  - Formally: A Labeled Transition System (LTS) L_1 Σ−refines the LTS L_2 iff for every trace τ of L_1, there exists a trace τ' of L_2 such that $\tau|_\Sigma = \tau'|_\Sigma$.
- Example: $\mathrm{L}_1 = \langle S_1 = \{s_0, s_1, \dots\}, \Sigma_1 = \{\alpha, \beta, \gamma\}, \delta_1 \rangle$ s.t. $\delta_1 \subseteq S_1 \times \Sigma_1 \times \mathrm{S}_1$

  $L_2 = \langle S_2 = \{q_0, q_1, \dots\}, \Sigma_2 = \{\alpha, \beta, \theta\}, \delta_2 \rangle$ s.t. $\delta_2 \subseteq S_2 \times \Sigma_2 \times S_2$

  $\Sigma = \{\alpha, \beta\}$


$$\begin{tikzcd} s_0 \arrow[r, "\boldsymbol{\alpha}"] & s_1 \arrow[r, "\gamma"] & s_2 \arrow[r, "\boldsymbol{\beta}"] & s_3 \arrow[r, "\boldsymbol{\beta}"] & s_4 \end{tikzcd}$$

$$\begin{tikzcd} q_0 \arrow[r, "\boldsymbol{\alpha}"] & q_1 \arrow[r, "\boldsymbol{\beta}"] & q_2 \arrow[r, "\theta"] & q_3 \arrow[r, "\boldsymbol{\beta}"] & q_4 \end{tikzcd}$$

How to Prove Refinement

- In general, proofs depend on finding a particular kind of relations/functions that relates states of L_1 to states of L_2.
  - Refinement mappings, forward simulation relations, backward simulation relations
- Completeness issues: None of these relations/functions are complete.
- Refinement Mappings
  - Complete if L_1 is a forest and L_2 is deterministic.[2]
  - Otherwise, history and/or prophecy variables may need to be added.[1]
- Forward Simulations
  - Complete if L_2 is deterministic.[2]
  - Otherwise, prophecy variables may need to be added.[2]
- Backward Simulations
  - Complete if L_1 is a forest.[2]
  - Otherwise, history variables may need to be added.[2]

1. Abadi, M., & Lamport, L. (1991). The existence of refinement mappings. Theoretical Computer Science, 82(2), 253-284.
2. Lynch, N. A., & Vaandrager, F. W. (1995). Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, *121*(2), 214-233.

Proving Linearizability using Forward Simulations[‡]

Joint work with Ahmed Bouajjani[1], Constantin Enea[1] and Michael Emmi[2]

[1]:IRIF, University of Paris Diderot

[2]:Nokia Bell Labs

‡: To Appear in CAV'17

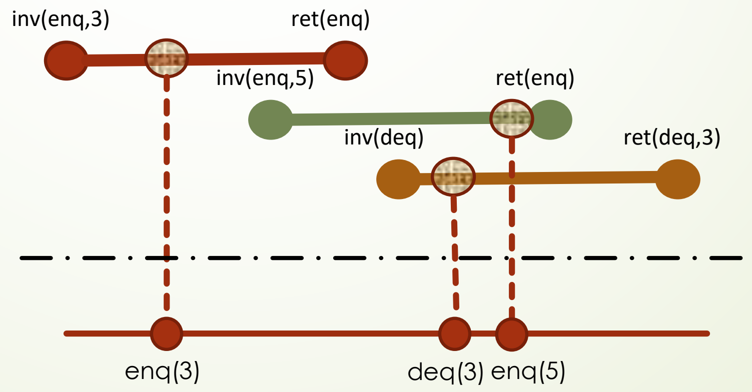
A Brief Overview

- Scope: Proving correctness of concurrent stack and queue implementations (which eventually boils down to a refinement proof).
- Contributions: A new stack and queue LTS specifications that are more useful than the standard specifications for the proofs
  - Shown the equivalence to the standard specifications
  - Existence of forward simulations is guaranteed if some properties are known for the dequeue/pop methods of the implementations.
- Experiments/Applications
  - Shown the correctness of Herlihy-Wing Queue[1] by finding a forward simulation relation to the new queue implementation.
  - Shown correctness of Time-Stamped Stack[2] finding a forward simulation relation to the new stack implementation.

1. Herlihy, Maurice P., and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.3 (1990): 463-492.
2. Dodds, Mike, Andreas Haas, and Christoph M. Kirsch. "A scalable, correct time-stamped stack." ACM SIGPLAN Notices. Vol. 50. No. 1. ACM, 2015.

