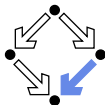


# Specifying and Verifying System Properties

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

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





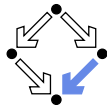
---

# 1. The Basics of Temporal Logic

## 2. Specifying with Linear Time Logic

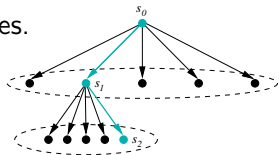
## 3. Verifying Safety Properties by Computer-Supported Proving

# Motivation



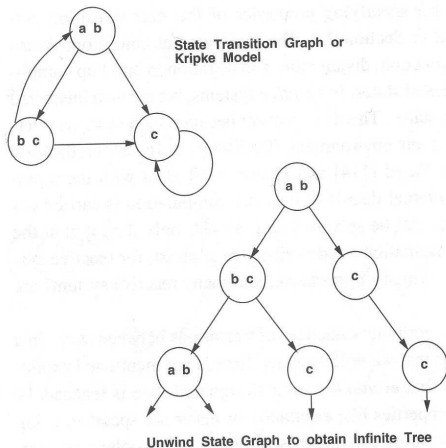
We need a language for specifying system properties.

- 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 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  $s_0 \rightarrow \dots \rightarrow s_n$  is extended to an infinite run  $s_0 \rightarrow \dots \rightarrow s_n \rightarrow s_n \rightarrow s_n \rightarrow \dots$
- 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:

$[a, b] \rightarrow c \rightarrow c \rightarrow \dots$

$[a, b] \rightarrow [b, c] \rightarrow c \rightarrow \dots$

$[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \dots$

$[a, b] \rightarrow [b, c] \rightarrow [a, b] \rightarrow \dots$

...

Figure 3.1  
Computation trees.

Edmund Clarke et al: "Model Checking", 1999.



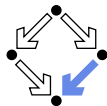
# State Formula

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:  
 $p(x), \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1, \forall x : F, \exists x : F$ .
  - In **propositional temporal logic** only propositional logic formulas are state formulas (no quantification):  
 $p, \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1$ .
- **Semantics**:  $s \models F$  (“ $F$  holds in state  $s$ ”).
  - Example: semantics of conjunction.
    - $(s \models F_0 \wedge F_1) :\Leftrightarrow (s \models F_0) \wedge (s \models F_1)$ .
    - “ $F_0 \wedge F_1$  holds in  $s$  if and only if  $F_0$  holds in  $s$  and  $F_1$  holds in  $s$ ”.

**Classical logic reasoning on individual states.**

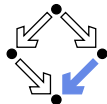
# Temporal Logic



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: **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.

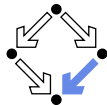
# Branching Time Logic (CTL)



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

- **Core question:**  $S \models F$  (" $F$  holds in system  $S$ ").
  - System  $S = \langle I, R \rangle$ , 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.**



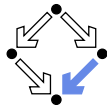
# State Formulas

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)  $p = p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots$   
 $R(p_i, p_{i+1})$  for every  $i$ ;  $p_0$  need not be an initial state.
  - Then the following are **state formulas**:
    - A**  $P$  (“in every path  $P$ ”),
    - E**  $P$  (“in some path  $P$ ”).
  - **Path quantifiers: A, 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:**  $S, p \models P$  (" $P$  holds in path  $p$  of system  $S$ ").

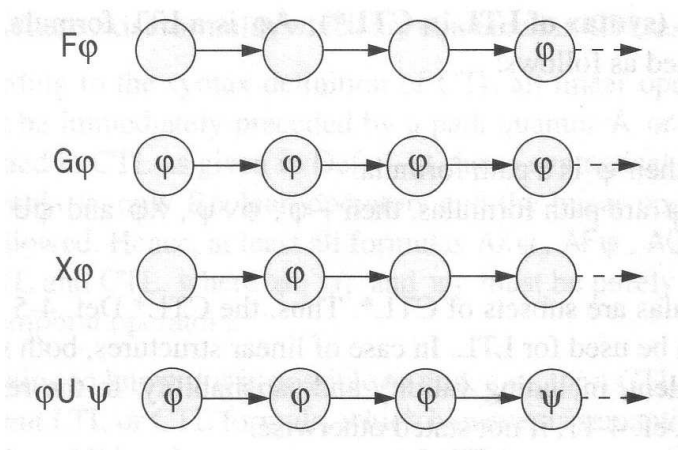
$$S, p \models \mathbf{X} F \Leftrightarrow S, p_1 \models F.$$

$$S, p \models \mathbf{G} F \Leftrightarrow \forall i \in \mathbb{N} : S, p_i \models F.$$

$$S, p \models \mathbf{F} F \Leftrightarrow \exists i \in \mathbb{N} : S, p_i \models F.$$

