Revisiting Auxiliary Variables

Stephan Merz
joint work with Leslie Lamport

Inria & LORIA, Nancy, France

IFIP Working Group 2.2

Bordeaux, September 2017

Specifications of State Machines

- Standard way of describing algorithms
  - initial condition, next-state relation express what may happen
  - fairness / liveness conditions assert what must happen
- Part of the state may be hidden
  - do not expose implementation details
  - delimit observable behavior that should be implemented
- Concrete syntax: TLA$^+$ $\quad \exists\!\!\!\exists\, x : \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land L$

Refinement of State Machines

- From high-level specification to concrete implementation
  - executions of lower-level state machine coherent with specification
  - formally: inclusion of set of (observable) state sequences

$$(\exists\, y : \mathit{Impl}) \Rightarrow (\exists\, x : \mathit{Spec})$$

Refinement of State Machines

- From high-level specification to concrete implementation
  - executions of lower-level state machine coherent with specification
  - formally: inclusion of set of (observable) state sequences

$$(\exists\, y : \mathit{Impl}) \Rightarrow (\exists\, x : \mathit{Spec})$$

- Standard proof technique: refinement mapping
  - reconstruct high-level internal state from low-level state

$$\mathit{Impl} \Rightarrow \mathit{Spec}\,\{f/x\}$$

  - pointwise computation of internal state components

Example: Compute the Maximum Input Value

- First specification: store the set of all inputs

$$
\begin{aligned}
Init_1 &\triangleq inp = \{\} \land lastinp = -\infty \land max = -\infty \\
Input_1(x) &\triangleq inp' = inp \cup \{x\} \land lastinp' = x \land max' = Max(inp') \\
Next_1 &\triangleq \exists x \in Int : Input_1(x) \\
Spec_1 &\triangleq \boldsymbol{\exists}\, inp, lastinp : Init_1 \land \Box[Next_1]_{\langle inp, lastinp, max \rangle}
\end{aligned}
$$

Example: Compute the Maximum Input Value

- First specification: store the set of all inputs

$$
\begin{aligned}
Init_1 &\triangleq inp = \{\} \land lastinp = -\infty \land max = -\infty \\
Input_1(x) &\triangleq inp' = inp \cup \{x\} \land lastinp' = x \land max' = Max(inp') \\
Next_1 &\triangleq \exists x \in Int : Input_1(x) \\
Spec_1 &\triangleq \boldsymbol{\exists}\, inp, lastinp : Init_1 \land \Box[Next_1]_{\langle inp, lastinp, max \rangle}
\end{aligned}
$$

- Second specification: store just the maximum value

$$
\begin{aligned}
Init_2 &\triangleq lastinp = -\infty \land max = -\infty \\
Input_2(x) &\triangleq lastinp' = x \land max' = \text{IF } x > max \text{ THEN } x \text{ ELSE } max \\
Next_2 &\triangleq \exists x \in Int : Input_2(x) \\
Spec_2 &\triangleq \boldsymbol{\exists}\, lastinp : Init_2 \land \Box[Next_2]_{\langle lastinp, max \rangle}
\end{aligned}
$$

Example: Compute the Maximum Input Value

- First specification: store the set of all inputs

$$
\begin{aligned}
Init_1 &\triangleq inp = \{\} \land lastinp = -\infty \land max = -\infty \\
Input_1(x) &\triangleq inp' = inp \cup \{x\} \land lastinp' = x \land max' = Max(inp') \\
Next_1 &\triangleq \exists x \in Int : Input_1(x) \\
Spec_1 &\triangleq \boldsymbol{\exists}\, inp, lastinp : Init_1 \land \Box[Next_1]_{\langle inp, lastinp, max \rangle}
\end{aligned}
$$

- Second specification: store just the maximum value

$$
\begin{aligned}
Init_2 &\triangleq lastinp = -\infty \land max = -\infty \\
Input_2(x) &\triangleq lastinp' = x \land max' = \text{IF } x > max \text{ THEN } x \text{ ELSE } max \\
Next_2 &\triangleq \exists x \in Int : Input_2(x) \\
Spec_2 &\triangleq \boldsymbol{\exists}\, lastinp : Init_2 \land \Box[Next_2]_{\langle lastinp, max \rangle}
\end{aligned}
$$

What is the formal relationship between the two specifications?

