State Transition System alternative to Linearizability

Ilya Sergey

joint work (in progress) with
Aleks Nanevski, Anindya Banerjee,
Ruy Ley-Wild and Germán Delbianco
Linearizability

- Golden standard for canonical specifications
- A tool for granularity abstraction
Canonical Specifications

\{ S = xs \} \text{push}(x) \{ S = x :: xs \}

\{ S = xs \} \text{pop()} \{ \begin{align*}
\text{res} &= \text{Nothing} \land S = \text{Nil} \\
\lor \exists x, xs. \text{res} &= \text{Just}(x) \land S = x :: xs \land S' = xs
\end{align*} \}

Suitable for sequential case
Canonical Specifications

\[
\{ S = xs \} \ push(x) \ { S = x :: xs \}
\]

\[
\{ S = xs \} \ pop() \ { \text{res} = \text{Nothing} \land S = \text{Nil} \\
\lor \exists x, xs. \text{res} = \text{Just}(x) \land S = x :: xs \land \\
S' = xs \}
\]

Bad for concurrent use:
not stable under interference
Stable Concurrent Specifications

\[ \forall P : \text{Elem} \rightarrow \text{Prop}. \]
\[ \{ P(x) \} \quad \text{push}(x) \quad \{ \text{true} \} \]
\[ \{ \text{true} \} \quad \text{pop}(x) \quad \{ \begin{array}{l} \text{res} = \text{Nothing} \\
\lor \exists x. \text{res} = \text{Just}(x) \land P(x) \end{array} \} \]

Not a canonical spec: the same one holds for queues, sets, bags

Svendsen-al:ESOP13
Turon-al:ICFP13
Making things worse

\[ \forall P : \text{Elem} \rightarrow \text{Prop}. \]

\[ \{ P(x) \} \quad \text{push}(x) \quad \{ \text{true} \} \]

\[ \{ \text{true} \} \quad \text{pop}() \quad \{ \text{res} = \text{Nothing} \]

\[ \lor \exists x. \text{res} = \text{Just}(x) \land P(x) \} \]

\[ \{ P(x) \} \quad \text{contains}(x) \quad \{ \text{res} = \ldots \} \]
Linearizability to the rescue

canonical spec = sequential spec *

\[
\{ S = xs \} \; \text{push}(x) \; \rightsquigarrow \; \{ S = x :: xs \}
\]

\[
\{ S = xs \} \; \text{pop}() \; \rightsquigarrow \; \{ \text{res} = \text{Nothing} \land S = \text{Nil} \land
\exists x, xs. \; \text{res} = \text{Just}(x) \land S = x :: xs \land S' = xs \}
\]

\[
\{ S = xs \} \; \text{contains}(x) \; \rightsquigarrow \; \{ \text{res} = (x \in xs) \land S' = xs \}
\]

* or atomic operations with the sequential spec above
Can we provide a convenient *concurrent* specification for `contains()` without appealing to linearizability?

(probably, it will also be more straightforward to prove)
Reasoning with hindsight

contains(x) = true  
x was in the contents of the stack S at some moment before or during the execution of contains()

contains(x) = false  
x was not in the contents S at some moment before or during the execution of contains()

Hindsight is a property of a resource’s past history
Formalising the idea of *hindsight* for a large class of concurrent protocols.
A model for resources with histories

- Resources represented by State-Transition Systems (STS)
- Transitions define Rely/Guarantee of a resource
- Auxiliaries are ghost parts of the resource’s state

Concurroids — Subjective STSs

- Self - (possibly ghost) resources owned by me
- Other - (possibly ghost) resources owned by all others
- Shared - resources owned by the protocol module
- Self and Other are elements of a Partial Commutative Monoid (PCM): (S, 0, ⊕).
Specifications with Concurroids

\[ C = \text{\{ p \}} \quad \text{c} \quad \text{\{ q \}} \quad \text{\@} \quad C \]

defines Rely/Guarantee and RI
A model for resources with histories

- Resources represented by *State-Transition Systems (STS)*
- Transitions define *Rely/Guarantee* of a resource
- **Auxiliaries are ghost parts of the resource’s state**

A model for resources with histories

- Resources represented by State-Transition Systems (STS)
- Transitions define Rely/Guarantee of a resource
- Auxiliaries are ghost parts of the resource’s state
- Histories are a particular case of ghosts
Capturing histories with timestamps

... $\ast t_i$

per-resource shared timestamp counter
time increased at every change in “visible” state
We will record only interesting projections of the shared state.
• $H_s, H_o$ — *self/other* contributions to the protocol history

• *Timestamped histories* form a PCM $\Rightarrow$ can be split
Reasoning about pair snapshots

Qadeer-al:TR09,Liang-Feng:PLDI13

Atomically update and increase the version

```plaintext
write_x(v) { <x := (x.v, x.s++)> }
write_y(v) { <y := (y.v, y.s++)> }

letrec read_pair(): (Val, Val) = {
  (v, s) <- <read_x()>;
  (w, _) <- <read_y()>;
  if (s == <read_x>().s) then (v, w);
  else read_pair();
}
```

