Motivation

We need a language for specifying system properties.

- A system $S$ is a pair $\langle I, R \rangle$.
  - Initial states $I$, transition relation $R$.
  - More intuitive: reachability graph.
    - Starting from an initial state $s_0$, the system runs evolve.
- Consider the reachability graph as an infinite computation tree.
  - Different tree nodes may denote occurrences of the same state.
    - Each occurrence of a state has a unique predecessor in the tree.
  - Every path in this tree is infinite.
    - Every finite run $s_0 \rightarrow \ldots \rightarrow s_n$ is extended to an infinite run $s_0 \rightarrow \ldots \rightarrow s_n \rightarrow s_n \rightarrow \ldots$
- Or simply consider the graph as a set of system runs.
  - Same state may occur multiple times (in one or in different runs).

Temporal logic describes such trees respectively sets of system runs.

---

Computation Trees versus System Runs


Set of system runs:

- $[a, b] \rightarrow c \rightarrow c \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow c \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots$
  ...
State Formula

Temporal logic is based on classical logic.
- **A state formula** \( F \) is evaluated on a state \( s \).
  - Any predicate logic formula is a state formula:
    \[ p(x), \neg F, F_0 \land F_1, F_0 \lor F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1, \forall x : F, \exists x : F. \]
  - In **propositional temporal logic** only propositional logic formulas are state formulas (no quantification):
    \[ p, \neg F, F_0 \land F_1, F_0 \lor F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1. \]
- **Semantics**: \( s \models F \) ("\( F \) holds in state \( s \)).
  - Example: semantics of conjunction.
    - \( (s \models F_0 \land F_1) \Leftrightarrow (s \models F_0) \land (s \models F_1). \)
    - "\( F_0 \land F_1 \) holds in \( s \) if and only if \( F_0 \) holds in \( s \) and \( F_1 \) holds in \( s \)."

Classical logic reasoning on individual states.

Temporal Logic

Extension of classical logic to reason about multiple states.
- **Temporal logic** is an instance of modal logic.
  - Logic of "multiple worlds (situations)" that are in some way related.
  - Relationship may e.g. be a temporal one.
  - Amir Pnueli, 1977: temporal logic is suited to system specifications.
  - Many variants, two fundamental classes.
- **Branching Time Logic**
  - Semantics defined over computation trees.
    At each moment, there are multiple possible futures.
  - Prominent variant: \( \text{CTL} \).
    Computation tree logic; a propositional branching time logic.
- **Linear Time Logic**
  - Semantics defined over sets of system runs.
    At each moment, there is only one possible future.
  - Prominent variant: \( \text{PLTL} \).
    A propositional linear time logic.

State Formulas

We have additional state formulas.
- **A state formula** \( F \) is evaluated on state \( s \) of System \( S \).
  - Every (classical) state formula \( f \) is such a state formula.
  - Let \( P \) denote a path formula (later).
    - Evaluated on a path (state sequence) \( p = p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow ... \).
    - \( R(p_i, p_{i+1}) \) for every \( i \); \( p_0 \) need not be an initial state.
  - Then the following are state formulas:
    - \( \mathbf{A} P \) ("in every path \( P \)."
    - \( \mathbf{E} P \) ("in some path \( P \)."
- **Path quantifiers**: \( \mathbf{A}, \mathbf{E} \).
  - Semantics: \( S, s \models F \) ("\( F \) holds in state \( s \) of system \( S \))."
    - \( S, s \models f : \Leftrightarrow s \models f. \)
    - \( S, s \models \mathbf{A} P : \Leftrightarrow S, p \models P, \) for every path \( p \) of \( S \) with \( p_0 = s. \)
    - \( S, s \models \mathbf{E} P : \Leftrightarrow S, p \models P, \) for some path \( p \) of \( S \) with \( p_0 = s. \)
Path Formulas

We have a class of formulas that are not evaluated over individual states.

- A path formula \( P \) is evaluated on a path \( p \) of system \( S \).
- Let \( F \) and \( G \) denote state formulas.
- Then the following are path formulas:
  - \( X F \) (“next time \( F \)”).
  - \( G F \) (“always \( F \)”).
  - \( F F \) (“eventually \( F \)”).
  - \( F U G \) (“\( F \) until \( G \)”).

- Temporal operators: \( X, G, F, U \).

- Semantics:
  \[ S, p |\models X F :\Leftrightarrow S, p_{i+1} |\models F. \]
  \[ S, p |\models G F :\Leftrightarrow \forall i \in \mathbb{N} : S, p_i |\models F. \]
  \[ S, p |\models F F :\Leftrightarrow \exists i \in \mathbb{N} : S, p_i |\models F. \]
  \[ S, p |\models F U G :\Leftrightarrow \exists i \in \mathbb{N} : S, p_i |\models G \land \forall j \in \mathbb{N} : S, p_j |\models F. \]

---

Linear Time Logic (LTL)

We use temporal logic to specify a system property \( P \).

