Towards Algorithmic Synthesis of Synchronization for Shared-Memory Concurrent Programs

Roopsha Samanta

The University of Texas at Austin

July 6, 2012

Problem Definition

Given a shared-memory concurrent program P, composed of unsynchronized processes $P_1, \ldots, P_k$, and a (branching-time) temporal logic specification ϕ_{spec} such that $P \not\models \phi_{spec}$, automatically generate synchronized processes $P_1^s, \ldots, P_k^s$ such that $P^s \models \phi_{spec}$.

Example

P :

$x : \{0, 1, 2\}$ with $x = 1$

$P_1 \parallel P_2$

P_1 :

```
L1:   while (x < 2)
L2:       x := x + 1;
L3:   while true;
```

P_2 :

```
T1:   while (x > 0)
T2:       x := x - 1;
T3:   while true;
```

$$\phi_{spec} : AF(L_3 \wedge T_3 \wedge (x = 0 \vee x = 2))$$

Example

P :

$x : \{0, 1, 2\}$ `with` $x = 1$

$P_1 \;\|\; P_2$

$$\not\models$$

$$\phi_{spec} : AF(L_3 \wedge T_3 \wedge (x = 0 \vee x = 2))$$

Q&A

- Why shared-memory concurrent programs?
  - Ubiquitous
  - Hard to write
  - Harder to verify
- Why temporal logic specifications? Why branching-time?
  - Terminating, ongoing programs
  - Safety, liveness
  - Expressivity
- Why synthesize only the synchronization?
  - Trickiest part
  - Simple specifications, tractable

Q&A

- Why shared-memory concurrent programs?
  - Ubiquitous
  - Hard to write
  - Harder to verify
- Why temporal logic specifications? Why branching-time?
  - Terminating, ongoing programs
  - Safety, liveness
  - Expressivity
- Why synthesize only the synchronization?
  - Trickiest part
  - Simple specifications, tractable

Q&A

- Why shared-memory concurrent programs?
  - Ubiquitous
  - Hard to write
  - Harder to verify
- Why temporal logic specifications? Why branching-time?
  - Terminating, ongoing programs
  - Safety, liveness
  - Expressivity
- Why synthesize only the synchronization?
  - Trickiest part
  - Simple specifications, tractable

Q&A

- Why shared-memory concurrent programs?
  - Ubiquitous
  - Hard to write
  - Harder to verify
- Why temporal logic specifications? Why branching-time?
  - Terminating, ongoing programs
  - Safety, liveness
  - Expressivity
- Why synthesize only the synchronization?
  - Trickiest part
  - Simple specifications, tractable

A Vocabulary L

- L-symbols
  - Variable symbols (x)
  - Function, constant symbols (1, 2, $+$, $-$)
  - Predicate, proposition symbols ($\mathtt{L}_1$, $\mathtt{T}_3$, $>$, $<$)
  - *Sort* symbols (`bool`, `int0to2`)
  - Distinguished equality predicate symbol $=$
- (Sorted) L-terms and L-atoms:
  - L-term (t): v or $f(t_1, \ldots, t_m)$ (x, 1, $x + 1$)
  - L-atom(G): $B(t_1, \ldots, t_m)$ or $t_1 = t_2$ ($x > 1$, $\mathtt{L}_1$, $x = 1$)
  - Sorts mapped to nonempty domains ($\mathtt{bool} \rightarrow \{\mathtt{T}, \mathtt{F}\}$, $\mathtt{int0to2} \rightarrow \{0, 1, 2\}$)
  - Symbols, terms, atoms mapped to values in appropriate domains.