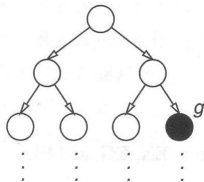
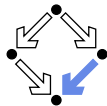
$$S, p \models F \mathbf{U} G \Leftrightarrow \exists i \in \mathbb{N} : S, p_i \models G \wedge \forall j \in \mathbb{N}_i : S, p_j \models F.$$

# Path Formulas

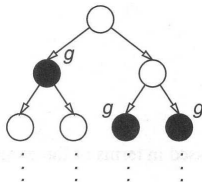


Thomas Kropf: "Introduction to Formal Hardware Verification", 1999.

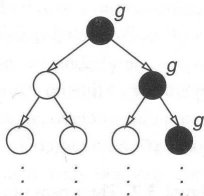
# Path Quantifiers and Temporal Operators



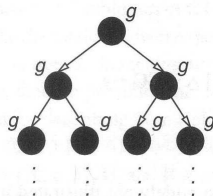
$M, s_0 \models \mathbf{EF} g$



$M, s_0 \models \mathbf{AF} g$



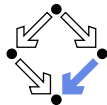
$M, s_0 \models \mathbf{EG} g$



$M, s_0 \models \mathbf{AG} g$

Edmund Clarke et al: "Model Checking", 1999.

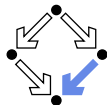
# Linear Time Logic (LTL)



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  $S$ .

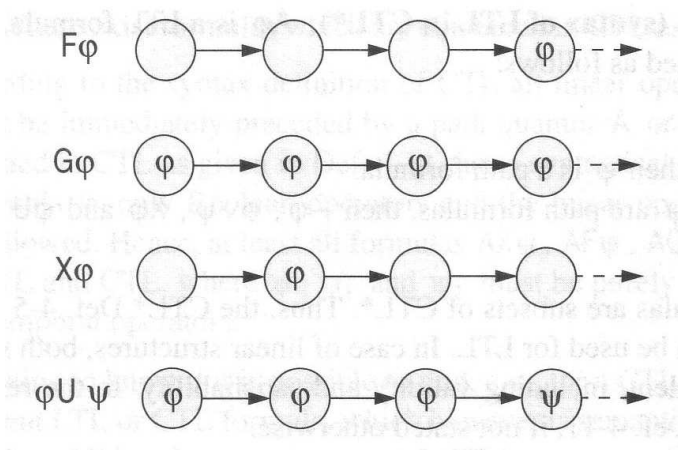
**LTL formulas are evaluated on system runs.**



No path quantifiers; all formulas are path formulas.

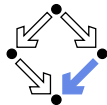
- Every **formula** is evaluated on a path  $p$ .
  - Also every state formula  $f$  of classical logic (see below).
  - Let  $F$  and  $G$  denote formulas.
  - Then also the following are formulas:
    - $\mathbf{X} F$  ("next time  $F$ "), often written  $\bigcirc F$ ,
    - $\mathbf{G} F$  ("always  $F$ "), often written  $\square F$ ,
    - $\mathbf{F} F$  ("eventually  $F$ "), often written  $\diamond F$ ,
    - $F \mathbf{U} G$  (" $F$  until  $G$ ").
- **Semantics:**  $p \models P$  (" $P$  holds in path  $p$ ").
  - $p^i := \langle p_i, p_{i+1}, \dots \rangle$ .
  - $p \models f \Leftrightarrow p_0 \models f$ .
  - $p \models \mathbf{X} F \Leftrightarrow p^1 \models F$ .
  - $p \models \mathbf{G} F \Leftrightarrow \forall i \in \mathbb{N} : p^i \models F$ .
  - $p \models \mathbf{F} F \Leftrightarrow \exists i \in \mathbb{N} : p^i \models F$ .
  - $p \models F \mathbf{U} G \Leftrightarrow \exists i \in \mathbb{N} : p^i \models G \wedge \forall j \in \mathbb{N}_i : p^j \models F$ .

# Formulas



Thomas Kropf: "Introduction to Formal Hardware Verification", 1999.

# 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:**
  - $S \models P \Leftrightarrow S, s_0 \models P$ , 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:**
  - $S \models P \Leftrightarrow r \models P$ , 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  $S$ .