- Core question: \( S |\models P \) (“\( P \) holds in system \( S \)”).
- System \( S = \langle I, R \rangle \), temporal logic formula \( P \).
- Linear time logic:
  - \( S |\models P :\Leftrightarrow r |\models P \), for every run \( r \) of \( S \).
  - Property \( P \) must be evaluated on every run \( r \) of \( S \).
  - Given a computation tree with root \( s_0 \), \( P \) is evaluated on every path of that tree originating in \( s_0 \).
  - If \( P \) holds for every path, \( P \) holds on \( S \).

LTL formulas are evaluated on system runs.
Formulas

No path quantifiers; all formulas are path formulas.

- Every formula is evaluated on a path \( p \).
- Also every state formula \( f \) of classical logic (see below).
- Let \( F \) and \( G \) denote formulas.
- Then also the following are formulas:
  - \( X F \) ("next time \( F \"), often written \( \Diamond F \),
  - \( G F \) ("always \( F \"), often written \( \Box F \),
  - \( F F \) ("eventually \( F \"), often written \( \Diamond F \),
  - \( F U G \) ("\( F \) until \( G \")

Semantics: \( p \models P \) ("\( P \) holds in path \( p \)).

\[
p^i := \langle p_1, p_{i+1}, \ldots \rangle \ni \models P \implies p_0 \models f.
\]

\[
p^i \models F \implies p_{i+1} \models F.
\]

\[
p^i \models G F \iff \forall i \in \mathbb{N} : p_i \models F.
\]

\[
p^i \models F F \iff \exists i \in \mathbb{N} : p_i \models F.
\]

\[
p^i \models F U G \iff \exists i \in \mathbb{N} : p_i \models G \land \forall j \in \mathbb{N} i : p_j \models F.
\]

Branching versus Linear Time Logic

We use temporal logic to specify a system property \( P \).

- Core question: \( S \models P \) ("\( P \) holds in system \( S \")).
- System \( S = \langle I, R \rangle \), temporal logic formula \( P \).
- Branching time logic:
  - \( S \models P :\iff \forall s_0 \models P \), for every initial state \( s_0 \) of \( S \).
  - Property \( P \) must be evaluated on every pair \( (S, s_0) \) of system \( S \) and
    initial state \( s_0 \).
  - Given a computation tree with root \( s_0 \), \( P \) is evaluated on that tree.
- Linear time logic:
  - \( S \models P :\iff \forall r \models P \), for every run \( r \) of \( S \).
  - Property \( P \) must be evaluated on every run \( r \) of \( S \).
  - Given a computation tree with root \( s_0 \), \( P \) is evaluated on every path
    of that tree originating in \( s_0 \).
  - If \( P \) holds for every path, \( P \) holds on \( S \).

Linear time logic: both systems have the same runs.
- Thus every formula has same truth value in both systems.

Branching time logic: the systems have different computation trees.
- Take formula \( \Diamond \Box (\Box Q \land \Box \neg Q) \).
- True for left system, false for right system.

The two variants of temporal logic have different expressive power.
Branching versus Linear Time Logic

Is one temporal logic variant more expressive than the other one?
- CTL formula: $\text{AG} (\text{EF } F)$.
  - “In every run, it is at any time still possible that later $F$ will hold”.
  - Property cannot be expressed by any LTL logic formula.
- LTL formula: $\Box \text{EF } F$ (i.e. $\text{FG } F$).
  - “In every run, there is a moment from which on $F$ holds forever.”.
  - Naive translation $\text{AFAG } F$ is not a CTL formula.
  - $\text{G } F$ is a path formula, but $F$ expects a state formula!
  - Translation $\text{AFAG } F$ expresses a stronger property (see next page).
  - Property cannot be expressed by any CTL formula.

None of the two variants is strictly more expressive than the other one; no variant can express every system property.

Property cannot be expressed by any CTL formula.


Wolfgang Schreiner
http://www.risc.jku.at

17/65

Proof that $\text{AFAG } F$ (CTL) is different from $\Box F$ (LTL).

AFAG $F$ <-> lâise
$\neg \Box F$ <-> true

In every run, there is a moment when it is guaranteed that from now on $F$ holds forever.
In every run, there is a moment from which on $F$ holds forever.

Wolfgang Schreiner
http://www.risc.jku.at

18/65

Linear Time Logic

Why using linear time logic (LTL) for system specifications?
- LTL has many advantages:
  - LTL formulas are easier to understand.
  - Reasoning about computation paths, not computation trees.
  - No explicit path quantifiers used.
  - LTL can express most interesting system properties.
    - Invariance, guarantee, response, … (see later).
  - LTL can express fairness constraints (see later).
    - CTL cannot do this.
    - But CTL can express that a state is reachable (which LTL cannot).
- LTL has also some disadvantages:
  - LTL is strictly less expressive than other specification languages.
    - CTL* or $\mu$-calculus.
  - Asymptotic complexity of model checking is higher.
    - LTL: exponential in size of formula; CTL: linear in size of formula.
  - In practice the number of states dominates the checking time.
Examples

- Mutual exclusion: \( \square \neg (pc_1 = C \land pc_2 = C) \).
  - Alternatively: \( \neg \diamond (pc_1 = C \land pc_2 = C) \).
  - Never both components are simultaneously in the critical region.

