Specifying and Verifying System Properties

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

1. The Basics of Temporal Logic

2. Specifying with Linear Time Logic

3. Verifying Safety Properties by Computer-Supported Proving

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 reachable 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:

- $[a, b] \rightarrow c \rightarrow c \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow c \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots$
- $[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \ldots$

Figure 3.1 Computation trees.  

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:
    $\rho(x), \neg F, F_0 \land F_1, F_0 \lor F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1, \forall x : F, \exists x : F$.
  - In propositional temporal logic only propositional logic formulas are state formulas (no quantification):
    $\rho, \neg F, F_0 \land F_1, F_0 \lor F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1$.
- Semantics: $s \models F$ ("$F$ holds in state $s$").
  - Example: semantics of conjunction.
    $\models (s \models F_0 \land F_1) :\Leftrightarrow (s \models F_0) \land (s \models F_1)$.
  - "$F_0 \land F_1$ holds in $s$ if and only if $F_0$ holds in $s$ and $F_1$ holds in $s$".

Classical logic reasoning on individual states.

Branching Time Logic (CTL)

We use temporal logic to specify a system property $F$.

- Core question: $S \models F$ ("$F$ holds in system $S$").
  - System $S = (I, R)$, temporal logic formula $F$.
  - Branching time logic:
    - $S \models F \Leftrightarrow S, s_0 \models F$, for every initial state $s_0$ of $S$.
    - Property $F$ must be evaluated on every pair of system $S$ and initial state $s_0$.
    - Given a computation tree with root $s_0$, $F$ is evaluated on that tree.

CTL formulas are evaluated on computation trees.

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: 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: 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) $p = p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \ldots$
      $R(p_i, p_{i+1})$ 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 $p_0 = s$. 

Wolfgang Schreiner
http://www.risc.jku.at
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 $G$").


- Semantics: $S, p |= P$ ("$P$ holds in path $p$ of system $S$").
  
  $S, p |= X F :⇔ S, p_1 |= F$.
  $S, p |= G F :⇔ \forall i \in \mathbb{N} : S, p_i |= F$.
  $S, p |= F F :⇔ \exists i \in \mathbb{N} : S, p_i |= F$.
  $S, p |= F U G :⇔ \exists i \in \mathbb{N} : S, p_i |= G \land \forall j \in \mathbb{N} : S, p_j |= F$.

Path Quantifiers and Temporal Operators

We use temporal logic to specify a system property $P$.

- Core question: $S |= P$ ("$P$ holds in system $S$").
  - System $S = \langle I, R \rangle$, temporal logic formula $P$.
  - Linear time logic:
    - $S |= P :⇔ r |= P$, 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.

Linear Time Logic (LTL)

We use temporal logic to specify a system property $P$.

- Core question: $S |= P$ ("$P$ holds in system $S$").
  - System $S = \langle I, R \rangle$, temporal logic formula $P$.
  - Linear time logic:
    - $S |= P :⇔ r |= P$, 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:

$\models p = P$ ("$P$ holds in path $p$.

$\models p_i := \langle p_i, p_{i+1}, \ldots \rangle$.

$\models p = f :\iff \models p_0 = f$.

$\models p = X F :\iff \models p_1 = F$.

$\models p = G F :\iff \forall i \in \mathbb{N} : p_i = F$.

$\models p = F F :\iff \exists i \in \mathbb{N} : p_i = F$.

$\models p = F U G :\iff \exists i \in \mathbb{N} : p_i = G \land \forall j \in \mathbb{N} : (p_j \models F$.

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 $S = (I, R)$, temporal logic formula $P$.

Branching time logic:

- $S \models P :\iff S, s_0 \models P$, 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:

- $S \models P :\iff r \models P$, 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 $S$.

The two variants of temporal logic have different expressive power.

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 $AX(\exists X Q \land \exists X \neg Q)$.
- True for left system, false for right system.
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**: $\Diamond \Box F$ (i.e. $FG F$).
  - "In every run, there is a moment from which on $F$ holds forever".
  - Naive translation $\text{AFG } F$ is not a CTL formula.
  - $G F$ 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.