Proving Refinement

- The two specifications are equivalent
  - they generate same externally visible behaviors (variable *max*)

Proving Refinement

- The two specifications are equivalent
  - they generate same externally visible behaviors (variable *max*)
- $Spec_1$ refines $Spec_2$
  1. prove invariant max = Max(inp)
  2. use identical refinement mapping for variable *lastinp*

$$Spec_1 \land \Box(max = Max(inp)) \Rightarrow Init_2 \land \Box[Next_2]_{\langle lastinp, max \rangle}$$

Proving Refinement

- The two specifications are equivalent
  - they generate same externally visible behaviors (variable *max*)
- $Spec_1$ refines $Spec_2$
  1. prove invariant max = Max(inp)
  2. use identical refinement mapping for variable *lastinp*

$$Spec_1 \land \Box(max = Max(inp)) \Rightarrow Init_2 \land \Box[Next_2]_{\langle lastinp, max \rangle}$$

- $Spec_2$ refines $Spec_1$
  - holds semantically, but no refinement mapping
  - cannot compute set of inputs given the maximum value

Proving Refinement

- The two specifications are equivalent
  - they generate same externally visible behaviors (variable *max*)
- $Spec_1$ refines $Spec_2$
  1. prove invariant $\boxed{\text{max} = \text{Max(inp)}}$
  2. use identical refinement mapping for variable *lastinp*

$$Spec_1 \wedge \Box(max = Max(inp)) \Rightarrow Init_2 \wedge \Box[Next_2]_{\langle lastinp, max \rangle}$$

- $Spec_2$ refines $Spec_1$
  - holds semantically, but no refinement mapping
  - cannot compute set of inputs given the maximum value

Refinement mappings alone are incomplete

Auxiliary Variables

- Augment implementation, then construct refinement mapping
  1. specific rules justifying auxiliary variables: $\mathit{Impl} \equiv \exists\, a : \mathit{Impl}^a$
  2. augmented specification refines high-level: $\mathit{Impl}^a \Rightarrow \exists\, x : \mathit{Spec}$

Auxiliary Variables

- Augment implementation, then construct refinement mapping
  1. specific rules justifying auxiliary variables: $\mathit{Impl} \equiv \exists\, a : \mathit{Impl}^a$
  2. augmented specification refines high-level: $\mathit{Impl}^a \Rightarrow \exists\, x : \mathit{Spec}$
- Two particular kinds of auxiliary variables
  - history variables: record information about previous states
  - prophecy variables: predict information about future states

Auxiliary Variables

- Augment implementation, then construct refinement mapping
  1. specific rules justifying auxiliary variables: $Impl \equiv \exists\, a : Impl^a$
  2. augmented specification refines high-level: $Impl^a \Rightarrow \exists\, x : Spec$
- Two particular kinds of auxiliary variables
  - history variables: record information about previous states
  - prophecy variables: predict information about future states
- Classic reference

  M. Abadi, L. Lamport. The Existence of Refinement Mappings. TCS (1991).

  - introduces history and prophecy variables
  - proves completeness under certain conditions
  - closely related: forward / backward simulations

Outline

Record Information About Past States

- Update history variable at every transition

$$Spec \equiv \exists\, h : Spec \land h = h_0 \land \Box[vars' \neq vars \land h' = f(h)]_{\langle vars, h \rangle}$$

  - variable h does not occur in *Spec*, *vars* or h_0
  - term $f(h)$ does not contain h'
  - h_0 is the initial value of the history variable
  - f represents the update function applied at every observable step