- No starvation: \( \forall i : \square (pc_i = W) \Rightarrow \diamond pc_i = R) \).
  - Always, if component \( i \) waits for a response, it eventually receives it.

- No deadlock: \( \square \forall i : pc_i = W \).
  - Never all components are simultaneously in a wait state \( W \).

- Precedence: \( \forall i : \square (pc_i \neq C) \Rightarrow (pc_i \neq C \lor lock = i)) \).
  - Always, if component \( i \) is out of the critical region, it stays out until it receives the shared lock variable (which it eventually does).

- Partial correctness: \( \square (pc = L \Rightarrow C) \).
  - Always if the program reaches line \( L \), the condition \( C \) holds.

- Termination: \( \forall i : \diamond (pc_i = T) \).
  - Every component eventually terminates.

Temporal Rules

Temporal operators obey a number of fairly intuitive rules.

- Extraction laws:
  - \( \square F \iff F \land \square F \).
  - \( \diamond F \iff F \lor \diamond F \).
  - \( F U G \iff G \lor (F \land \diamond (F U G)) \).

- Negation laws:
  - \( \neg \square F \iff \diamond \neg F \).
  - \( \neg \diamond F \iff \square \neg F \).
  - \( \neg (F U G) \iff (\neg G) U (\neg F \land \neg G) \lor \neg \diamond G \).

- Distributivity laws:
  - \( \square (F \land G) \iff (\square F) \land (\square G) \).
  - \( \diamond (F \lor G) \iff (\diamond F) \lor (\diamond G) \).
  - \( (F \land G) U H \iff (F U H) \land (G U H) \).
  - \( F U (G \lor H) \iff (F U G) \lor (F U H) \).
  - \( \square \diamond (F \lor G) \iff (\square \diamond F) \lor (\square \diamond G) \).
  - \( \diamond \diamond (F \land G) \iff (\diamond \diamond F) \land (\diamond \diamond G) \).
Classes of System Properties

There exist two important classes of system properties.

- **Safety Properties:**
  - A safety property is a property such that, if it is violated by a run, it is already violated by some finite prefix of the run.
  - This finite prefix cannot be extended in any way to a complete run satisfying the property.
  - Example: $\Diamond F$ (with state property $F$).
  - The violating run $F \rightarrow F \rightarrow \neg F \rightarrow \ldots$ has the prefix $F \rightarrow F \rightarrow \neg F$ that cannot be extended in any way to a run satisfying $\Box F$.

- **Liveness Properties:**
  - A liveness property is a property such that every finite prefix can be extended to a complete run satisfying this property.
  - Only a complete run itself can violate that property.
  - Example: $\Box F$ (with state property $F$).
  - Any finite prefix $p$ can be extended to a run $p \rightarrow F \rightarrow \ldots$ which satisfies $\Diamond F$.

Verifying Safety

We only consider a special case of a safety property.

- **Theorem:**
  - Every system property $P$ is a conjunction $S \land L$ of some safety property $S$ and some liveness property $L$.
  - If $L$ is “true”, then $P$ itself is a safety property.
  - If $S$ is “true”, then $P$ itself is a liveness property.

- **Consequence:**
  - Assume we can decompose $P$ into appropriate $S$ and $L$.
  - For verifying $M \models P$, it then suffices to verify:
    - **Safety:** $M \models S$.
    - **Liveness:** $M \models L$.
  - Different strategies for verifying safety and liveness properties.

For verification, it is important to decompose a system property in its “safety part” and its “liveness part”.

System Properties

The real importance of the distinction is stated by the following theorem.

- **Theorem:**
  - Every system property $P$ is a conjunction $S \land L$ of some safety property $S$ and some liveness property $L$.

- **Consequence:**
  - If $L$ is “true”, then $P$ itself is a safety property.
  - If $S$ is “true”, then $P$ itself is a liveness property.

- **Verification:**
  - For verifying $M \models P$, it then suffices to verify:
    - **Safety:** $M \models S$.
    - **Liveness:** $M \models L$.
  - Different strategies for verifying safety and liveness properties.

Not every system property is itself a safety property or a liveness property.

- **Example:** $P :\iff (\Box A) \land (\Diamond B)$ (with state properties $A$ and $B$)
  - Conjunction of a safety property and a liveness property.
  - Take the run $[A, \neg B] \rightarrow [A, \neg B] \rightarrow [A, \neg B] \rightarrow \ldots$ violating $P$.
  - Any prefix $[A, \neg B] \rightarrow \ldots$ of this run can be extended to a run $[A, \neg B] \rightarrow \ldots [A, \neg B] \rightarrow [A, B] \rightarrow [A, B] \rightarrow \ldots$.
  - Thus $P$ is not a safety property.
  - Take the finite prefix $[\neg A, B]$.
  - This prefix cannot be extended in any way to a run satisfying $P$.
  - Thus $P$ is not a liveness property.