A Vocabulary **L**

- **L**-symbols
  - Variable symbols (x)
  - Function, constant symbols (1, 2, $+$, $-$)
  - Predicate, proposition symbols ($\mathtt{L}_1$, $\mathtt{T}_3$, $>$, $<$)
  - *Sort* symbols (`bool`, `int0to2`)
  - Distinguished equality predicate symbol $=$
- (Sorted) **L**-terms and **L**-atoms:
  - **L**-term (t): v or $f(t_1, \ldots, t_m)$ (x, 1, $x + 1$)
  - **L**-atom(G): $B(t_1, \ldots, t_m)$ or $t_1 = t_2$ ($x > 1$, $\mathtt{L}_1$, $x = 1$)
  - Sorts mapped to nonempty domains ($\mathtt{bool} \rightarrow \{\mathtt{T}, \mathtt{F}\}$, $\mathtt{int0to2} \rightarrow \{0, 1, 2\}$)
  - Symbols, terms, atoms mapped to values in appropriate domains.

(Finite-state) Concurrent Programs over **L**

- Syntax:
  - P ::= vardeclaration process || process
  - vardeclaration ::= v_1 : $domain_1, \ldots, v_n$: $domain_n$
  - process ::= localvardeclaration body
  - localvardeclaration ::= vardeclaration
  - body ::= labeledstmt ; body | labeledstmt
  - labeledstmt ::= LOC: atomicstmt
  - atomicstmt ::= assignment | conditiontest | goto | CCR
  - assignment ::= ⟨ v:=t ⟩
  - conditiontest ::= ⟨ `if` G LOC$_{if}$ `else`, LOC$_{else}$ ⟩
  - goto ::= ⟨ `goto` LOC ⟩
  - CCR ::= ⟨ $G \rightarrow$ body ⟩
  - v,LOC ::= **L**-symbol, t ::= **L**-term, G ::= **L**-atom
- Semantics: Finite-state transition system (S, S_0, R)

(Finite-state) Concurrent Programs over **L**

- Syntax:
  - P ::= vardeclaration process || process
  - vardeclaration ::= v_1 : $domain_1, \ldots, v_n$: $domain_n$
  - process ::= localvardeclaration body
  - localvardeclaration ::= vardeclaration
  - body ::= labeledstmt ; body | labeledstmt
  - labeledstmt ::= LOC: atomicstmt
  - atomicstmt ::= assignment | conditiontest | goto | CCR
  - assignment ::= ⟨ v:=t ⟩
  - conditiontest ::= ⟨ `if` G LOC$_{if}$ `else`, LOC$_{else}$ ⟩
  - goto ::= ⟨ `goto` LOC ⟩
  - CCR ::= ⟨ G → body ⟩
  - v,LOC ::= **L**-symbol, t ::= **L**-term, G ::= **L**-atom
- Semantics: Finite-state transition system (S, S_0, R)

(Finite-state) Concurrent Programs over **L**

- Syntax:
  - P ::= vardeclaration process $\|$ process
  - vardeclaration ::= v_1 : $domain_1, \ldots, v_n$: $domain_n$
  - process ::= localvardeclaration body
  - localvardeclaration ::= vardeclaration
  - body ::= labeledstmt ; body | labeledstmt
  - labeledstmt ::= LOC: atomicstmt
  - atomicstmt ::= assignment | conditiontest | goto | CCR
  - assignment ::= ⟨ $v := t$ ⟩
  - conditiontest ::= ⟨ if G LOC$_{if}$ else, LOC$_{else}$ ⟩
  - goto ::= ⟨ goto LOC ⟩
  - CCR ::= ⟨ $G \rightarrow$ body ⟩
  - v,LOC ::= **L**-symbol, t ::= **L**-term, G ::= **L**-atom
- Semantics: Finite-state transition system (S, S_0, R)