Record Information About Past States

- Update history variable at every transition

$$Spec \equiv \exists\, h : Spec \wedge h = h_0 \wedge \Box[vars' \neq vars \wedge h' = f(h)]_{\langle vars,h \rangle}$$

  - variable h does not occur in $Spec$, $vars$ or h_0
  - term $f(h)$ does not contain h'
  - h_0 is the initial value of the history variable
  - f represents the update function applied at every observable step

- Example: step counter

$$Spec \equiv \exists\, h : Spec \wedge h = 0 \wedge \Box[vars' \neq vars \wedge h' = h + 1]_{\langle vars,h \rangle}$$

  - similar: record the input values during executions of $Spec_2$

Parameterized Refinement Mappings

- Idea: many refinement mappings are better than one

  1. introduce parameterized specification equivalent to low-level spec

     $$Impl \equiv \exists \beta \in S : PImpl(\beta)$$

  2. define separate refinement mappings per parameter value

     $$\forall \beta \in S : PImpl(\beta) \Rightarrow Spec\,\{f(\beta)/x\}$$

Parameterized Refinement Mappings

- Idea: many refinement mappings are better than one
  1. introduce parameterized specification equivalent to low-level spec

     $$Impl \equiv \exists \beta \in S : PImpl(\beta)$$

  2. define separate refinement mappings per parameter value

     $$\forall \beta \in S : PImpl(\beta) \Rightarrow Spec\,\{f(\beta)/x\}$$

- Example: introduce a downward counter

$$
\begin{aligned}
Impl &\triangleq n = 0 \land \Box[n' = n+1]_{\langle n \rangle} \land \Diamond\Box[n' = n]_{\langle n \rangle} \\
Spec &\triangleq n = 0 \land k \in \mathbb{N} \land \Box[k > 0 \land n' = n+1 \land k' = k-1]_{\langle k,n \rangle}
\end{aligned}
$$

Prove $Impl \Rightarrow \exists\, k : Spec$

Parameterized Refinement Mappings

- Idea: many refinement mappings are better than one
  1. introduce parameterized specification equivalent to low-level spec

     $$Impl \equiv \exists \beta \in S : PImpl(\beta)$$

  2. define separate refinement mappings per parameter value

     $$\forall \beta \in S : PImpl(\beta) \Rightarrow Spec\,\{f(\beta)/x\}$$

- Example: introduce a downward counter

$$\begin{array}{lcl} Impl & \triangleq & n = 0 \land \Box[n' = n+1]_{\langle n \rangle} \land \Diamond\Box[n' = n]_{\langle n \rangle} \\ Spec & \triangleq & n = 0 \land k \in \mathbb{N} \land \Box[k > 0 \land n' = n+1 \land k' = k-1]_{\langle k,n \rangle} \end{array}$$

Prove $Impl \Rightarrow \exists\!\!\exists\, k : Spec$

  1. observe $Impl \equiv \exists m \in \mathbb{N} : Impl \land \Box(n \leq m)$
  2. prove $\forall m \in \mathbb{N} : Impl \land \Box(n \leq m) \Rightarrow Spec\,\{m-n/k\}$

Completeness: Assume $Impl \Rightarrow \exists\, x : Spec$

1. Introduce a step counter h as a history variable

   $Impl \equiv \exists\, h : Impl^h$

Completeness: Assume $Impl \Rightarrow \exists\, x : Spec$

1. Introduce a step counter h as a history variable

$$Impl \equiv \exists\, h : Impl^h$$

2. Define parameterized low-level specification

$$PImpl(\beta) \triangleq Impl^h \wedge \Box(vars = \beta[h])$$

- β : sequence of states that "predicts" actual execution
- $Impl \equiv \exists \beta \in [\mathbb{N} \to St] : PImpl(\beta)$ semantic reasoning