---

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, \ldots (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.
Frequently Used LTL Patterns

In practice, most temporal formulas are instances of particular patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pronounced</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Box F$</td>
<td>always $F$</td>
<td>invariance</td>
</tr>
<tr>
<td>$\Diamond F$</td>
<td>eventually $F$</td>
<td>guarantee</td>
</tr>
<tr>
<td>$\Diamond \Box F$</td>
<td>$F$ holds infinitely often</td>
<td>recurrence</td>
</tr>
<tr>
<td>$\Box \Diamond F$</td>
<td>eventually $F$ holds permanently</td>
<td>stability</td>
</tr>
<tr>
<td>$\Box (F \Rightarrow \Diamond G)$</td>
<td>always, if $F$ holds, then $G$ holds</td>
<td>response</td>
</tr>
<tr>
<td>$\Diamond (G \lor H)$</td>
<td>always, if $F$ holds, then $G$ holds until $H$ holds</td>
<td>precedence</td>
</tr>
</tbody>
</table>

Typically, there are at most two levels of nesting of temporal operators.

Temporal Rules

Temporal operators obey a number of fairly intuitive rules.

- **Extraction laws:**
  - $\Box F \iff F \land \Box F$.
  - $\Diamond F \iff F \lor \Diamond F$.
  - $F \lor G \iff G \lor (F \lor \Diamond (F \land G))$.

- **Negation laws:**
  - $\neg \Box F \iff \Diamond \neg F$.
  - $\neg \Diamond F \iff \Box \neg F$.
  - $\neg(F \lor G) \iff (\neg G) \lor (\neg F \land \neg G) \lor \Diamond \neg G$.

- **Distributivity laws:**
  - $\Box (F \land G) \iff (\Box F) \land (\Box G)$.
  - $(F \land \Diamond) \lor G \iff (\Diamond F) \lor (\Diamond G)$.
  - $(F \land G) \lor H \iff (F \lor H) \land (G \lor H)$.
  - $F \lor (G \lor H) \iff (F \lor G) \lor (F \lor H)$.
  - $\Diamond \Box F \lor G \iff (\Diamond F) \lor (\Diamond F) \lor (\Diamond G)$.
System Properties

There exists 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: $\square F$ (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 $\square F$.

- **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: $\Diamond F$ (with state property $F$).
  - Any finite prefix $p$ can be extended to a run $p \rightarrow F \rightarrow \ldots$ which satisfies $\Diamond F$.

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 $L$.
  - 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”.

Verifying Safety

We only consider a special case of a safety property.

- **Example:** $P : \iff (\square A) \land (\Diamond B)$ (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 $[A, \neg B] \rightarrow \ldots \rightarrow [A, \neg B]$ of this run can be extended to a run $[A, \neg B] \rightarrow \ldots \rightarrow [A, \neg B] \rightarrow [A, B] \rightarrow [A, B] \rightarrow \ldots$ satisfying $P$.
  - 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 || q0 : wait x = 1
  p1 : x := x + 1
  q1 : x := x - 1

State = \{p0, p1\} \times \{q0, q1\} \times \mathbb{Z}.
```

```
I(p, q, x) \iff p = p0 \land q = q0 \land x = 0.
R((p, q, x), (p', q', x')) \iff P0(...) \lor P1(...) \lor Q0(...) \lor Q1(...).
```

```
P0((p, q, x), (p', q', x')) \iff p = p0 \land q = q0 \land x = x + 1.
P1((p, q, x), (p', q', x')) \iff p = p1 \land q = q1 \land x' = x + 1.
Q0((p, q, x), (p', q', x')) \iff q = q0 \land x = 1 \land p = p' \land q = q' = q1 \land x' = x.
Q1((p, q, x), (p', q', x')) \iff q = q1 \land p = p' \land q = q' = q0 \land x' = x - 1.
```

Prove \(\langle I, R\rangle \models \Box(x = 0 \lor x = 1)\).

Example

```
var x := 0, y := 0
loop
  x := x + 1 || y := y + 1
```

```
State = \mathbb{N} \times \mathbb{N}; Label = \{p, q\}.
I(x, y) \iff x = 0 \land y = 0.
R(I, (x, y), (x', y')) \iff
  (I = p \land x = x + 1 \land y' = y) \lor (I = q \land x' = x \land y' = y + 1).
```

Problem: F is not inductive.

- F is too weak to prove the induction step.
  - \(F(s) \land R(s, s') \Rightarrow F(s')\).

Solution: find stronger invariant I.

- If I \models F, then (\Box I) \models (\Box F).
- It thus suffices to prove \Box I.

Rationale: I may be inductive.

- If yes, I is strong enough to prove the induction step.
  - \(I(s) \land R(s, s') \Rightarrow I(s')\).
- If not, find a stronger invariant I' and try again.

Invariant I represents additional knowledge for every proof.
- Rather than proving \Box P, prove \Box(I \Rightarrow P).

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

Verifying Liveness

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

```
Prove \(\langle I, R\rangle \models \Box(x = 0 \lor x = 1)\).
```

Proof attempt fails.