(Finite-state) Concurrent Programs over **L**

- Syntax:
  - P ::= vardeclaration process || process
  - vardeclaration ::= v_1 : $domain_1, \ldots, v_n$: $domain_n$
  - process ::= localvardeclaration body
  - localvardeclaration ::= vardeclaration
  - body ::= labeledstmt ; body | labeledstmt
  - labeledstmt ::= LOC: atomicstmt
  - atomicstmt ::= assignment | conditiontest | goto | CCR
  - assignment ::= ⟨ v:=t ⟩
  - conditiontest ::= ⟨ `if` G LOC$_{if}$ `else`, LOC$_{else}$ ⟩
  - goto ::= ⟨ `goto` LOC ⟩
  - CCR ::= ⟨ $G \rightarrow$ body ⟩
  - v,LOC ::= **L**-symbol, t ::= **L**-term, G ::= **L**-atom
- Semantics: Finite-state transition system (S, S_0, R)

(Finite-state) Concurrent Programs over **L**

- Syntax:
  - P ::= vardeclaration process || process
  - vardeclaration ::= v_1 : $domain_1, \ldots, v_n$: $domain_n$
  - process ::= localvardeclaration body
  - localvardeclaration ::= vardeclaration
  - body ::= labeledstmt ; body | labeledstmt
  - labeledstmt ::= LOC: atomicstmt
  - atomicstmt ::= assignment | conditiontest | goto | CCR
  - assignment ::= ⟨ v:=t ⟩
  - conditiontest ::= ⟨ `if` G LOC_{if} `else`, LOC_{else} ⟩
  - goto ::= ⟨ `goto` LOC ⟩
  - CCR ::= ⟨ $G \rightarrow$ body ⟩
  - v,LOC ::= **L**-symbol, t ::= **L**-term, G ::= **L**-atom
- Semantics: Finite-state transition system (S, S_0, R)

CTL-like Specifications over **L** (**L**CTL)

- Syntax: $\phi ::= \mathbf{L}\text{-}atom \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{EX}\,\phi \mid \mathsf{EX}_i\,\phi \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi] \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi]$
- Semantics:
  Defined from semantics of **L**-atoms, propositional and temporal operators

Revisiting Example

P :
$x : \{0, 1, 2\}$ `with` $x = 1$
$P_1 \parallel P_2$

P_1 :

```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:      ⟨ x := x + 1 ⟩;
L3:      ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2 :

```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:      ⟨ x := x − 1 ⟩;
T3:      ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

$$AF(L_4 \wedge T_4 \wedge (x = 0 \vee x = 2))$$

Synthesis Algorithm Sketch

1. Formulate an **L**CTL formula ϕ_P for semantics of P.
2. Construct a tableau T_ϕ for $\phi : \phi_P \wedge \phi_{spec}$.
   If T_ϕ is empty, declare specification as unsatisfiable.
3. If T_ϕ is non-empty, extract a model M for ϕ from it.
4. Decompose M to obtain CCRs to synchronize each process.
5. Compile CCRs into lower level synchronization primitives ([EmeSam2011])

Formulation of ϕ_P

P :
$x : \{0, 1, 2\}$ with $x = 1$
$P_1 \parallel P_2$

P$_1$:

```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:       ⟨ x := x + 1 ⟩;
L3:       ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P$_2$:

```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:       ⟨ x := x - 1 ⟩;
T3:       ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

- At any step, only one process moves
  - $AG(L_1 \Rightarrow AX_2(L_1)) \wedge \ldots$

Formulation of ϕ_P

P:
$x : \{0,1,2\}$ `with` $x = 1$
$P_1 \parallel P_2$

P_1:
```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:      ⟨ x := x + 1 ⟩;
L3:      ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2:
```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:      ⟨ x := x − 1 ⟩;
T3:      ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

- A process is always in exactly one of its locations
  - $AG(L_1 \lor L_2 \lor L_3 \lor L_4)$
  - $AG(L_1 \Rightarrow \neg(L_2 \lor L_3 \lor L_4))$
  - ...