# Branching versus Linear Time Logic

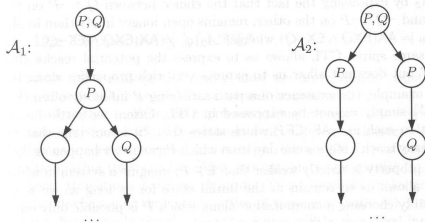
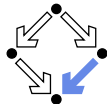


Fig. 2.4. Two automata, indistinguishable for PLTL

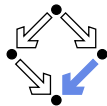
B. Berard et al: "Systems and Software Verification", 2001.

- **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  $\mathbf{AX}(\mathbf{EX} Q \wedge \mathbf{EX} \neg Q)$ .
  - 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: **AG(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 **AFG F** is **not** a CTL formula.
    - **G F** is a path formula, but **F** expects a state formula!
  - Translation **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.

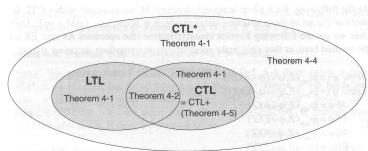
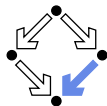


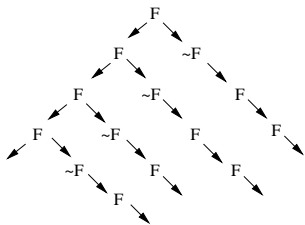
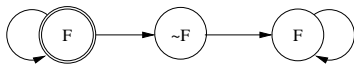
Fig. 4-8. Expressiveness of CTL\*, CTL+, CTL and LTL

Thomas Kropf: “Introduction to Formal Hardware Verification”, 1999.

# Branching versus Linear Time Logic

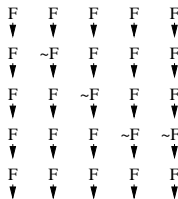


Proof that **AFAG F** (CTL) is different from  $\diamond\Box F$  (LTL).



**AFAG F**  $\Leftrightarrow$  false

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



$\diamond\Box F$   $\Leftrightarrow$  true

In every run, there is a moment from which on F holds forever.



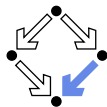
---

## 1. The Basics of Temporal Logic

## 2. Specifying with Linear Time Logic

## 3. Verifying Safety Properties by Computer-Supported Proving

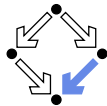
# Linear Time Logic



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.

# Frequently Used LTL Patterns

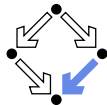


In practice, most temporal formulas are instances of particular patterns.

Pattern	Pronounced	Name
$\Box F$	always $F$	invariance
$\Diamond F$	eventually $F$	guarantee
$\Box \Diamond F$	$F$ holds infinitely often	recurrence
$\Diamond \Box F$	eventually $F$ holds permanently	stability
$\Box (F \Rightarrow \Diamond G)$	always, if $F$ holds, then eventually $G$ holds	response
$\Box (F \Rightarrow (G \mathbf{U} H))$	always, if $F$ holds, then $G$ holds until $H$ holds	precedence

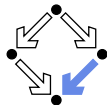
Typically, there are at most two levels of nesting of temporal operators.

# Examples



- **Mutual exclusion:**  $\Box \neg (pc_1 = C \wedge pc_2 = C)$ .
  - Alternatively:  $\neg \Diamond (pc_1 = C \wedge pc_2 = C)$ .
  - Never both components are simultaneously in the critical region.
- **No starvation:**  $\forall i : \Box (pc_i = W \Rightarrow \Diamond pc_i = R)$ .
  - Always, if component  $i$  waits for a response, it eventually receives it.
- **No deadlock:**  $\Box \neg \forall i : pc_i = W$ .
  - Never all components are simultaneously in a wait state  $W$ .
- **Precedence:**  $\forall i : \Box (pc_i \neq C \Rightarrow (pc_i \neq C \mathbf{U} lock = i))$ .
  - 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:**  $\Box (pc = L \Rightarrow C)$ .
  - Always if the program reaches line  $L$ , the condition  $C$  holds.
- **Termination:**  $\forall i : \Diamond (pc_i = T)$ .
  - Every component eventually terminates.

# Example



If event  $a$  occurs, then  $b$  must occur before  $c$  can occur (a run  $\dots, a, (\neg b)^*, c, \dots$  is illegal).

- **First idea (wrong)**

$$a \Rightarrow \dots$$

- Every run  $d, \dots$  becomes legal.

- **Next idea (correct)**

$$\Box(a \Rightarrow \dots)$$

- **First attempt (wrong)**

$$\Box(a \Rightarrow (b \mathbf{U} c))$$

- Run  $a, b, \neg b, c, \dots$  is illegal.