```
Prove \(\langle I, R\rangle \models \Box \langle \langle p, q, x, (p', q', x'), \Box G \rangle \rangle\).
```

```
G :\iff
  (x = 0 \lor x = 1) \land
  (p = p1 \Rightarrow x = 0) \land
  (q = q1 \Rightarrow x = 1).
```

Proof works.

```
G \Rightarrow (x = 0 \lor 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):**

\[
\begin{align*}
\text{Enabled}_R(p, (x, y)) & \Leftrightarrow \exists x', y' : R(p, (x, y), (x', y')) \\
& \Leftrightarrow \exists x', y' : (p = p \land x' = x + 1 \land y' = y) \lor \\
& \quad (p = q \land x' = x \land y' = y + 1) \\
& \Leftrightarrow (\exists x', y' : p = p \land x' = x + 1 \land y' = y) \lor \\
& \quad (\exists x', y' : p = q \land x' = x \land y' = y + 1) \\
& \Leftrightarrow \text{true} \lor \text{false} \\
& \Leftrightarrow \text{true}.
\end{align*}
\]

- Transition \( p \) is always enabled.

Weak Fairness

- **Weak Fairness**
  - A run \( s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots \) is weakly fair to a transition \( l \), if
    - If \( l \) is eventually permanently enabled in the run,
    - Then \( l \) is executed infinitely often in the run.
  - \( (\exists i : \forall j \geq i : \text{Enabled}_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l = l_j) \).
  - 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 \( WF_l :⇔ (\Box E_l) \Rightarrow (\Box X_l) \).
- If \( l \) is eventually enabled forever, it is executed infinitely often.
- Prove \( (I, S) \models (WF_l \Rightarrow P) \).
- Property \( P \) is only proved for runs that are weakly fair to \( l \).

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

Strong Fairness

- **Strong Fairness**
  - A run \( s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots \) is strongly fair to a transition \( l \), if
    - If \( l \) is infinitely often enabled in the run,
    - Then \( l \) is also infinitely often executed in the run.
  - \( (\forall i : \exists j \geq i : \text{Enabled}_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l) \).
  - If \( l \) 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 \( SF_l :⇔ (\Box E_l) \Rightarrow (\Box X_l) \).
- If \( l \) is enabled infinitely often, it is executed infinitely often.
- Prove \( (I, S) \models (SF_l \Rightarrow P) \).
- Property \( P \) is only proved for runs that are strongly fair to \( l \).

A much stronger requirement to the fairness of a system.
Example

\begin{verbatim}
var x=0
loop
  a: x := -x
  b: choose x := 0 [] x := 1

State := {a, b} × Z; Label = {A, B0, B1}.
I(p, x) : ...
R(l, ⟨p, x⟩, ⟨p', x'⟩) :
  (l = A ∧ (p = a ∧ p' = b ∧ x' = -x)) ∨
  (l = B0 ∧ (p = b ∧ p' = a ∧ x' = 0)) ∨
  (l = B1 ∧ (p = b ∧ p' = a ∧ x' = 1)).
\end{verbatim}

\begin{itemize}
  \item \(⟨I, R⟩ \models SF_{B1} \Rightarrow \Diamond x = 1\).
  \item \([a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_0} [a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_1} [a, 0] \xrightarrow{A} \ldots\)
  \item This (only) violating run is not strongly fair to \(B_1\) (but weakly fair).
    \begin{itemize}
      \item \(B_1\) is infinitely often enabled.
      \item \(B_1\) is never executed.
    \end{itemize}
\end{itemize}

System satisfies specification if strong fairness is assumed.

Weak versus Strong Fairness

1. The Basics of Temporal Logic
2. Specifying with Linear Time Logic
3. Verifying Safety Properties by Computer-Supported Proving

A Bit Transmission Protocol

\begin{verbatim}
var x, y
var v := 0, r := 0, a := 0
S: loop
  choose x ∈ {0, 1} || R: loop
    1: v, r := x, 1
    2: wait a = 1
      r := 0
    2: wait r = 0
      a := 0

Transmit a sequence of bits through a wire.
\end{verbatim}
The invariant captures the essence of the protocol.

\[ \text{S1} \lor \text{S2} \lor \text{S3} \lor \text{R1} \lor \text{R2} \]

\[ \text{R3}(p, q, x, y, v, r, a) : \Rightarrow p = q = 1 \land x \in \mathbb{N}_2 \land v = r = a = \ldots = 1 \Rightarrow a = 0 \land (q = 2 \Rightarrow (p = 2 \lor p = 3) \land a = 1 \land y = x) \]

The invariant captures the essence of the protocol.