Formulation of ϕ_P

P:
$x : \{0, 1, 2\}$ with $x = 1$
$P_1 \parallel P_2$

P_1:
```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:     ⟨ x := x + 1 ⟩;
L3:     ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2:
```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:     ⟨ x := x - 1 ⟩;
T3:     ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

- Some process always moves.
  - AG(EX *true*)

Formulation of ϕ_P

P:
$x : \{0, 1, 2\}$ `with` $x = 1$
$P_1 \parallel P_2$

P_1:
```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:      ⟨ x := x + 1 ⟩;
L3:      ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2:
```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:      ⟨ x := x - 1 ⟩;
T3:      ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

- Initial condition
  - $L_1 \wedge T_1 \wedge x = 1$

Formulation of ϕ_P

P :
$x : \{0, 1, 2\}$ `with` $x = 1$
$P_1 \parallel P_2$

P_1 :

```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:      ⟨ x := x + 1 ⟩;
L3:      ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2 :

```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:      ⟨ x := x - 1 ⟩;
T3:      ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

- Local process execution
  - $AG((L_1 \wedge x < 2) \Rightarrow AX_1(L_2))$
  - $AG((L_1 \wedge x \geq 2) \Rightarrow AX_1(L_4))$
  - $AG(\bigwedge_{c \in \{0,1,2\}}(L_2 \wedge x = c) \Rightarrow AX_1(L_3 \wedge x = c + 1))$
  - $AG(L_3 \Rightarrow AX_1(L_1))$
  - ...

Construction of Tableau T_ϕ

- Tableau basics:
  - Finite, rooted, directed AND/OR graph
  - Nodes labeled with formulas
  - OR-node formulas valid iff formulas in some AND-node succ. valid
  - AND-node formulas valid iff formulas in all OR-node succ. valid
  - Assume ability to evaluate **L**-atoms and **L**-terms
- Algorithm Sketch:
  - Let root node of T_ϕ be OR-node, labeled ϕ
  - Repeat until no new nodes can be added:
    Add appropriate AND-node succ. of OR-nodes and OR-node succ. of AND-nodes, merging *equivalent* nodes
  - Prune *inconsistent* nodes to get final T_ϕ

Construction of Tableau T_ϕ

- Tableau basics:
  - Finite, rooted, directed AND/OR graph
  - Nodes labeled with formulas
  - OR-node formulas valid iff formulas in some AND-node succ. valid
  - AND-node formulas valid iff formulas in all OR-node succ. valid
  - Assume ability to evaluate **L**-atoms and **L**-terms
- Algorithm Sketch:
  - Let root node of T_ϕ be OR-node, labeled ϕ
  - Repeat until no new nodes can be added:
    Add appropriate AND-node succ. of OR-nodes and OR-node succ. of AND-nodes, merging *equivalent* nodes
  - Prune *inconsistent* nodes to get final T_ϕ

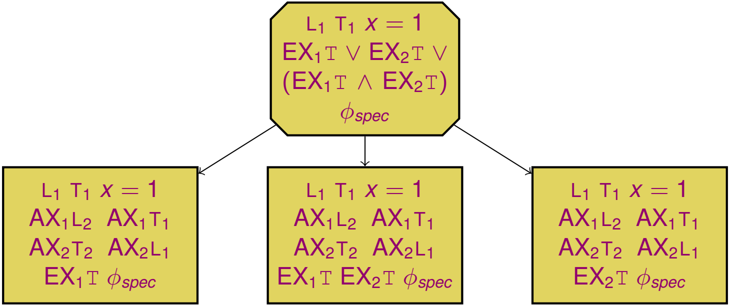
Construction of Tableau T_ϕ

$$\phi_{spec} = \mathsf{AF}(\mathrm{L}_4 \wedge \mathrm{T}_4 \wedge (x = 0 \vee x = 2))$$

$\mathrm{L}_1\ \mathrm{T}_1\ x = 1$
$\mathsf{EX}_1\mathtt{T} \vee \mathsf{EX}_2\mathtt{T} \vee$
$(\mathsf{EX}_1\mathtt{T} \wedge \mathsf{EX}_2\mathtt{T})$
ϕ_{spec}