- **Second attempt (better)**

$$\Box(a \Rightarrow (\neg c \mathbf{U} b))$$

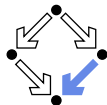
- Run  $a, \neg c, \neg c, \neg c, \dots$  is illegal.

- **Third attempt (correct)**

$$\Box(a \Rightarrow ((\Box \neg c) \vee (\neg c \mathbf{U} b)))$$

Specifier has to think in terms of allowed/prohibited sequences.

# Temporal Rules



Temporal operators obey a number of fairly intuitive rules.

## ■ Extraction laws:

- $\Box F \Leftrightarrow F \wedge \Box F.$
- $\Diamond F \Leftrightarrow F \vee \Diamond F.$
- $F \mathbf{U} G \Leftrightarrow G \vee (F \wedge \Box(F \mathbf{U} G)).$

## ■ Negation laws:

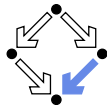
- $\neg \Box F \Leftrightarrow \Diamond \neg F.$
- $\neg \Diamond F \Leftrightarrow \Box \neg F.$
- $\neg(F \mathbf{U} G) \Leftrightarrow ((\neg G) \mathbf{U} (\neg F \wedge \neg G)) \vee \neg \Diamond G.$

## ■ Distributivity laws:

- $\Box(F \wedge G) \Leftrightarrow (\Box F) \wedge (\Box G).$
- $\Diamond(F \vee G) \Leftrightarrow (\Diamond F) \vee (\Diamond G).$
- $(F \wedge G) \mathbf{U} H \Leftrightarrow (F \mathbf{U} H) \wedge (G \mathbf{U} H).$
- $F \mathbf{U} (G \vee H) \Leftrightarrow (F \mathbf{U} G) \vee (F \mathbf{U} H).$
- $\Box \Diamond(F \vee G) \Leftrightarrow (\Box \Diamond F) \vee (\Box \Diamond G).$
- $\Diamond \Box(F \wedge G) \Leftrightarrow (\Diamond \Box F) \wedge (\Diamond \Box G).$



# Classes of System Properties



There exists two important classes of system properties.

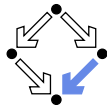
## ■ Safety Properties:

- 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:  $\Box F$  (with state property  $F$ ).
  - The violating run  $F \rightarrow F \rightarrow \neg F \rightarrow \dots$  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:  $\Diamond F$  (with state property  $F$ ).
  - Any finite prefix  $p$  can be extended to a run  $p \rightarrow F \rightarrow \dots$  which satisfies  $\Diamond F$ .

# System Properties

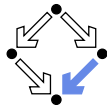


Not every system property is itself a safety property or a liveness property.

- **Example:**  $P :\Leftrightarrow (\Box A) \wedge (\Diamond B)$  (with state properties  $A$  and  $B$ )
  - Conjunction of a safety property and a liveness property.
- Take the run  $[A, \neg B] \rightarrow [A, \neg B] \rightarrow [A, \neg B] \rightarrow \dots$  violating  $P$ .
  - Any prefix  $[A, \neg B] \rightarrow \dots \rightarrow [A, \neg B]$  of this run can be extended to a run  $[A, \neg B] \rightarrow \dots \rightarrow [A, \neg B] \rightarrow [A, B] \rightarrow [A, B] \rightarrow \dots$  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?

# System Properties



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

## ■ Theorem:

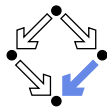
Every system property  $P$  is a conjunction  $S \wedge L$  of some safety property  $S$  and some liveness property  $L$ .

- 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.

- $M \models \Box F$ .
  - $F$  is a state formula (a formula without temporal operator).
  - Verify that  $F$  is an **invariant** of system  $M$ .
- $M = \langle I, R \rangle$ .
  - $I(s) :\Leftrightarrow \dots$
  - $R(s, s') :\Leftrightarrow R_0(s, s') \vee R_1(s, s') \vee \dots \vee R_{n-1}(s, s')$ .
- **Induction Proof.**
  - $\forall s : I(s) \Rightarrow F(s)$ .
    - Proof that  $F$  holds in every initial state.
  - $\forall s, s' : F(s) \wedge R(s, s') \Rightarrow F(s')$ .
    - Proof that each transition preserves  $F$ .
    - Reduces to a number of subproofs:
      - $F(s) \wedge R_0(s, s') \Rightarrow F(s')$
      - $\dots$
      - $F(s) \wedge R_{n-1}(s, s') \Rightarrow F(s')$

# Example


$$\begin{array}{l} \text{var } x := 0 \\ \text{loop} \\ \quad p_0 : \text{wait } x = 0 \\ \quad p_1 : x := x + 1 \end{array} \quad || \quad \begin{array}{l} \text{loop} \\ \quad q_0 : \text{wait } x = 1 \\ \quad q_1 : x := x - 1 \end{array}$$

$State = \{p_0, p_1\} \times \{q_0, q_1\} \times \mathbb{Z}$ .