Linearizability

- The standard correctness condition for concurrent data structures/libraries.
- Call and return actions mark the beginning and end of methods.
- History: Projection of a trace over call and return actions ($C \cup R$).
- L_1 is linearizable with respect to the specification L_2 iff there exists a linearization point of every operation in the history h_1 btw its call and return points such that the same operation of L_2 takes place atomically at that point.

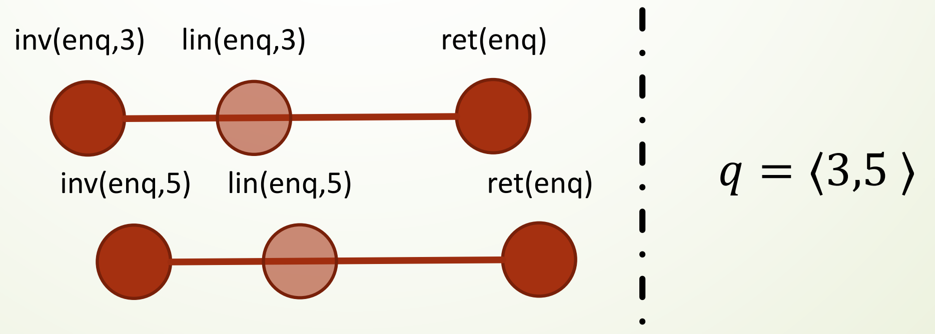


How to Prove Linearizability of Queues

- Standard abstract specification $AbsQ_0$:
  - State: $q\colon \mathbb{N}^{\omega}$
  - Actions: $inv(enq, d)$, $lin(enq, d)$, $ret(enq)$, $inv(deq)$, $lin(deq, d)$, $ret(deq, d)$
  - $lin(enq, d) \; \coloneqq q' = q \circ \langle d \rangle$
  - $lin(deq, \mathrm{EMPTY}) \; \coloneqq q = \langle\, \rangle \Rightarrow \mathrm{q}' = \mathrm{q}$
  - $lin(deq, d) \coloneqq q = \langle d \rangle \circ s \wedge d \neq \mathrm{EMPTY} \Rightarrow q' = s$
- Showing that the implementation L_1 $C \cup R$ –refines $AbsQ_0$ is sufficient.
- If we know the linearization points of enqueue or dequeue methods, finding $C \cup R \cup lin$ –refinements are easier.

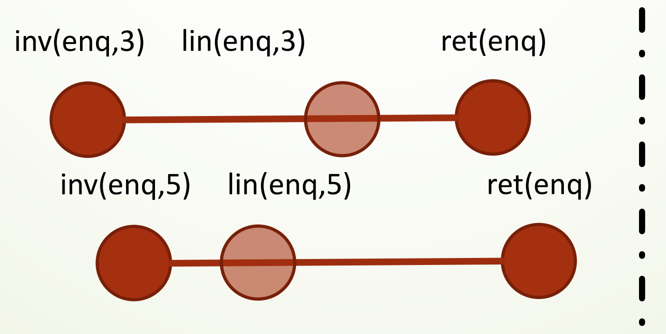
Observations about Implementations

- Linearization points of enqueues are usually not fixed (depends on the execution).
- Linearization points of dequeues are usually fixed and easy to determine.
- $AbsQ_0$ is not deterministic in terms of $C \cup R$ and $C \cup R \cup lin(deq)$.



Observations about Implementations

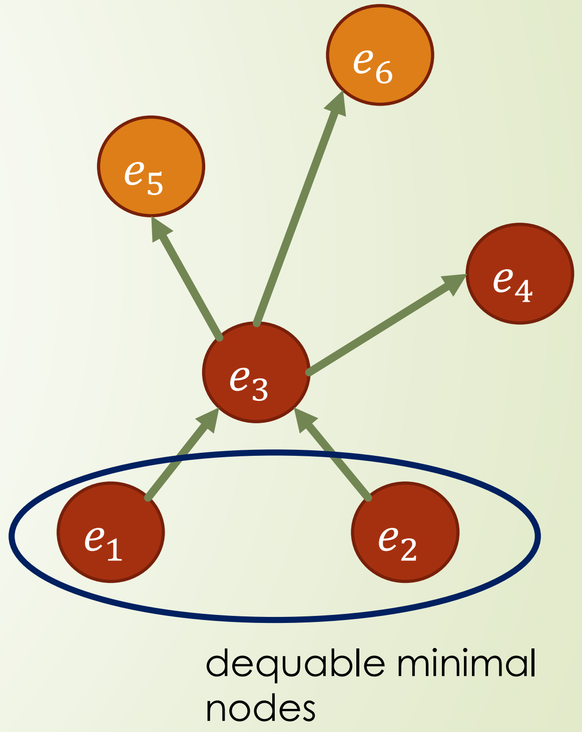
- Linearization points of enqueues are usually not fixed (depends on the execution).
- Linearization points of dequeues are usually fixed and easy to determine.
- $AbsQ_0$ is not deterministic in terms of $C \cup R$ and $C \cup R \cup lin(deq)$.