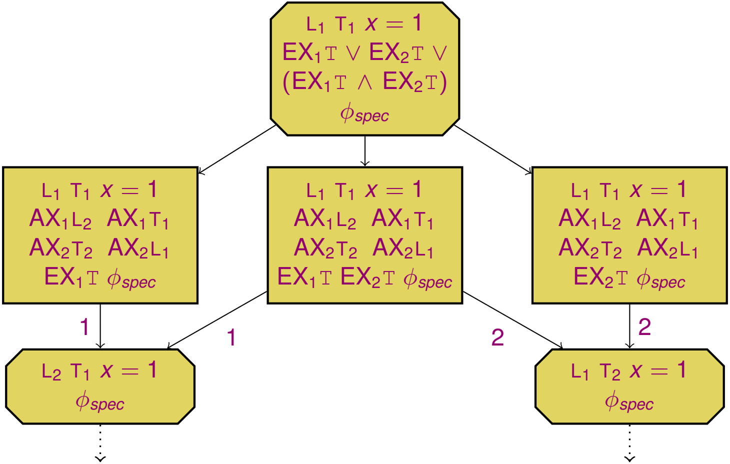
Construction of Tableau T_ϕ

$$\phi_{spec} = \mathsf{AF}(\mathsf{L}_4 \wedge \mathsf{T}_4 \wedge (x = 0 \vee x = 2))$$



Construction of Tableau T_ϕ

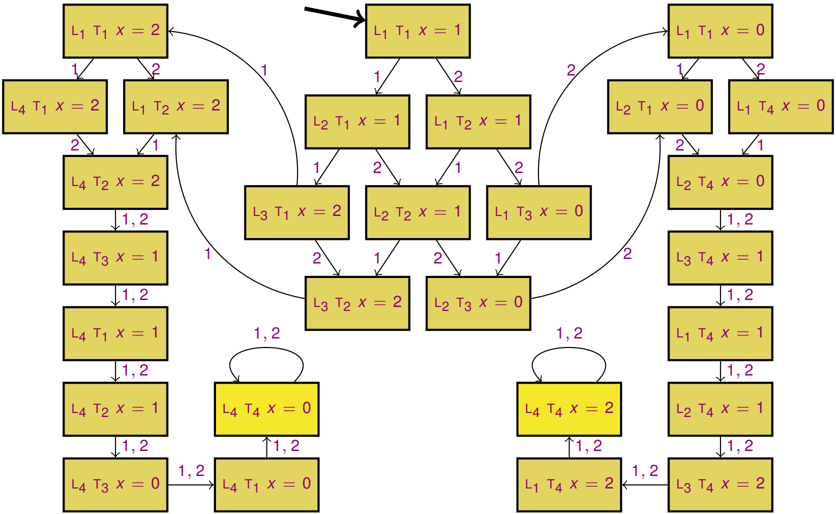
$$\phi_{spec} = AF(L_4 \wedge T_4 \wedge (x = 0 \vee x = 2))$$