Completeness: Assume $Impl \Rightarrow \exists\, x : Spec$

1. Introduce a step counter h as a history variable

   $$Impl \equiv \exists\, h : Impl^h$$

2. Define parameterized low-level specification

   $$PImpl(\beta) \;\triangleq\; Impl^h \land \Box(vars = \beta[h])$$

   - β : sequence of states that "predicts" actual execution
   - $Impl \equiv \exists \beta \in [\mathbb{N} \to St] : PImpl(\beta)$ semantic reasoning

3. Define parameterized refinement mapping

   - fix $\beta \in [\mathbb{N} \to St]$ such that $PImpl(\beta)$ holds
   - because of refinement, there exists $\gamma_\beta \in [\mathbb{N} \to X]$ satisfying *Spec*

Completeness: Assume $Impl \Rightarrow \exists\, x : Spec$

1. Introduce a step counter h as a history variable

   $Impl \equiv \exists\, h : Impl^h$

2. Define parameterized low-level specification

   $PImpl(\beta) \triangleq Impl^h \wedge \Box(vars = \beta[h])$

   - β : sequence of states that "predicts" actual execution
   - $Impl \equiv \exists \beta \in [\mathbb{N} \to St] : PImpl(\beta)$ semantic reasoning

3. Define parameterized refinement mapping

   - fix $\beta \in [\mathbb{N} \to St]$ such that $PImpl(\beta)$ holds
   - because of refinement, there exists $\gamma_\beta \in [\mathbb{N} \to X]$ satisfying *Spec*
   - conclude $\forall \beta \in [\mathbb{N} \to St] : PImpl(\beta) \Rightarrow Spec\,\{\, \gamma_\beta[h] \,/\, x \,\}$

Completeness: Assume $Impl \Rightarrow \exists\, x : Spec$

1. Introduce a step counter h as a history variable

   $$Impl \equiv \exists\, h : Impl^h$$

2. Define parameterized low-level specification

   $$PImpl(\beta) \triangleq Impl^h \land \Box(vars = \beta[h])$$

   - β : sequence of states that "predicts" actual execution
   - $Impl \equiv \exists \beta \in [\mathbb{N} \to St] : PImpl(\beta)$ semantic reasoning

3. Define parameterized refinement mapping

   - fix $\beta \in [\mathbb{N} \to St]$ such that $PImpl(\beta)$ holds
   - because of refinement, there exists $\gamma_\beta \in [\mathbb{N} \to X]$ satisfying *Spec*
   - conclude $\forall \beta \in [\mathbb{N} \to St] : PImpl(\beta) \Rightarrow Spec\,\{\, \gamma_\beta[h] \,/\, x \,\}$

Related to Hesselink's eternity variables

Outline

Abadi-Lamport Prophecy Variables

- Similar to history variables, with "time running backwards"

$$\frac{S \neq \{\} \qquad IsFiniteSet(S) \qquad Spec \Rightarrow \Box(\forall y \in S : f(y) \in S)}{Spec \;\equiv\; \exists\!\!\!\exists\, p : Spec \land \Box(p \in S) \land \Box[vars' \neq vars \land p = f(p')]_{\langle vars,p \rangle}}$$

  - variable p does not occur in *Spec*, *vars* or S
  - $f(y)$ does not contain p

Abadi-Lamport Prophecy Variables

- Similar to history variables, with "time running backwards"

$$\frac{S \neq \{\} \qquad IsFiniteSet(S) \qquad Spec \Rightarrow \Box(\forall y \in S : f(y) \in S)}{Spec \;\equiv\; \exists\!\!\exists\, p : Spec \wedge \Box(p \in S) \wedge \Box[vars' \neq vars \wedge p = f(p')]_{\langle vars, p \rangle}}$$

  - variable p does not occur in *Spec*, *vars* or S
  - $f(y)$ does not contain p

- Differences to rule for history variables
  - invariant "type condition" replaces initialization
  - finiteness of S required for soundness (König's lemma)