$q = \langle 5,3 \rangle$

New Abstract Queue $AbsQ$

- States: Strict partial order of enqueue operations based on happens-before relation. They can be pending or completed
- Actions: $C \cup R \cup lin(deq)$
- $AbsQ$ is deterministic in terms of $C \cup R \cup lin(deq)$
- $AbsQ$ produces same histories with $AbsQ_0$.
- Example Application: Showing linearizability of Herlihy & Wing Queue[1] by finding a forward simulation to $AbsQ$.



● :COMPLETED ● :PENDING

1. Herlihy, Maurice P., and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.3 (1990): 463-492.

The Stack Case

- A natural conversion of $AbsQ$ to $AbsS$ exists. Pops remove maximal elements instead of minimal elements.
- Similar observations on implementations: linearization points of pushes are not fixed. For complicated examples, linearization points of pops are not fixed neither. But, we can determine commit points (that fixes the return value) of pops.
- $AbsS_0$ is not deterministic in terms of $C \cup R$ or $C \cup R \cup com(pop)$.
- We introduce a new $AbsS$ that produces different from the dual of $AbsQ$, equivalent executions with $AbsS_0$ and deterministic in terms of $C \cup R \cup com(pop)$.
- We have shown its applicability by finding a forward simulation from the complicated Time-Stamped Stack[1] implementation to $AbsS$.

1. Dodds, Mike, Andreas Haas, and Christoph M. Kirsch. "A scalable, correct time-stamped stack." ACM SIGPLAN Notices. Vol. 50. No. 1. ACM, 2015.

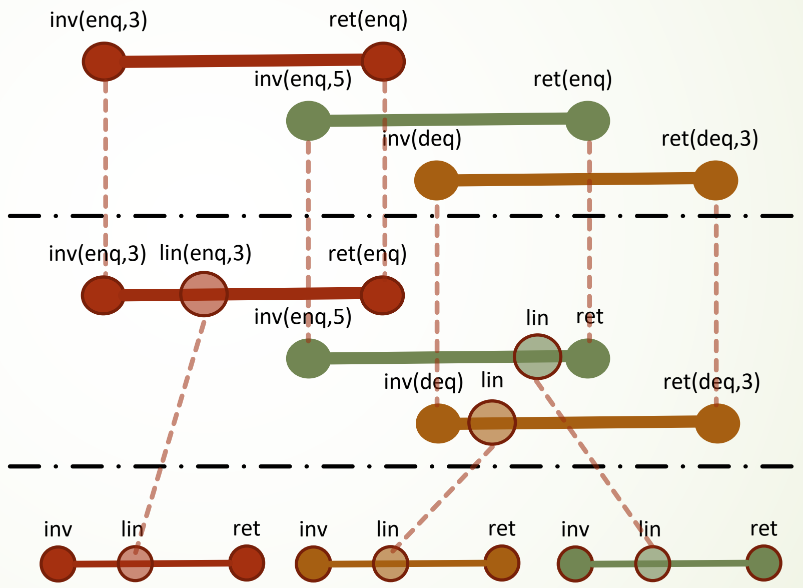
Conclusions & Other Interests

- Future work: Extending the idea to other data structures like sets.
- Future work: Mechanizing the proofs on Boogie/CIVL proof system developed by Microsoft Research and Koc University.
- Other interests:
  - Refinement proofs for weak memory models.
  - Particularly, extending the CIVL proof system for TSO memory model.
  - New proof rules for TSO.
  - Extending the concept of linearizability for WMM.