Atomically read each component
If x wasn’t changed until this moment, then return a snapshot, else try again.
The Pair snapshot concurroid is defined by the function $F_{ps}$.

$$F_{ps} = H_s \cup H_o$$

where $H_s$ and $H_o$ are sets of transitions:

- $H_s, H_o = \{ t_k \mapsto (v_x, v_y, s_x), \ldots \}$
- $H = H_s \cup H_o$

Additional coherence constraint:

$$H(t) = (v_x, v_y, s_x) \land H'(t') = (v'_x, v'_y, s_x) \implies v_x = v'_x$$

Transitions (R/G) are writes with versions incrementation.
Pair snapshot specification

\[ H' = H'_s \cup H'_o \]

\[ \{ H_s = \emptyset \} \text{write}_{-x}(v) \{ \exists t, v_y, s_x. H'(t) = (-, v_y, s_x) \]
\[ \quad \wedge H'_s = [t+1 \mapsto (v, v_y, s_x+1)] \} \at F_{ps} \]

\[ \{ H_s = \emptyset \} \text{write}_{-y}(v) \{ \exists t, v_x, s_x. H'(t) = (v_x, -, s_x) \]
\[ \quad \wedge H'_s = [t+1 \mapsto (v_x, v, s_x)] \} \at F_{ps} \]

\[ \{ H_s = \emptyset \} \text{read}_{\text{pair}}() \{ \exists t, v_x, v_y, s_x. H'(t) = (v_x, v_y, s_x) \}
\[ \quad \wedge H'_s = \emptyset \]
\[ \wedge \text{res} = (v_x, v_y) \} \at F_{ps} \]

The proof is trivial, by coherence requirement and Rely
Stacks specification

\[ H' = H'_s \cup H'_o \]

\{ \( H_s = \emptyset \) \} \textbf{push}(x) \{ \exists t, xs. H'(t) = xs \land H'_s = [t+1 \mapsto (x::xs)] \}@C_{stack}

\{ \( H_s = \emptyset \) \} \textbf{pop}() \{ \text{if (res = } \text{Just}(x)) \]

\text{then } \exists t, xs. H'(t) = x::xs \land H'_s = [t+1 \mapsto xs] \\
\text{else } \exists t. H'_s = \emptyset \land H'(t) = \text{Nil} \}@C_{stack}

\{ \( H_s = \emptyset \) \} \textbf{contains}(x) \{ \exists t, xs. H'(t) = xs \land x \in xs \}

\text{if (res) then } \exists t, xs. H'(t) = xs \land x \in xs \\
\text{else } \exists t. H(t) = xs \land x \not\in xs \}@C_{stack}
What about granularity abstraction?

(for the sake of Hoare-style reasoning simplification)
Granularity abstraction via linearizability

- If and ADT $c_1$ is linearizable wrt to $c_2$, we can replace $c_1$ by $c_2$ for the sake of simpler reasoning (Vafeiadis:PhD08, Liang-Feng:PLDI13)
- Alternatively, if $c_1$ is contextual refinement of $c_2$, its clients can reason as about $c_2$ (Filipović-al:TCS10, Turon-al:ICFP13)
- Both linearizability and CR are relations on program modules
- Logics for them are inherently relational
Why don’t us relate \textit{state-transition systems} instead?

(which is, presumably, easier than relating \textit{programs})
defines Rely/Guarantee

Refinement function:

\[ \Phi : F \rightarrow C \]
Refinement function:

$$\Phi: F \rightarrow C$$

simple “coarse-grained” concurroid
A state in implementation concurroid

$\sigma_{fg}$
Establishing Refinement

Transitions of implementation concurroid

\[\sigma_{fg} \rightarrow \ldots \rightarrow \sigma'_{fg}\]

Transitions of specification concurroid

\[\sigma_{cg} \rightarrow \sigma'_{cg}\]

stuttering
Refinement for pair snapshots
Pair spec we used to have

\[ H' = H'_s \cup H'_o \]

\[
\{ H_s = \emptyset \} \text{write}_x(v) \{ \exists t, v_y, s_x. H'(t) = (-, v_y, s_x) \\
\quad \land H'_s = [t+1 \mapsto (v, v_y, s_x+1)] \} \text{@F}_{ps}
\]

\[
\{ H_s = \emptyset \} \text{write}_y(v) \{ \exists t, v_x, s_x. H'(t) = (v_x, -, s_x) \\
\quad \land H'_s = [t+1 \mapsto (v_x, v, s_x)] \} \text{@F}_{ps}
\]

\[
\{ H_s = \emptyset \} \text{read_pair()} \{ \exists t, v_x, v_y, s_x. H'(t) = (v_x, v_y, s_x) \land H'_s = H_s \\
\quad \land res = (v_x, v_y) \} \text{@F}_{ps}
\]
Pair spec we used to have

\[ H' = H'_s \cup H'_o \]

\[
\{ H_s = \emptyset \} \text{write}_x(v) \{ \exists t, v_y, s_x. H'(t) = (-, v_y, s_x) \wedge H'_s = [t+1 \mapsto (v, v_y, s_x+1)] \} @F_{ps}
\]