Abadi-Lamport Prophecy Variables

- Similar to history variables, with "time running backwards"

$$\frac{S \neq \{\} \qquad \mathit{IsFiniteSet}(S) \qquad \mathit{Spec} \Rightarrow \Box(\forall y \in S : f(y) \in S)}{\mathit{Spec} \;\equiv\; \exists\!\!\!\exists\, p : \mathit{Spec} \land \Box(p \in S) \land \Box[\mathit{vars}' \neq \mathit{vars} \land p = f(p')]_{\langle \mathit{vars},p \rangle}}$$

  - variable p does not occur in Spec, vars or S
  - $f(y)$ does not contain p

- Differences to rule for history variables
  - invariant "type condition" replaces initialization
  - finiteness of S required for soundness (König's lemma)

- Rule found to be difficult to apply in practice

Prophesize Next Occurrence of an Action

- Consider a system that repeatedly produces integer values

$$
\begin{aligned}
Init &\triangleq val = \{\} \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \\
Spec &\triangleq Init \land \Box[\exists n \in \mathbb{N} : Prod(n)]_{\langle val \rangle}
\end{aligned}
$$

Prophesize Next Occurrence of an Action

- Consider a system that repeatedly produces integer values

$$
\begin{aligned}
Init &\triangleq val = \{\} \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \\
Spec &\triangleq Init \land \Box[\exists n \in \mathbb{N} : Prod(n)]_{\langle val \rangle}
\end{aligned}
$$

- Now predict the next value to be produced

$$Spec \Rightarrow \boldsymbol{\exists}\, p : Init \land p \in \mathbb{N} \land \Box[Prod(p) \land p' \in \mathbb{N}]_{\langle val, p \rangle}$$

Prophesize Next Occurrence of an Action

- Consider a system that repeatedly produces integer values

$$
\begin{aligned}
Init &\triangleq val = \{\} \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \\
Spec &\triangleq Init \land \Box[\exists n \in \mathbb{N} : Prod(n)]_{\langle val \rangle}
\end{aligned}
$$

- Now predict the next value to be produced

$$Spec \Rightarrow \boldsymbol{\exists}\, p : Init \land p \in \mathbb{N} \land \Box[Prod(p) \land p' \in \mathbb{N}]_{\langle val, p \rangle}$$

- Note: one need not be able to construct a prophecy variable
  - prophecy variables are used in proofs
  - the augmented specification need not be implementable

Simple Prophecy Variables: Introduction Rule

- Predict the parameter of next occurrence of given action

$$\frac{S \neq \{\} \qquad Spec \Rightarrow \Box[\forall x : A(x) \Rightarrow x \in S]_{\langle vars \rangle}}{\begin{array}{l} Spec \equiv \exists\!\!\exists\, p : \land\, Spec \land p \in S \\ \qquad\qquad\qquad \land\, \Box[\land\, vars' \neq vars \\ \qquad\qquad\qquad\qquad \land\, \text{IF } \exists x : A(x) \text{ THEN } A(p) \land p' \in S \text{ ELSE } p' = p]_{\langle vars, p \rangle} \end{array}}$$

  - $A(x)$ an action without occurrences of p
  - p predicts for which value A will occur next
  - other actions leave p unchanged
  - variant: predict which action will be performed next

Simple Prophecy Variables: Introduction Rule

- Predict the parameter of next occurrence of given action

$$\frac{S \neq \{\} \qquad Spec \Rightarrow \Box[\forall x : A(x) \Rightarrow x \in S]_{\langle vars \rangle}}{\begin{array}{l} Spec \equiv \exists\!\!\exists\, p : \land\ Spec \land p \in S \\ \qquad\qquad \land\ \Box[\land\ vars' \neq vars \\ \qquad\qquad\quad \land\ \text{IF } \exists x : A(x) \text{ THEN } A(p) \land p' \in S \text{ ELSE } p' = p]_{\langle vars, p \rangle} \end{array}}$$