Thank You

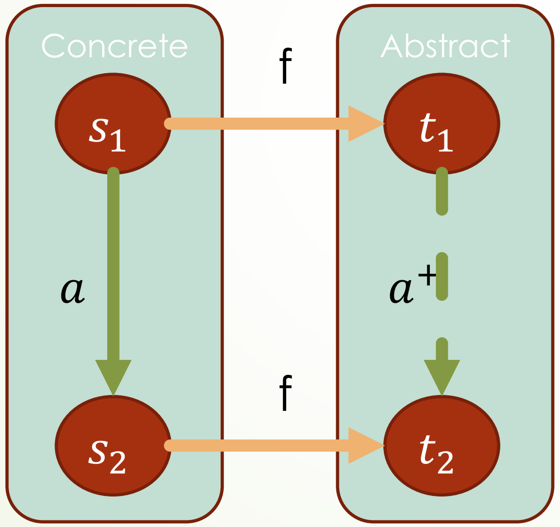
- Any Questions?

How to Prove Linearizability



How to Prove Refinement -1

- Refinement Mappings: $f: Q_C \rightarrow Q_A$
- Initial: $s \in Init(L_C) \Rightarrow f(s) \in Init(L_A)$
- Step:


$$\begin{tikzcd} s_1 \arrow[r, "f"] \arrow[d, "a"'] & t_1 \arrow[d, dashed, "a^+"'] \\ s_2 \arrow[r, "f"] & t_2 \end{tikzcd}$$

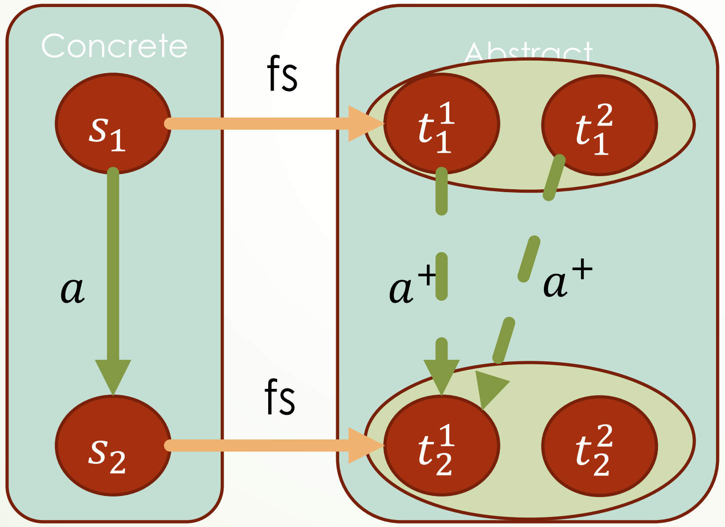
Let $\Sigma \subseteq \Sigma_C, \Sigma_A$ be the refinement alphabet.
If $a \in \Sigma$, then, $a^+ \in (\Sigma_A \setminus \Sigma)^* a (\Sigma_A \setminus \Sigma)^*$.
If $a \notin \Sigma$, then $a^+ \in (\Sigma_A \setminus \Sigma)^*$.

- Complete if L_c is a forest and L_A is deterministic.[2]
- History and/or Prophecy variables may be needed to be added to find a ref. map.[1]

1. Abadi, M., & Lamport, L. (1991). The existence of refinement mappings. Theoretical Computer Science, 82(2), 253-284.
2. Lynch, N. A., & Vaandrager, F. W. (1995). Forward and backward simulations. Part I: Untimed systems. *Information and Computation*, *121*(2), 214-233.

How to Prove Refinement -2

- Forward Simulation Relations: $\text{fs} \subseteq Q_C \times Q_A$
- Initial: $s \in Init(L_C) \Rightarrow fs[s] \cap Init(L_A) \neq \emptyset$
- Step:

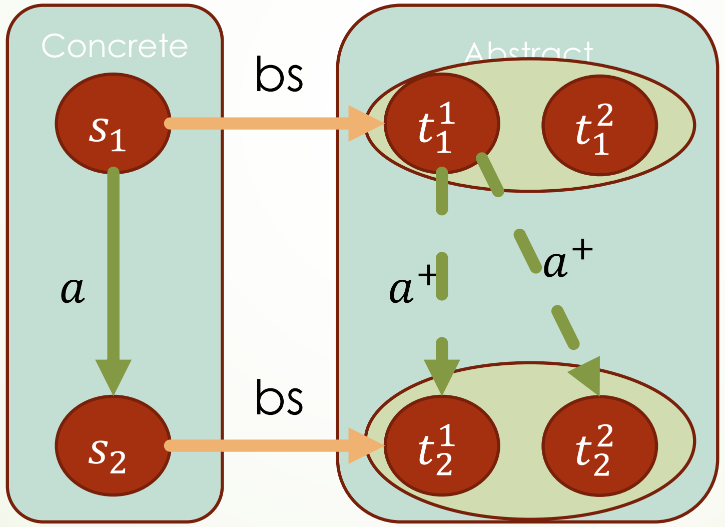


- Complete if L_A is deterministic.[1]
- Prophecy variables may be needed to be added to find a frw. sim. rln.[1]

1. Lynch, N. A., & Vaandrager, F. W. (1995). Forward and backward simulations. Part I: Untimed systems. Information and Computation, 121(2), 214-233.