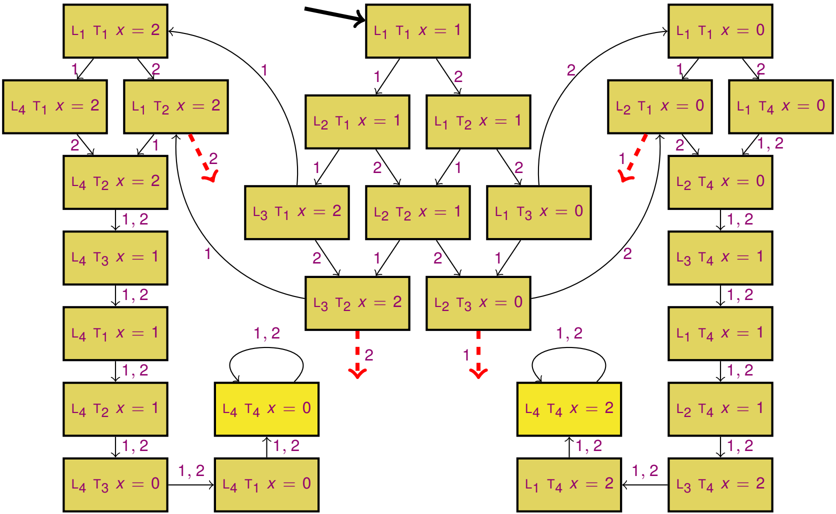
Extraction of Model M from T_ϕ

- Model-fragment:
  - Rooted DAG of AND-nodes in T_ϕ
  - All eventualities in root node fulfilled
- Identify model fragments rooted at each AND-node of T_ϕ
- If multiple model fragments, pick one of *minimal size*
- Join model fragments to get model

Extraction of Model M from T_ϕ



Obtaining P_1^s, P_2^s from M



Obtaining P_1^s, P_2^s from M

P :
  $x : \{0, 1, 2\}$ with $x = 1$
  $P_1 \parallel P_2$

P_1^s :

```
L1 :  ⟨ if (x < 2) L2, L4 ⟩;
L2 :     ⟨ ¬((x = 0) ∧ (T1 ∨ T3))
         → x := x + 1 ⟩;
L3 :      ⟨ goto L1 ⟩;
L4 :  ⟨ goto L4 ⟩;
```

P_2^s :

```
T1 :  ⟨ if (x > 0) T2, T4 ⟩;
T2 :     ⟨ ¬((x = 2) ∧ (L1 ∨ L3))
         → x := x − 1 ⟩;
T3 :      ⟨ goto T1 ⟩;
T4 :  ⟨ goto T4 ⟩;
```

$$AF(L_4 \wedge T_4 \wedge (x = 0 \vee x = 2))$$

Assumptions, Correctness, Complexity

[Assumptions]:

- P_1, P_2 are finite-state programs over **L**
- All program variables are shared variables, initialized to specific values
- **L**-atoms and **L**-terms can be evaluated

[Soundness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, if the algorithm generates P^s, then $P^s \models \phi_{spec}$.

[Completeness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, the algorithm generates P^s such that $P^s \models \phi_{spec}$ if $\phi = \phi_{spec} \wedge \phi_P$ is satisfiable.

[Complexity]: Exponential in the size of ϕ_{spec} and P variables.

Assumptions, Correctness, Complexity

[Assumptions]:

- P_1, P_2 are finite-state programs over **L**
- All program variables are shared variables, initialized to specific values
- **L**-atoms and **L**-terms can be evaluated

[Soundness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, if the algorithm generates P^s, then $P^s \models \phi_{spec}$.

[Completeness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, the algorithm generates P^s such that $P^s \models \phi_{spec}$ if $\phi = \phi_{spec} \wedge \phi_P$ is satisfiable.

[Complexity]: Exponential in the size of ϕ_{spec} and P variables.

Assumptions, Correctness, Complexity

[Assumptions]:

- P_1, P_2 are finite-state programs over **L**
- All program variables are shared variables, initialized to specific values
- **L**-atoms and **L**-terms can be evaluated

[Soundness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, if the algorithm generates P^s, then $P^s \models \phi_{spec}$.

[Completeness]:
Given unsynchronized processes P_1, P_2 and an **L**CTL formula ϕ_{spec}, the algorithm generates P^s such that $P^s \models \phi_{spec}$ if $\phi = \phi_{spec} \wedge \phi_P$ is satisfiable.

[Complexity]: Exponential in the size of ϕ_{spec} and P variables.