  - $A(x)$ an action without occurrences of p
  - p predicts for which value A will occur next
  - other actions leave p unchanged
  - variant: predict which action will be performed next

- Simple soundness proof
  - suitable value for p determined at next occurrence of A
  - if A doesn't occur anymore, the value of p doesn't matter

Outline

Auxiliary Functions

- Aggregate auxiliary variables in an array

$$\frac{S \neq \{\} \qquad \forall x \in S : Spec \equiv \exists\!\!\exists\, y : \Box(y \in T) \land \Box[v \neq v]_{\langle y \rangle} \land ASpec(x,y)}{Spec \equiv \exists\!\!\exists\, f : \Box(f \in [S \to T]) \land \Box[v \neq v]_{\langle f \rangle} \land \forall x \in S : ASpec(x, f[x])}$$

  - y, f do not occur in v
  - similar to introducing a Skolem function in predicate logic
  - premise will be established by previous rules

Auxiliary Functions

- Aggregate auxiliary variables in an array

$$\frac{S \neq \{\} \qquad \forall x \in S : Spec \equiv \exists\, y : \Box(y \in T) \land \Box[v \neq v]_{\langle y \rangle} \land ASpec(x,y)}{Spec \equiv \exists\, f : \Box(f \in [S \to T]) \land \Box[v \neq v]_{\langle f \rangle} \land \forall x \in S : ASpec(x, f[x])}$$

  - y, f do not occur in v
  - similar to introducing a Skolem function in predicate logic
  - premise will be established by previous rules

- Generic principle for combining auxiliary variables
  - natural application: parameterized verification problems
  - can be used to justify original rule for prophecy variables

Example

- Extend the producer system by an explicit output action

$$\begin{aligned}
Init &\triangleq val = \{\} \land out = none \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \land out' = out \\
Out &\triangleq out' \in val \land val' = val \setminus \{out'\} \\
Next &\triangleq (\exists n \in \mathbb{N} : Prod(n)) \lor Out \\
Spec &\triangleq Init \land \Box[Next]_{\langle val, out \rangle}
\end{aligned}$$

Example

- Extend the producer system by an explicit output action

$$
\begin{aligned}
Init &\triangleq val = \{\} \land out = none \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \land out' = out \\
Out &\triangleq out' \in val \land val' = val \setminus \{out'\} \\
Next &\triangleq (\exists n \in \mathbb{N} : Prod(n)) \lor Out \\
Spec &\triangleq Init \land \Box[Next]_{\langle val, out \rangle}
\end{aligned}
$$

- Now add the possibility of "undoing" production

$$
\begin{aligned}
Undo(n) &\triangleq val' = val \setminus \{n\} \land out' = out \\
NextU &\triangleq Next \lor \exists n \in \mathbb{N} : Undo(n) \\
SpecU &\triangleq Init \land \Box[NextU]_{\langle val, out \rangle}
\end{aligned}
$$

Example

- Extend the producer system by an explicit output action

$$
\begin{aligned}
Init &\triangleq val = \{\} \land out = none \\
Prod(n) &\triangleq n \notin val \land val' = val \cup \{n\} \land out' = out \\
Out &\triangleq out' \in val \land val' = val \setminus \{out'\} \\
Next &\triangleq (\exists n \in \mathbb{N} : Prod(n)) \lor Out \\
Spec &\triangleq Init \land \Box[Next]_{\langle val, out \rangle}
\end{aligned}
$$

- Now add the possibility of "undoing" production

$$
\begin{aligned}
Undo(n) &\triangleq val' = val \setminus \{n\} \land out' = out \\
NextU &\triangleq Next \lor \exists n \in \mathbb{N} : Undo(n) \\
SpecU &\triangleq Init \land \Box[NextU]_{\langle val, out \rangle}
\end{aligned}
$$

- Prove equivalence of $\exists\!\!\exists\, val : SpecU$ and $\exists\!\!\exists\, val : Spec$