So is the distinction “safety” versus “liveness” really useful?
Example

var x := 0
loop
    p₀ : wait x = 0
    p₁ : x := x + 1
∥
    q₀ : wait x = 1
    q₁ : x := x − 1

State = {p₀, p₁} × {q₀, q₁} × Z.

I((p, q, x)) ⇔ p = p₀ ∧ q = q₀ ∧ x = 0.
R((p, q, x), (p’, q’, x’)) ⇔ P₀(... ∨ P₁(... ∨ Q₀(... ∨ Q₁(....).

P₀((p, q, x), (p’, q’, x’)) ⇔ p = p₀ ∧ q = q₀ ∧ x = 0 ∧ p’ = p₁ ∧ q’ = q ∧ x’ = x.
P₁((p, q, x), (p’, q’, x’)) ⇔ p = p₁ ∧ p’ = p₀ ∧ q’ = q ∧ x’ = x + 1.
Q₀((p, q, x), (p’, q’, x’)) ⇔ q = q₀ ∧ x = 1 ∧ p’ = p ∧ q’ = q₁ ∧ x’ = x.
Q₁((p, q, x), (p’, q’, x’)) ⇔ q = q₁ ∧ p’ = p ∧ q’ = q₀ ∧ x’ = x − 1.

Prove ⟨I, R⟩ |= □(x = 0 ∨ x = 1).

Verifying Liveness

var x := 0, y := 0
loop
    x := x + 1
∥
    y := y + 1

State = N × N; Label = {p, q}.
I(x, y) := x = 0 ∧ y = 0.
R(I, (x, y), (x’, y’)) :=
    (I = p ∧ x’ = x + 1 ∧ y’ = y) ∨ (I = q ∧ x’ = x ∧ y’ = y + 1).

⟨I, R⟩ \not\models □x = 1.
    [x = 0, y = 0] → [x = 0, y = 1] → [x = 0, y = 2] → ...
    This run violates (as the only one) □x = 1.
    Thus the system as a whole does not satisfy □x = 1.

For verifying liveness properties, “unfair” runs have to be ruled out.

Inductive System Properties

The induction strategy may not work for proving □F

- Problem: F is not inductive.
  - F is too weak to prove the induction step.
    F(s) ∧ R(s, s’) ⇒ F(s’).
  - Solution: find stronger invariant I.
    If I ⇒ F, then (□I) ⇒ (□F).
  - It thus suffices to prove □I.
- Rationale: I may be inductive.
  - If yes, I is strong enough to prove the induction step.
    I(s) ∧ R(s, s’) ⇒ I(s’).
  - If not, find a stronger invariant I’ and try again.
- Invariant I represents additional knowledge for every proof.
  - Rather than proving □P, prove □(I ⇒ P).

The behavior of a system is captured by its strongest invariant.

Example

Prove ⟨I, R⟩ |= □G.

G ⇐⇒
    (x = 0 ∨ x = 1) ∧
    (p = p₁ ⇒ x = 0) ∧
    (q = q₁ ⇒ x = 1).

Proof works.
G ⇒ (x = 0 ∨ x = 1) obvious.

See the proof presented in class.
Enabling Condition

When is a particular transition enabled for execution?

- **Enabled** \(_R(l, s) :⇔ \exists t : R(l, s, t)\).  
  - Labeled transition relation \(R\), label \(l\), state \(s\).
  - Read: "Transition (with label) \(l\) is enabled in state \(s\) (w.r.t. \(R\))".

- **Example** (previous slide):
  \[
  \text{Enabled}_R(p, (x, y)) \\
  \Leftrightarrow \exists x', y' : R(p, (x, y), (x', y')) \\
  \Leftrightarrow \exists x', y' : \\
  (p = p \land x' = x + 1 \land y' = y) \lor \\
  (p = q \land x' = x \land y' = y + 1) \\
  \Leftrightarrow (\exists x', y' : p = p \land x' = x + 1 \land y' = y) \lor \\
  (\exists x', y' : p = q \land x' = x \land y' = y + 1) \\
  \Leftrightarrow \text{true} \lor \text{false} \\
  \Leftrightarrow \text{true}.
  \]

- Transition \(p\) is always enabled.

Weak Fairness

- **Weak Fairness**
  
  - A run \(s_0 \xrightarrow{h_1} s_1 \xrightarrow{h_2} \ldots\) is weakly fair to a transition \(l\), if
    - if transition \(l\) is eventually permanently enabled in the run,
    - then transition \(l\) is executed infinitely often in the run.
    
    \((\exists i : \forall j \geq i : \text{Enabled}_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l = l)\).
  
  - The run in the previous example was not weakly fair to transition \(p\).

- LTL formulas may explicitly specify weak fairness constraints.

  - Let \(E_l\) denote the enabling condition of transition \(l\).
  - Let \(X_l\) denote the predicate "transition \(l\) is executed".
  - Define \(WF_l :⇔ (\Diamond \Box E_l) \Rightarrow (\Box \Diamond X_l)\).

  - If \(l\) is eventually enabled forever, it is executed infinitely often.

- Prove \((l, R) \models (WF_l \Rightarrow P)\).