\[
\{ H_s = \emptyset \} \text{write}_y(v) \{ \exists t, v_x, s_x. H'(t) = (v_x, -, s_x) \wedge H'_s = [t+1 \mapsto (v_x, v, s_x)] \} @F_{ps}
\]

\[
\{ H_s = \emptyset \} \text{read}_\text{pair}( ) \{ \exists t, v_x, v_y, s_x. H'(t) = (v_x, v_y, s_x) \wedge H'_s = H_s \wedge \text{res} = (v_x, v_y) \} @F_{ps}
\]
Pair spec we used to have

\[ H' = H'_s \cup H'_o \]

\[
\{ H_s = \emptyset \} \text{write}_x(v) \{ \exists t, v_y, s_x. H'_s(t) = (-, v_y, s_x) \]
\[
\wedge H'_s = [t+1 \mapsto (v, v_y, s_x+1)] \} @F_{ps}
\]

\[
\{ H_s = \emptyset \} \text{write}_y(v) \{ \exists t, v_x, s_x. H'_s(t) = (v_x, -, s_x) \]
\[
\wedge H'_s = [t+1 \mapsto (v_x, v, s_x)] \} @F_{ps}
\]

\[
\{ H_s = \emptyset \} \text{read}_\text{pair}() \{ \exists t, v_x, v_y, s_x. H'_s(t) = (v_x, v_y, s_x) \wedge H'_s = H_s \]
\[
\wedge \text{res} = (v_x, v_y) \} @F_{ps}
\]
Coarse-grained Pair concurroid

$C_{ps} = H_s \cup H_o$

- No timestamps, no value versions
- $H_s, H_o = \{ (v_x, v_y), \ldots \}$ — multi-sets
- $H = H_s \cup H_o$
- Transitions (R/G) are just atomic writes
- $\Phi: F_{ps} \rightarrow C_{ps}$ erases versions and timestamps
Pair spec we have now

$H' = H'_s \cup H'_o$

\[ \begin{array}{l}
\{ H_s = \emptyset \} \text{write}_x(v) \{ \exists v_y. (-, v_y) \in H' \\
\quad \land H'_s = \{(v, v_y)\} \} \at C_{ps}\\
\{ H_s = \emptyset \} \text{write}_y(v) \{ \exists v_x, s_x. H'(t) = (v_x, -) \\
\quad \land H'_s = \{(v_x, v)\} \} \at C_{ps}\\
\{ H_s = \emptyset \} \text{read_pair}() \{ \exists v_x, v_y. (v_x, v_y) \in H' \land H'_s = H_s \\
\quad \land \text{res} = (v_x, v_y) \} \at C_{ps}\\
\end{array} \]
Meeting some old friends
CSL Resource Rule

\[ \Gamma, r : l \vdash \{ p \} c \{ q \} \]

\[ \Gamma \vdash \{ p \} \text{resource } r \text{ in } c \{ q \} \]
FCSL Generalized Resource Rule

Nanevski-al:ESOP14

\[ \vdash \{ \text{priv} \mapsto^s h \ast p \} \ _c \ {\{ \text{priv} \mapsto^s h' \ast q \} @ (P \times U) \times V} \]

\[ \vdash \{ \Phi(g, h) \ast (\Psi(g) \rightarrow p) \} \ _{\text{hide}_{\Psi,g}(c)} \ {\{ \exists g' \Phi(g, h) \ast (\Psi(g) \rightarrow q) \} @ P \times U} \]

where \( \Phi \) is defined as \( \Psi \)-based refinement

Scoped resource allocation is a particular case of refinement!
Exploring the zoo of STS simulations
fine-grained implementation

refinement

coarse-grained implementation
Restricted Stacks

∀ P : Elem → Prop.

\{ P(x) \} push(x) \{ true \}

\{ true \} pop() \{ res = Nothing \\
\quad ∨ ∃x. res = Just(x) ∧ P(x) \}
\{ p \} c \{ q \} \at \ C

\{ S(p) \} \textbf{simulate}_S (c) \{ S(q) \} \at \ C_R

course-grained implementation

restricted implementation
Restricted stacks

∀ \( P: Elem \rightarrow Prop. \)

\[
\{ P(x) \} \quad \text{push}(x) \quad \{ \text{true} \}
\]

\[
\{ \text{true} \} \quad \text{pop}() \quad \{ \quad \text{res} = \text{Nothing} \\
\quad \lor \quad \exists x. \text{res} = \text{Just}(x) \land P(x) \quad \}
\]

accepts any elements

\[ S(P) \]
simulation

accepts \( P \)-admissible elements
To take away

- We suggest an alternative to linearizability as the only way to provide canonical specifications and establish granularity abstraction;
- Histories-as-resource give “canonical” concurrent specs;
- Granularity abstraction could be established via STS simulation techniques (hopefully).

Some open questions:

- What is use for other simulation (backwards, FB, BF)?
- So far we didn’t need prophecy variables? Can we avoid them at all?
- Can we define the notion of “atomicity” in terms of STS and simulations?

Thanks!