Proving $SpecU \Rightarrow \exists\, val : Spec$

- For every $k \in \mathbb{N}$, predict if it will be output or not

$$
\begin{aligned}
Init^p &\triangleq Init \land p \in \{\text{“out”}, \text{“undo”}\} \\
Prod^p(n) &\triangleq Prod(n) \land \text{IF } n = k \text{ THEN } p' \in \{\text{“out”}, \text{“undo”}\} \text{ ELSE } p' = p \\
Out^p &\triangleq Out \land (out' = k \Rightarrow p = \text{“out”}) \land p' = p \\
Undo^p(n) &\triangleq Undo \land (n = k \Rightarrow p = \text{“undo”}) \land p' = p \\
NextU^p &\triangleq Out^p \lor (\exists n \in \mathbb{N} : Prod^p(n) \lor Undo^p(n)) \\
SpecU^p &\triangleq Init^p \land \Box[NextU^p]_{\langle val, out, p\rangle}
\end{aligned}
$$

  - use simple prophecy rule to prove $\forall k \in \mathbb{N} : SpecU \equiv \exists\, p : SpecU^p$

Proving $SpecU \Rightarrow \exists\, val : Spec$

- For every $k \in \mathbb{N}$, predict if it will be output or not

$$
\begin{aligned}
Init^p &\triangleq Init \land p \in \{\text{“out”}, \text{“undo”}\} \\
Prod^p(n) &\triangleq Prod(n) \land \text{IF } n = k \text{ THEN } p' \in \{\text{“out”}, \text{“undo”}\} \text{ ELSE } p' = p \\
Out^p &\triangleq Out \land (out' = k \Rightarrow p = \text{“out”}) \land p' = p \\
Undo^p(n) &\triangleq Undo \land (n = k \Rightarrow p = \text{“undo”}) \land p' = p \\
NextU^p &\triangleq Out^p \lor (\exists n \in \mathbb{N} : Prod^p(n) \lor Undo^p(n)) \\
SpecU^p &\triangleq Init^p \land \Box[NextU^p]_{\langle val, out, p\rangle}
\end{aligned}
$$

  - use simple prophecy rule to prove $\forall k \in \mathbb{N} : SpecU \equiv \exists\, p : SpecU^p$

- Use array rule to combine these predictions

$$SpecU \equiv \exists\, f : \Box(f \in [\mathbb{N} \to \{\text{“out”}, \text{“undo”}\}]) \land \ldots$$

Proving $SpecU \Rightarrow \exists\, val : Spec$

- For every $k \in \mathbb{N}$, predict if it will be output or not

$$
\begin{array}{lcl}
Init^p & \triangleq & Init \land p \in \{\text{“out”}, \text{“undo”}\} \\
Prod^p(n) & \triangleq & Prod(n) \land \text{IF } n = k \text{ THEN } p' \in \{\text{“out”}, \text{“undo”}\} \text{ ELSE } p' = p \\
Out^p & \triangleq & Out \land (out' = k \Rightarrow p = \text{“out”}) \land p' = p \\
Undo^p(n) & \triangleq & Undo \land (n = k \Rightarrow p = \text{“undo”}) \land p' = p \\
NextU^p & \triangleq & Out^p \lor (\exists n \in \mathbb{N} : Prod^p(n) \lor Undo^p(n)) \\
SpecU^p & \triangleq & Init^p \land \Box[NextU^p]_{\langle val, out, p\rangle}
\end{array}
$$

  - use simple prophecy rule to prove $\forall k \in \mathbb{N} : SpecU \equiv \exists\, p : SpecU^p$

- Use array rule to combine these predictions

$$SpecU \equiv \exists\, f : \Box(f \in [\mathbb{N} \to \{\text{“out”}, \text{“undo”}\}]) \land \ldots$$