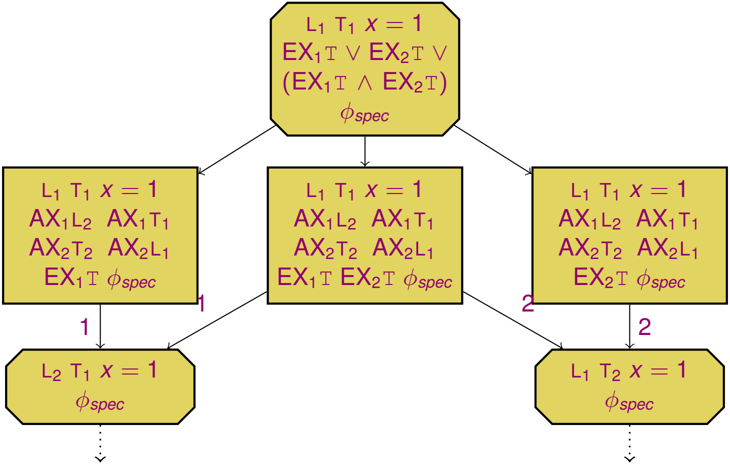
Uninitialized Variables

$$\phi_{spec} = \mathsf{AF}(\mathsf{L}_4 \wedge \mathsf{T}_4 \wedge (x = 0 \vee x = 2))$$

$\mathsf{L}_1\ \mathsf{T}_1\ x = 1$
$\mathsf{EX}_1\mathtt{T} \vee \mathsf{EX}_2\mathtt{T} \vee$
$(\mathsf{EX}_1\mathtt{T} \wedge \mathsf{EX}_2\mathtt{T})$
ϕ_{spec}

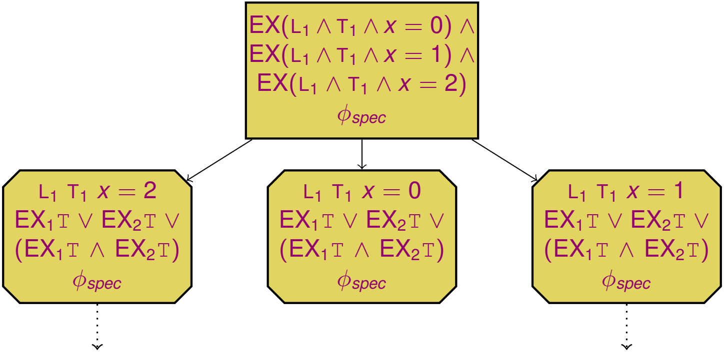
Uninitialized Variables

$$\phi_{spec} = \mathsf{AF}(\mathsf{L}_4 \wedge \mathsf{T}_4 \wedge (x = 0 \vee x = 2))$$

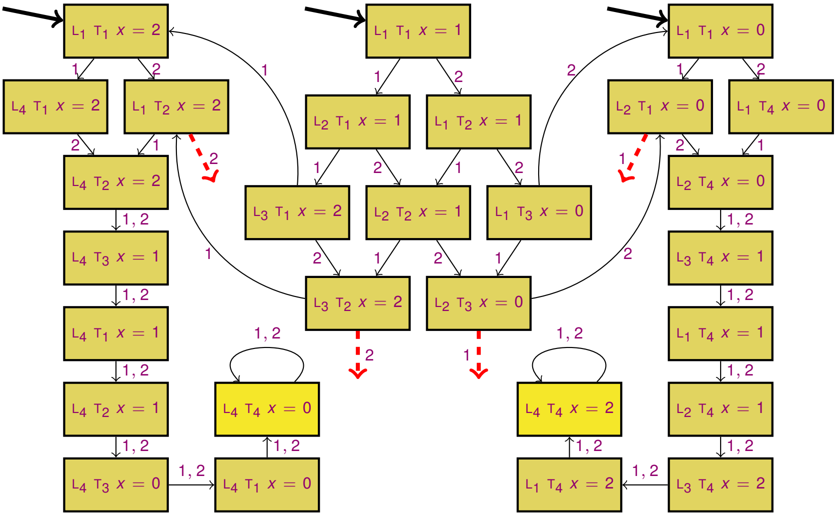


Uninitialized Variables

$$\phi_{spec} = \mathsf{AF}(\mathsf{L}_4 \wedge \mathsf{T}_4 \wedge (x = 0 \vee x = 2))$$