  - Property \(P\) is only proved for runs that are weakly fair to \(l\).

Alternatively, a model may also have weak fairness "built in".

Example

\[
\begin{align*}
\text{State} & = \mathbb{N} \times \mathbb{N}; \text{Label} = \{p, q\}. \\
I(x, y) & :⇔ x = 0 \land y = 0. \\
R(l, (x, y), (x', y')) & :⇔ \\
(l = p \land x' = x + 1 \land y' = y) \lor (l = q \land x' = x \land y' = y + 1).
\end{align*}
\]

- \((l, R) \models WF_p \Rightarrow \Diamond x = 1\).
  - \([x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \ldots\)

  - This (only) violating run is not weakly fair to transition \(p\).
    - \(p\) is always enabled.
    - \(p\) is never executed.

System satisfies specification if weak fairness is assumed.

Strong Fairness

- **Strong Fairness**
  
  - A run \(s_0 \xrightarrow{h_1} s_1 \xrightarrow{h_2} \ldots\) is strongly fair to a transition \(l\), if
    - if \(l\) is infinitely often enabled in the run,
    - then \(l\) is also infinitely often executed the run.
    
    \((\forall i : \exists j \geq i : \text{Enabled}_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l = l)\).
  
  - If \(r\) is strongly fair to \(l\), it is also weakly fair to \(l\) (but not vice versa).

- LTL formulas may explicitly specify strong fairness constraints.

  - Let \(E_l\) denote the enabling condition of transition \(l\).
  - Let \(X_l\) denote the predicate "transition \(l\) is executed".
  - Define \(SF_l :⇔ (\Box \Diamond E_l) \Rightarrow (\Diamond \Box X_l)\).

  - If \(l\) is enabled infinitely often, it is executed infinitely often.

- Prove \((l, R) \models (SF_l \Rightarrow P)\).

  - Property \(P\) is only proved for runs that are strongly fair to \(l\).

A much stronger requirement to the fairness of a system.
Weak versus Strong Fairness

In which situations is which notion of fairness appropriate?

- **Process just waits to be scheduled for execution.**
  - Only CPU time is required.
  - Weak fairness suffices.
- **Process waits for resource that may be temporarily blocked.**
  - Critical region protected by lock variable (mutex/semaphore).
  - Strong fairness is required.
- **Non-deterministic choices are repeatedly made in program.**
  - Simultaneous listing on multiple communication channels.
  - Strong fairness is required.

Many other notions or fairness exist.

### Example

\[
\text{var } x=0 \\
\text{loop} \\
\quad a: x := -x \\
\quad b: \text{choose } x := 0 \land x := 1 \\
\text{State} := \{a,b\} \times \mathbb{Z}; \text{Label} = \{A,B_0,B_1\}. \\
I(p,x) :\Leftrightarrow p = a \land x = 0. \\
R(\langle l, (p,x) \rangle, (p',x')) :\Leftrightarrow \\
(\langle l = A \land (p = a \land p' = b \land x' = -x) \rangle \lor \\
(\langle l = B_0 \land (p = b \land p' = a \land x' = 0) \rangle \lor \\
(\langle l = B_1 \land (p = b \land p' = a \land x' = 1) \rangle.
\]

\[\langle I, R \rangle \models SF_{B_1} \Rightarrow \Diamond x = 1.\]

\[\vdash [a,0] A [b,0] B_1 [a,0] A [b,0] B_1 [a,0] A \ldots\]

This (only) violating run is *not strongly fair* to \(B_1\) (but weakly fair).

- \(B_1\) is infinitely often enabled.
- \(B_1\) is never executed.

System satisfies specification if strong fairness is assumed.

--

A Bit Transmission Protocol

\[
\begin{align*}
S &: \text{loop} \\
&\quad \text{choose } x \in \{0,1\} \\
&\quad 1: v, r := x, 1 \\
&\quad 2: \text{wait } a = 1 \\
&\quad r := 0 \\
&\quad 3: \text{wait } a = 0 \\
\end{align*}
\]

\[
\begin{align*}
R &: \text{loop} \\
&\quad 1: \text{wait } r = 1 \\
&\quad y, a := v, 1 \\
&\quad 2: \text{wait } r = 0 \\
&\quad a := 0 \\
\end{align*}
\]

Transmit a sequence of bits through a wire.
A (Simplified) Model of the Protocol

State := $PC^2 \times (\mathbb{N}_2)^5$

$I(p, q, x, y, v, r, a) \iff p = q = 1 \land x \in \mathbb{N}_2 \land v = r = a = 0$.

Invariant(p, ... ) $\Rightarrow$ (q = 2 $\Rightarrow$ y = x)

A Verification Task

\[ I(p, ... ) \Rightarrow \text{Invariant}(p, ... ) \]

\[ R((p, ... ), (p', ... )) \land \text{Invariant}(p, ... ) \Rightarrow \text{Invariant}(p', ... ) \]

The invariant captures the essence of the protocol.

The RISC ProofNavigator Theory

Init: BOOLEAN =

$\forall p, q, x, y, v, r, a: p = 1 \land q = 1 \land (x = 0 \lor x = 1) \land v = 0 \land r = 0 \land a = 0$.