The RISC ProofNavigator Theory

newcontext "protocol";

\[ p: \text{NAT}; q: \text{NAT}; x: \text{NAT}; y: \text{NAT}; v: \text{NAT}; r: \text{NAT}; a: \text{NAT}; \]

\[ p0: \text{NAT}; q0: \text{NAT}; x0: \text{NAT}; y0: \text{NAT}; v0: \text{NAT}; r0: \text{NAT}; a0: \text{NAT}; \]

Init: \( \text{BOOLEAN} = p = 1 \land q = 1 \land (x = 0 \lor x = 1) \land v = 0 \land r = 0 \land a = 0; \)

Step: \( \text{BOOLEAN} = \text{S1} \lor \text{S2} \lor \text{S3} \lor \text{R1} \lor \text{R2}; \)

Invariant: \( \text{NAT}, \text{NAT}, \text{NAT}, \text{NAT}, \text{NAT}, \text{NAT}, \text{NAT}, \text{NAT}) \rightarrow \text{BOOLEAN} = \lambda \text{p}, q, x, y, v, r, a: \text{NAT}: \)

\[ p = 1 \lor p = 2 \lor p = 3 \land (q = 1 \lor q = 2) \land (x = 0 \lor x = 1) \land (v = 0 \lor v = 1) \land (r = 0 \lor r = 1) \land (a = 0 \lor a = 1) \]

\[ (p = 1 \Rightarrow q = 1 \land r = 0 \land a = 0) \land (p = 2 \Rightarrow r = 1 \land v = x) \land (p = 3 \Rightarrow r = 0) \land (q = 1 \Rightarrow a = 0) \land (q = 2 \Rightarrow (p = 2 \lor p = 3) \land a = 1 \land y = x); \]
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);

The Proofs

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

[wd2]: expand Init, Invariant in nra
[ipl]: 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

Client system Ci = (ICi, RCi).
State := PC × N2 × N2.
Int := {Ri, Si, Ci}.
ICi (pc, ... // critical region
...
sendRequest()
endloop
end Client

Server system S = ⟨IS, RS⟩.
State := (N3)3 × ((1, 2) → N2)2.
Int := {D1, D2, F, A1, A2, W}.

Server:
local given, waiting, sender begin
given := 0; waiting := 0
loop
if sender = given then
sendRequest()
else
if waiting = 0 then
sendAnswer(given)
else
sendAnswer(given)
endif
endif
endloop
end Server

A Client/Server System (Contd)

D: sender := receiveRequest() if sender = given then
else
if waiting = 0 then
waiting := 0
else
endif
endif

A Client/Server System (Contd)

A1: given := waiting; waiting := 0
A2: given := sender sendAnswer(given)
W: waiting := sender endif

∀i ∈ {1, 2} : (l = Di ∧ sender = 0 ∧ rbuffer(i) = 0 ∧ sender1 = i ∧ rbuffer1(i) = 0 ∧
U(given, waiting, sbuffer) ∧
∀j ∈ {1, 2} \ {i} : Uj(rbuffer(j))

U(x1, ..., x0) := x1 = x0 ∧ ... ∧ x'1 = x0,
Uj(x1, ..., x0) := x'j = xj(j) ∧ ... ∧ x'j = xj(j).

Int :=
IS :=
RS :=
given := waiting;
U :=

∀i ∈ {1, 2} : (l = Di ∧ sender = 0 ∧ rbuffer(i) = 0 ∧ sender1 = i ∧ rbuffer1(i) = 0 ∧
U(given, waiting, sbuffer) ∧
∀j ∈ {1, 2} \ {i} : Uj(rbuffer(j))