Uninitialized Variables



Observability

P :

$x : \{0, 1, 2\}$ with $x = 1$

$P_1 \parallel P_2$

P_1^s :

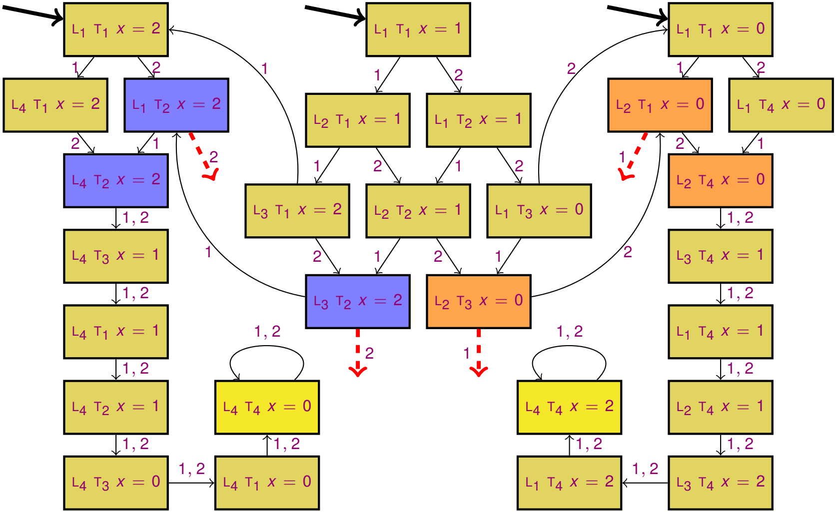
```
L1:  ⟨ if (x < 2) L2, L4 ⟩;
L2:     ⟨ ¬((x = 0) ∧ (T1 ∨ T3))
        → x := x + 1 ⟩;
L3:     ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2^s :

```
T1:  ⟨ if (x > 0) T2, T4 ⟩;
T2:     ⟨ ¬((x = 2) ∧ (L1 ∨ L3))
        → x := x − 1 ⟩;
T3:     ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

$$AF(L_4 \wedge T_4 \wedge (x = 0 \vee x = 2))$$

Observability



Observability

P :

$x : \{0,1,2\}$ with $x = 1$
t : bool with $t = 1$
l : bool with $l = 1$
$P_1 \parallel P_2$

P_1^s :

```
L1:  ⟨ if (x < 2) L2, L4;
         l := 0 ⟩;
L2:    ⟨ ¬((x = 0) ∧ (t = 1))
         → x := x + 1;  l := 1 ⟩;
L3:    ⟨ goto L1 ⟩;
L4:  ⟨ goto L4 ⟩;
```

P_2^s :

```
T1:  ⟨ if (x > 0) T2, T4;
         t := 0 ⟩;
T2:    ⟨ ¬((x = 2) ∧ (l = 1))
         → x := x − 1;  t := 1 ⟩;
T3:    ⟨ goto T1 ⟩;
T4:  ⟨ goto T4 ⟩;
```

$$AF(L_4 \wedge T_4 \wedge (x = 0 \vee x = 2))$$

Related Work

- Inference of high-level synchronization, guarded commands: [EC82, VYY09]
- Synthesis of low-level synchronization for relaxed memory models: [KVV10]
- Sketching: [S-LRBE05]
- Open systems [PR89]

Ongoing Work

- Efficient tableau construction?
- Infinite-state programs?
- Pleasant guards?
- Other synchronization primitives?