How to Prove Refinement -3

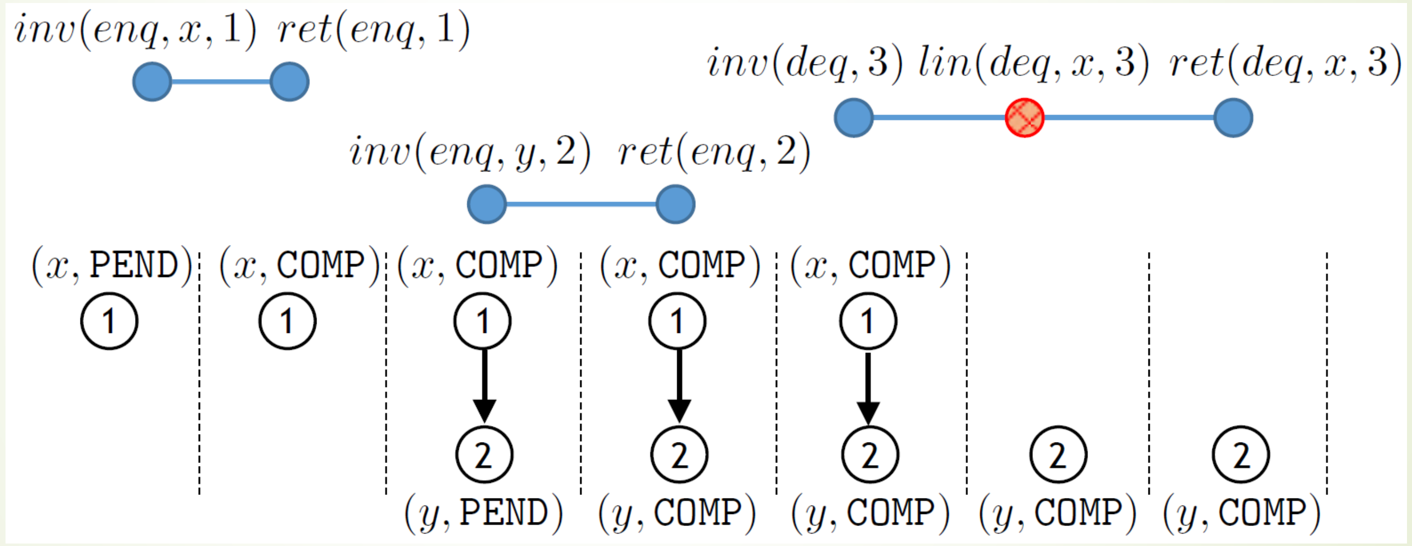
- Backward Simulation Relations: $\text{bs} \subseteq Q_C \times Q_A$
- Initial: $s \in Init(L_C) \Rightarrow bs[s] \subseteq Init(L_A)$
- Step:



- Complete if L_c is a forest.[1]
- History variables may be needed to be added to find a bck. sim. rln.[1]

1. Lynch, N. A., & Vaandrager, F. W. (1995). Forward and backward simulations. Part I: Untimed systems. Information and Computation, 121(2), 214-233.