Step: BOOLEAN =

$S1 \lor S2 \lor S3 \lor R1 \lor R2$;

Invariant: (NAT, NAT, NAT, NAT, NAT, NAT, NAT, NAT) $\Rightarrow$ BOOLEAN =

LAM(p, q, x, y, v, r, a: NAT):

$\forall p, q, x, y, v, r, a: p = 1 \lor p = 2 \lor p = 3 \land (q = 1 \lor q = 2) \land (x = 0 \lor x = 1) \land (v = 0 \lor v = 1) \land (r = 0 \lor r = 1) \land (a = 0 \lor a = 1) \land (p = 1 \Rightarrow q = 1 \land r = 0 \land a = 0) \land (p = 2 \Rightarrow r = 1 \land x = x) \land (p = 3 \Rightarrow r = 0) \land (q = 1 \Rightarrow a = 0) \land (q = 2 \Rightarrow (p = 2 \lor p = 3) \land a = 1 \land y = x)$.
The RISC ProofNavigator Theory

Property: BOOLEAN = 
q = 2 => y = x;

VC0: FORMULA
Invariant(p, q, x, y, v, r, a) => Property;

VC1: FORMULA
Init => Invariant(p, q, x, y, v, r, a);

VC2: FORMULA
Step AND Invariant(p, q, x, y, v, r, a) => Invariant(p0, q0, x0, y0, v0, r0, a0);

The Proofs

[vd2]: expand Invariant, Property in m2v
[rlc]: proved (CVCL)

[wd2]: expand Init, Invariant in nra
[ip1]: proved (CVCL)

[xd2]: expand Step, Invariant, S1, S2, S3, R1, R2
[6ss]: proved (CVCL)

More instructive: proof attempts with wrong or too weak invariants (see demonstration).

A Client/Server System (Contd)

Server system $S = (IS, RS)$.

State := $(N_3)^3 \times \{(1, 2) \rightarrow N_2\}^2$.

Int := \{(D1, D2, F, A1, A2, W)\}.

I5(given, waiting, sender, rbuffer, sbuffer) \iff given = waiting = sender = 0 \land rbuffer(1) = rbuffer(2) = sbuffer(1) = sbuffer(2) = 0.

RS(l, \{(given, waiting, sender, rbuffer, sbuffer)\}).

\forall i \in \{1, 2\}:

\begin{align*}
& (l = D1 \land \text{sender} = 0 \land \text{rbuffer}(i) \neq 0 \land \\
& \text{sender}' = i \land \text{rbuffer}'(i) = 0 \land \\
& \text{U}(\text{given}, \text{waiting}, \text{sender}) \land \\
& \forall j \in \{1, 2\} \setminus \{i\} : U_j(\text{rbuffer}) \lor \\
& \ldots
\end{align*}

Server:
local given, waiting, sender
begin
given := 0; waiting := 0
loop
D: sender := receiveRequest()
if sender = given then
if waiting = 0 then
else
A1: given := waiting;
waiting := 0
sendAnswer(given)
endif
else
given := 0
else
A2: given := sender
sendAnswer(given)
endif
endif
end loop

W: waiting := sender
end if
end loop
A Client/Server System (Contd’2)

\[(l = F \land \text{sender} \neq 0 \land \text{sender} = \text{given} \land \text{waiting} = 0 \land \text{given'} = 0 \land \text{sender'} = 0 \land \text{U}(\text{waiting}, \text{rbuffer}, \text{sbuffer})) \lor \]

\[(l = A1 \land \text{sender} \neq 0 \land \text{sbuffer}(\text{waiting}) = 0 \land \text{sender} = \text{given} \land \text{waiting} = 0 \land \text{given'} = 0 \land \text{sender'} = 0 \land \text{U}(\text{rbuffer}) \land \forall i \in \{1, 2\} \setminus \{\text{sender}\} : U_i(\text{sbuffer})) \lor \]

\[(l = A2 \land \text{sender} \neq 0 \land \text{sbuffer}(\text{sender}) = 0 \land \text{sender} \neq \text{given} \land \text{given} = 0 \land \text{given'} = 0 \land \text{sender'} = 0 \land \text{U}(\text{waiting}, \text{rbuffer}) \land \forall i \in \{1, 2\} \setminus \{\text{sender}\} : U_i(\text{sbuffer})) \lor \]

\[
\text{if sender = given then if waiting = 0 then sendAnswer(given) else given = 0 then sendAnswer(given)}
\]

\[
\text{D: sender := receiveRequest()}
\]

\[
\text{if waiting = 0 then sendAnswer(given) else given = 0 then sendAnswer(given)}
\]

\[
\forall i \in \{1, 2\} : \forall j \in \{1, 2\} \setminus \{i\} : U_j(\text{rbuffer}) \cup
\]

\[
\begin{align*}
& (l = \text{REQ} \land \text{rbuffer'}(i) = 1 \land \text{U}(\text{given}, \text{waiting}, \text{sender}, \text{sbuffer}) \land \\
& \forall j \in \{1, 2\} \setminus \{i\} : \text{U}_j(\text{rbuffer})) \lor \\
& (l = \text{ANS} \land \text{sbuffer'}(i) \neq 0 \land \\
& \text{U}(\text{given}, \text{waiting}, \text{sender}, \text{rbuffer}) \land \\
& \forall j \in \{1, 2\} \setminus \{i\} : \text{U}_j(\text{sbuffer})).
\end{align*}
\]