- Finally, define suitable refinement mapping

$$SpecU^f \Rightarrow Spec\,\{\, (val \setminus \{k \in \mathbb{N} : f[k] = \text{“undo”}\}) \,/\, val \,\}$$

Outline

Stuttering Steps and Refinement

- Adjust for different granularity of atomic actions
  - typically, refinement introduces lower-level detail
  - low-level transitions are invisible at higher level
  - TLA$^+$ bakes stuttering invariance into the language

Stuttering Steps and Refinement

- Adjust for different granularity of atomic actions
  - typically, refinement introduces lower-level detail
  - low-level transitions are invisible at higher level
  - TLA^+ bakes stuttering invariance into the language
- Occasionally, the high-level specification may take more steps
  - toy example: clock specified with invisible seconds display
  - more realistic: thread completes operation on behalf of another
- Introduce explicit stuttering for defining refinement mapping
  - Abadi-Lamport: stuttering combined with prophecy variables
  - here: separate category of stuttering variables

Proof Rule

- Add stuttering steps in between visible transitions

$$\frac{s_0 \in Nat \qquad \bigwedge_{i \in I} Spec \Rightarrow \Box(st_i \in \mathbb{N})}{Spec \equiv \exists\!\!\exists\, s : Init^s \wedge \Box\big[Dec \vee \bigvee_{i \in I} A_i^s\big]_{\langle v,s \rangle} \wedge L}$$

  - original specification $Spec \;\equiv\; Init \wedge \Box\big[\bigvee_{i \in I} A_i\big]_{\langle v \rangle} \wedge L$
  - initial stuttering $Init^s \;\triangleq\; Init \wedge s = s_0$
  - stuttering after transition $A_i^s \;\triangleq\; A_i \wedge s = 0 \wedge s' = st_i$
  - decrement variable s $Dec \;\triangleq\; s > 0 \wedge s' = s - 1 \wedge v' = v$

Proof Rule

- Add stuttering steps in between visible transitions

$$\frac{s_0 \in Nat \qquad \bigwedge_{i \in I} Spec \Rightarrow \Box(st_i \in \mathbb{N})}{Spec \equiv \exists\!\!\exists\, s : Init^s \wedge \Box\big[Dec \vee \bigvee_{i \in I} A_i^s\big]_{\langle v,s \rangle} \wedge L}$$

  - original specification $Spec \;\equiv\; Init \wedge \Box\big[\bigvee_{i \in I} A_i\big]_{\langle v \rangle} \wedge L$
  - initial stuttering $Init^s \;\triangleq\; Init \wedge s = s_0$
  - stuttering after transition $A_i^s \;\triangleq\; A_i \wedge s = 0 \wedge s' = st_i$
  - decrement variable s $Dec \;\triangleq\; s > 0 \wedge s' = s - 1 \wedge v' = v$

- Obvious generalizations
  - allow for jumps instead of just counting down
  - variable taking values in set with well-founded ordering

Outline

Two Completeness Proofs

1. Predict low-level execution
   - add a step counter to low-level specification
   - use simple prophecy variables to predict n-th state
   - combine these into function predicting low-level behavior
   - choose high-level behavior and define refinement mapping
2. Predict high-level behavior
   - use history variable to record finite prefixes of low-level behavior
   - predict prefixes of high-level behavior compatible with all low-level prefixes, then define refinement mapping

- Remarks
  - second approach: reasoning about finite prefixes suffices …
  - … but "internal continuity" is necessary
  - cf. AL'91: no machine closure or finite internal non-determinism

Wrapping Up

- New look at an old problem
  - refinement mappings are very successful, but incomplete
  - generalization to parameterized refinement mappings
  - auxiliary variables can yield completeness results
  - simple prophecy variables + arrays easier to apply ?
- Validation of the approach
  - catalogue of directly applicable TLA^+ rules
  - applied to toy examples and linearizability proofs
  - formalization in Isabelle/HOL ongoing

Lamport, M.: Auxiliary Variables in TLA^+. arXiv, 2017.