U(x1, ..., x0) := x1 = x0 ∧ ... ∧ x'1 = x0,
Uj(x1, ..., x0) := x'j = xj(j) ∧ ... ∧ x'j = xj(j).
A Client/Server System (Contd'2)

\[
(I = F ∧ sender ≠ 0 ∧ sender = given ∧ waiting = 0 ∧
given' = 0 ∧ sender' = 0 ∧
U(waiting, rbuffer, sbuffer)) ∨
\]

\[
(I = A1 ∧ sender ≠ 0 ∧ sbuffer(waiting) = 0 ∧
sender = given ∧ waiting ≠ 0 ∧
given' = 0 ∧ sender' = 0 ∧
U(rbuffer) ∧
\forall j ∈ \{1, 2\}, \{waiting : U_j(sbuffer)\} ∨
\]

\[
(I = A2 ∧ sender ≠ 0 ∧ sbuffer(sender) = 0 ∧
sender ≠ given ∧ given = 0 ∧
given' = sender ∧
sender' = given = 0 ∧
U(waiting, rbuffers) ∧
\forall j ∈ \{1, 2\}, \{sender : U_j(sbuffer)\} ∨
\]

\[
(I = F ∧ waiting' := sender ∧...
\]

\[
linewidth{0.0pt} \quad \toprule
\end{tabular}
\]

A Client/Server System (Contd'3)

\[
(I = W ∧ sender ≠ 0 ∧ sender ≠ given ∧ given ≠ 0 ∧
waiting' := sender ∧ sender' = 0 ∧
U(rbuffer, sbuffer) ∨
\]

\[
\exists j ∈ \{1, 2\}:
\]

\[
(I = REQ ∧ sbuffer'(i) = 1 ∧
U(given, waiting, sender, rbuffers) ∧
\forall j ∈ \{1, 2\}, \{i : U_j(rbuffer)\} ∨
\]

\[
(l = ANS ∧ sbuffer(i) ≠ 0 ∧
sender'(i) = 0 ∧
U(given, waiting, sender, rbuffers) ∧
\forall j ∈ \{1, 2\}, \{i : U_j(sbuffer)\}).
\]

The Verification Task

\[
\{I, R\} \models □¬(pc_1 = C ∧ pc_2 = C)
\]

Invariant(pc, request, answer, sender, given, waiting, rbuffer, sbuffer) :=

\[
\forall i ∈ \{1, 2\}:
\]

\[
(pc(i) = C ∨ \text{sender}(i) = 1 ∨ \text{answer}(i) = 1)
\]

\[
\forall i ∈ \{1, 2\}:
\]

\[
(pc(i) = R ⇒ \text{sender}(i) = 0 ∧ \text{answer}(i) = 0 ∧
(i = \text{given} ⇒ \text{request}(i) = 1 ∨ \text{rbuffer}(i) = 1 ∨ \text{sender} = i) ∧
(\text{request}(i) = 0 ∨ \text{rbuffer}(i) = 0) ∧
\]

\[
(pc(i) = S ⇒ \text{rbuffer}(i) = 1 ∨ \text{answer}(i) = 1 ⇒
\]

\[
\text{request}(i) = 0 ∧ \text{rbuffer}(i) = 0 ∧ \text{sender} = i) ∧
\]

\[
(pc(i) = C ⇒
\]

\[
\text{request}(i) = 0 ∨ \text{rbuffer}(i) = 0 ∧ \text{sender} = i ∧
\text{sender}(i) = 0 ∧ \text{answer}(i) = 0 ∧
\]

\[
\]
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) ∧ 
(pc(i) = S ∧ i = given ⇒ 
request(i) = 0 ∧ rbuffer(i) = 0) ∧ 
(waiting = i ⇒ 
given ≠ i ∧ pc(i) = S ∧ request = 0 ∧ rbuffer(i) = 0 ∧ 
sbuffer = 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

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

IS: (Index0, Index0, Index0, Index->BOOLEAN) -> BOOLEAN = 
 LAMBDA(given: Index0, waiting: Index0, sender: Index0, 
 rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN): 
 given = 0 AND waiting = 0 AND sender = 0 AND 
(FORALL(i:Index): (rbuffer(i)<=>FALSE) AND (sbuffer(i)<=>FALSE));

Initial: BOOLEAN = 
(FORALL(i:Index): IC(pc(i), request(i), answer(i))) AND 
IS(given, waiting, sender, rbuffer, sbuffer);
The RISC ProofNavigator Theory (Contd'3)

\[
\text{In} = (\exists i : \text{Index}:
\begin{array}{l}
\text{sender} = 0 \text{ AND } (\text{rbuffer}(i) \iff \text{TRUE}) \text{ AND } \\
\text{sender0} = i \text{ AND } (\text{rbuffer0}(i) \iff \text{FALSE}) \text{ AND } \\
\text{given} = \text{given0} \text{ AND } \text{waiting} = \lnot \text{given0} \text{ AND } \text{rbuffer} = \text{rbuffer0} \text{ AND } \\
(\forall j : \text{Index}:
\begin{array}{l}
\text{send} = i \implies (\text{rbuffer}(j) \iff \text{rbuffer0}(j))
\end{array})
\end{array})
\]