A Client/Server System (Contd’3)

Server:

\[
\text{local given, waiting, sender begin}
\]

\[
\text{given := 0; waiting := 0 loop}
\]

\[
\text{D: sender := receiveRequest() if sender = given then if waiting = 0 then sendAnswer(given) else given = 0 then sendAnswer(given)}
\]

\[
\text{D: sendAnswer(given)}
\]

\[
\text{else given = 0 then sendAnswer(given)}
\]

\[
\begin{align*}
& \exists i \in \{1, 2\} : \\
& (l = \text{REQ} \land \text{rbuffer'}(i) = 1 \land \\
& \text{U}(\text{given}, \text{waiting}, \text{sender}, \text{sbuffer}) \land \\
& \forall j \in \{1, 2\} \setminus \{i\} : \text{U}_j(\text{rbuffer})) \lor \\
& (l = \text{ANS} \land \text{sbuffer'}(i) \neq 0 \land \\
& \text{U}(\text{given}, \text{waiting}, \text{sender}, \text{rbuffer}) \land \\
& \forall j \in \{1, 2\} \setminus \{i\} : \text{U}_j(\text{sbuffer})).
\end{align*}
\]

A Client/Server System (Contd’4)

Server:

\[
\text{local given, waiting, sender begin}
\]

\[
\text{given := 0; waiting := 0 loop}
\]

\[
\text{D: sendAnswer(given)}
\]

\[
\text{endif endloop end Server}
\]

\[
\text{The Verification Task}
\]

\[
(I, R) \models \Box \neg (pc_1 = C \land pc_2 = C)
\]

\[
\text{Invariant(pc, request, answer, sender, given, waiting, rbuffer, sbuffer) :}\iff
\]

\[
\forall i \in \{1, 2\} : \\
(\forall j \neq i \Rightarrow \neg (pc(i) \neq C \land sbuffer(j) = 0 \land answer(i) = 0) \land \\
(pc(i) = R \Rightarrow \\
\text{answer}(i) = 0 \land \text{given} \Rightarrow request(i) = 1 \land \text{rbuffer}(i) = 1 \land \text{sender} = i) \land \\
\text{request}(i) = 0 \lor \text{rbuffer}(i) = 0) ) \land \\
(pc(i) = S \Rightarrow \\
\text{rbuffer}(i) = 1 \land \text{answer}(i) = 1 \Rightarrow \\
\text{request}(i) = 0 \land \text{rbuffer}(i) = 0 \land \text{sender} = i) \land \\
\text{request}(i) = 0 \lor \text{rbuffer}(i) = 0) ) \land \\
(pc(i) = C \Rightarrow \\
\text{request}(i) = 0 \land \text{rbuffer}(i) = 0 \land \text{sender} = i) \land \\
\text{request}(i) = 0 \land \text{answer}(i) = 0) \land
\]

\[
\text{\ldots}
\]
The Verification Task (Contd)

... (sender = 0 ∧ (request(i) = 1 ∨ rbuffer(i) = 1) ⇒
  sbuffer(i) = 0 ∧ answer(i) = 0) ∧
(sender = i ⇒
  (waiting = i) ∧
  (sender = given ∧ pc(i) = R ⇒
    request(i) = 0 ∧ rbuffer(i) = 0) ∧
  (pc(i) = S ∧ i ≠ given ⇒
    request(i) = 0 ∧ rbuffer(i) = 0) ∧
  (waiting = i ⇒
    given ≠ i ∧ pc(i) = S ∧ request(i) = 0 ∧ rbuffer(i) = 0 ∧
    sbuffer(i) = 0 ∧ answer(i) = 0) ∧
  (sbuffer(i) = 1 ⇒
    answer(i) = 0 ∧ request(i) = 0 ∧ rbuffer(i) = 0)

As usual, the invariant has been elaborated in the course of the proof.

The RISC ProofNavigator Theory (Contd)

% initial state condition
% -------------------------------
% IC: (PC, BOOLEAN, BOOLEAN) -> BOOLEAN =
% LAMBDA(pc, request, answer): pc = R AND (request <=> FALSE) AND (answer <=> FALSE);

% transition relation
% -------------------
% IC: (PC, BOOLEAN, BOOLEAN) -> BOOLEAN =
% LAMBDA(pc, request, answer): pc = R AND (request <=> FALSE) AND (answer <=> FALSE);

% server state
given: Index0; given0: Index0;
waiting: Index0; waiting0: Index0;
rbuffer: Index -> BOOLEAN; rbuffer0: Index -> BOOLEAN;
sbuffer: Index -> BOOLEAN; sbuffer0: Index -> BOOLEAN;

% initial state condition
Initial: BOOLEAN =
(FORALL(i:Index): (rbuffer(i)<=>FALSE) AND (sbuffer(i)<=>FALSE));

% server state
given = 0 AND waiting = 0 AND sender = 0 AND
(FORALL(i:Index): (rbuffer(i)<=>FALSE) AND (sbuffer(i)<=>FALSE));

As usual, the invariant has been elaborated in the course of the proof.