New Abstract Queue *AbsQ*

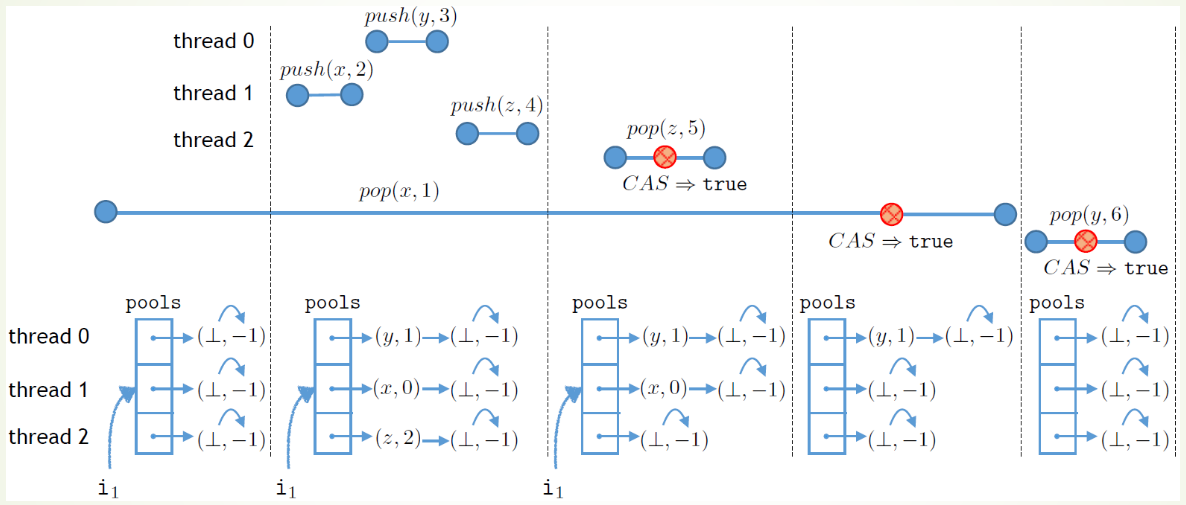


Results on $AbsQ$

- $AbsQ$ is a $C \cup R \cup lin(deq)$-refinement of $AbsQ_0$.
- $AbsQ_0$ is a $C \cup R \cup lin(deq)$-refinement of $AbsQ$.
- $AbsQ$ is deterministic in terms of $C \cup R \cup lin(deq)$.
- If L_C is a queue implementation for which linearization or commit points of dequeue is known and fixed, we can find a forward simulation relation from L_C to $AbsQ$.
  - Example: Herlihy-Wing Queue[1]

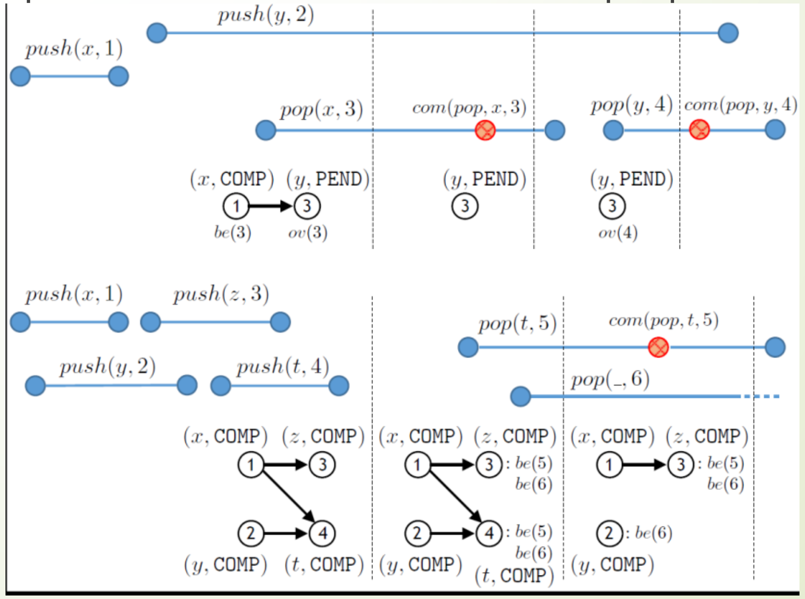
1. Herlihy, Maurice P., and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.3 (1990): 463-492.

What about Stacks?



AbsS Extensions

- Keep track of pushes that can be removed by a pop:
  - Nodes that are pending or maximally closed when the pop started (initialize be and ov sets)
  - Pushes that overlap with the pop (extend ov set)
  - Nodes that become maximal while the pop was executing (update be set)
- NOTE: New *AbsS* keeps working for implementations with fixed pop linearization points.
- How it actually works:



Results on $AbsS$

- $AbsS$ is a $C \cup R$-refinement of $AbsS_0$.
- $AbsS_0$ is a $C \cup R$-refinement of $AbsS$.
- $AbsS$ is deterministic in terms of $C \cup R \cup com(pop)$.
- If L_C is a stack implementation for which linearization or commit points of pop is known and fixed, we can find a forward simulation relation from L_C to $AbsS$.
  - Example: Time-Stamped Stack[1]

1. Dodds, Mike, Andreas Haas, and Christoph M. Kirsch. "A scalable, correct time-stamped stack." ACM SIGPLAN Notices. Vol. 50. No. 1. ACM, 2015.