The RISC ProofNavigator Theory (Contd'4)

\[
\text{External} : (\text{Index}, \text{PC}, \text{BOOLEAN}, \text{BOOLEAN}, \text{PC}, \text{BOOLEAN}, \text{BOOLEAN}, \\
\text{Index0}, \text{Index0}, \text{Index0}, \text{Index0} = \text{BOOLEAN}, \text{Index0} = \text{BOOLEAN}, \\
\text{Index0} = \text{BOOLEAN}, \text{Index0} = \text{BOOLEAN}, \text{Index0} = \text{BOOLEAN} = \text{BOOLEAN} = \text{BOOLEAN}) = \text{LAMBDA}(i : \text{Index}:
\begin{array}{l}
\text{pc} = i \text{ AND } (\text{request0} = \text{FALSE}) \text{ AND } (\text{rbuffer0}(i) = \text{TRUE})
\end{array})
\]

The RISC ProofNavigator Theory (Contd'5)

\[
\text{Next} = ((\exists i : \text{Index}:
\begin{array}{l}
\text{RC}(\text{pc}(i), \text{request}(i), \text{answer}(i), \\
\text{pc0}(i), \text{request0}(i), \text{answer0}(i)) \text{ AND } \\
(\forall j : \text{Index}:
\begin{array}{l}
\text{pc}(j) = \text{pc0}(j) \text{ AND } (\text{request0}(j) \iff \text{request}(j)) \text{ AND } \\
(\text{answer0}(j) \iff \text{answer}(j))
\end{array})
\end{array})
\]

The RISC ProofNavigator Theory (Contd'6)

% invariant

% Invariant : (\text{Index} = \text{Index0}, \text{Index0} = \text{Index0} = \text{BOOLEAN} = \text{BOOLEAN}) = \text{LAMBDA}(\text{pc} = \text{Index} = \text{PC}, \text{request} = \text{Index0} = \text{BOOLEAN}, \text{answer} = \text{Index0} = \text{BOOLEAN}, \\
\text{given} = \text{Index0} = \text{Index0} = \text{Index0} = \text{BOOLEAN} = \text{BOOLEAN} = \text{BOOLEAN} = \text{BOOLEAN}
\]

\[
(\forall i : \text{Index}:
\begin{array}{l}
(\text{pc}(i) = C \implies (\text{rbuffer}(i) \iff \text{TRUE}) \text{ AND } (\text{answer}(i) \iff \text{TRUE}) \text{ AND } \\
(\text{given} = \text{TRUE}) \text{ AND } (\text{rbuffer0}(i) \iff \text{FALSE}) \text{ AND } (\text{answer0}(i) \iff \text{FALSE}))
\end{array})
\]

\[
(\text{pc}(i) = \text{R} \implies (\text{rbuffer}(i) \iff \text{FALSE}) \text{ AND } (\text{answer}(i) \iff \text{FALSE}) \text{ AND } \\
(\text{given} = \text{TRUE}) \text{ AND } (\text{rbuffer0}(i) \iff \text{TRUE}) \text{ AND } (\text{answer0}(i) \iff \text{TRUE}) \text{ AND } \\
(\text{given0} = \text{TRUE} \implies (\text{rbuffer}(i) \iff \text{FALSE}) \text{ AND } (\text{answer}(i) \iff \text{TRUE}) \text{ AND } \\
(\text{given0} = \text{FALSE} \implies (\text{rbuffer}(i) \iff \text{TRUE}) \text{ AND } (\text{answer}(i) \iff \text{FALSE})))
\]
(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)) =>
(request(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(waiting) = S AND i /= given =>
(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 Proofs: Inv2

[pas]: scatter
[lab]: expand Next
[lab]: decompose
[phs]: expand NR
[lpn]: split 5xv
[pt6]: expand Invariant
[st6]: scatter
[txt]: expand Invariant
[qt6]: expand Invariant
[snq]: scatter
[avi]: auto
[cct]: proved (CVCL)
[seg]: proved (CVCL)

Ten main branches each requiring only single application of autostar.

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