The RISC ProofNavigator Theory

newcontext "clientServer";

Index: TYPE = SUBTYPE(LAMBDA(x:INT): x=1 OR x=2);
Index0: TYPE = SUBTYPE(LAMBDA(x:INT): x=0 OR x=1 OR x=2);

% program counter type
PCBASE: TYPE;
R: PCBASE; S: PCBASE; C: PCBASE;
PC: TYPE = SUBTYPE(LAMBDA(x:PCBASE): x=R OR x=S OR x=C);

% client states
pc: Index->PC; pc0: Index->PC;
request: Index->BOOLEAN; request0: Index->BOOLEAN;
answer: Index->BOOLEAN; answer0: Index->BOOLEAN;

% client states
pc: Index->PC; pc0: Index->PC;
request: Index->BOOLEAN; request0: Index->BOOLEAN;
answer: Index->BOOLEAN; answer0: Index->BOOLEAN;

% server state
given: Index0; given0: Index0;
waiting: Index0; waiting0: Index0;
rbuffer: Index -> BOOLEAN; rbuffer0: Index -> BOOLEAN;
sbuffer: Index -> BOOLEAN; sbuffer0: Index -> BOOLEAN;

% server state
given: Index0; given0: Index0;
waiting: Index0; waiting0: Index0;
rbuffer: Index -> BOOLEAN; rbuffer0: Index -> BOOLEAN;
sbuffer: Index -> BOOLEAN; sbuffer0: Index -> BOOLEAN;

As usual, the invariant has been elaborated in the course of the proof.
The RISC ProofNavigator Theory (Contd'3)

\[
\begin{align*}
\text{(EXISTS (i: Index):} & \quad \text{sender = 0 AND (rbuffer(i) \iff TRUE) AND} \\
& \quad \text{sender0 = i AND (rbuffer0(i) \iff FALSE) AND} \\
& \quad \text{given = given0 AND waiting = waiting0 AND sbuffer = sbuffer0 AND} \\
& \quad \text{(FORALL (j: Index):} \quad j /= i \Rightarrow (rbuffer(j) \iff rbuffer0(j))) \text{ OR} \\
& \quad \text{(sender /= 0 AND sender = given AND waiting = 0 AND} \\
& \quad \text{given0 = 0 AND sender0 = 0 AND} \\
& \quad \text{waiting = waiting0 AND rbuffer = rbuffer0 AND sbuffer = sbuffer0) OR} \\
\end{align*}
\]

The RISC ProofNavigator Theory (Contd'4)

External: \((\text{Index, PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN,}
\text{Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, Index0, PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN,BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, BOOLEAN, Boolean
The RISC ProofNavigator Theory (Contd’7)

\[(pc(i) = S => (sbuffer(i) <=> TRUE) OR (answer(i) <=> TRUE) => (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i) AND (i /= given => (request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND (pc(i) = C => (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i AND (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE)) AND (sender = 0 AND ((request(i) <=> TRUE) OR (rbuffer(i) <=> TRUE)) => (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE)) AND (sender = i => (sender = given AND pc(i) = R => (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE)) AND waiting /= i AND (pc(i) = S AND i /= given => (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE)) AND (pc(i) = S AND i = given => (request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND (waiting = i => given /= i AND pc(waiting) = S AND (request(waiting) <=> FALSE) AND (rbuffer(waiting) <=> FALSE) AND (sbuffer(waiting) <=> FALSE) AND (answer(waiting) <=> FALSE)) AND ((sbuffer(i) <=> TRUE) => (answer(i) <=> FALSE) AND (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE));

The RISC ProofNavigator Theory (Contd’8)

% mutual exclusion proof
% Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer) => NOT(pc(1) = C AND pc(2) = C);

% invariance proof
% Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer);

% expand Initial, Invariant, IC, IS

The Proofs: MutEx and Inv1

Single application of autostar.
The Proofs: Inv2

[aa]: scatter
[bb]: expand Next
[cc]: split bfv
[dd]: decompose
[ee]: expand RS
[ff]: split 5xv
[gg]: expand Invariant
[hh]: auto
[ii]: proved (CVCL)

... (20 times)

[jj]: proved (CVCL)

... (15 times)

[kk]: scatter
[ll]: expand Invariant

... (20 times)

[mm]: proved (CVCL)

... (6 times)

[nn]: scatter
[oo]: expand Invariant

... (26 times)

[pp]: expand External

... (6 times)

[qq]: scatter
[rr]: expand Invariant

... (31 times)

[ss]: proved (CVCL)

... (1 times)

Ten main branches each requiring only single application of autostar.

Wolfgang Schreiner
http://www.risc.jku.at