$I(p, q, x) :\Leftrightarrow p = p_0 \wedge q = q_0 \wedge x = 0$ .

$R(\langle p, q, x \rangle, \langle p', q', x' \rangle) :\Leftrightarrow P_0(\dots) \vee P_1(\dots) \vee Q_0(\dots) \vee Q_1(\dots)$ .

$P_0(\langle p, q, x \rangle, \langle p', q', x' \rangle) :\Leftrightarrow p = p_0 \wedge x = 0 \wedge p' = p_1 \wedge q' = q \wedge x' = x$ .

$P_1(\langle p, q, x \rangle, \langle p', q', x' \rangle) :\Leftrightarrow p = p_1 \wedge p' = p_0 \wedge q' = q \wedge x' = x + 1$ .

$Q_0(\langle p, q, x \rangle, \langle p', q', x' \rangle) :\Leftrightarrow q = q_0 \wedge x = 1 \wedge p' = p \wedge q' = q_1 \wedge x' = x$ .

$Q_1(\langle p, q, x \rangle, \langle p', q', x' \rangle) :\Leftrightarrow q = q_1 \wedge p' = p \wedge q' = q_0 \wedge x' = x - 1$ .

Prove  $\langle I, R \rangle \models \square(x = 0 \vee x = 1)$ .

# Inductive System Properties



The induction strategy may not work for proving  $\square F$

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

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



# Example

- Prove  $\langle I, R \rangle \models \Box(x = 0 \vee x = 1)$ .
  - Proof attempt fails.
- Prove  $\langle I, R \rangle \models \Box G$ .

$G :\Leftrightarrow$

$$(x = 0 \vee x = 1) \wedge \\ (p = p_1 \Rightarrow x = 0) \wedge \\ (q = q_1 \Rightarrow x = 1).$$

- Proof works.
- $G \Rightarrow (x = 0 \vee x = 1)$  obvious.

See the proof presented in class.



# Verifying Liveness

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

$State = \mathbb{N} \times \mathbb{N}; Label = \{p, q\}.$

$I(x, y) :\Leftrightarrow x = 0 \wedge y = 0.$

$R(I, \langle x, y \rangle, \langle x', y' \rangle) :\Leftrightarrow$

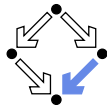
$(I = p \wedge x' = x + 1 \wedge y' = y) \vee (I = q \wedge x' = x \wedge y' = y + 1).$

- $\langle I, R \rangle \not\models \diamond x = 1.$ 
  - $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \dots$
  - This run violates (as the only one)  $\diamond x = 1.$
  - Thus the system as a whole does not satisfy  $\diamond x = 1.$

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



# Enabling Condition



When is a particular transition enabled for execution?

- $Enabled_R(l, s) :\Leftrightarrow \exists t : R(l, s, t)$ .
  - 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):

$$Enabled_R(p, \langle x, y \rangle)$$

$$\Leftrightarrow \exists x', y' : R(p, \langle x, y \rangle, \langle x', y' \rangle)$$

$$\Leftrightarrow \exists x', y' :$$

$$(p = p \wedge x' = x + 1 \wedge y' = y) \vee$$

$$(p = q \wedge x' = x \wedge y' = y + 1)$$

$$\Leftrightarrow (\exists x', y' : p = p \wedge x' = x + 1 \wedge y' = y) \vee$$

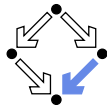
$$(\exists x', y' : p = q \wedge x' = x \wedge y' = y + 1)$$

$$\Leftrightarrow \text{true} \vee \text{false}$$

$$\Leftrightarrow \text{true}.$$

- Transition  $p$  is always enabled.

# Weak Fairness



## Weak Fairness

- A run  $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \dots$  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.

$$(\exists i : \forall j \geq i : Enabled_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l).$$

- 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  $WF_l : \Leftrightarrow (\diamond \square E_l) \Rightarrow (\square \diamond X_l)$ .
    - If  $l$  is eventually enabled forever, it is executed infinitely often.
  - Prove  $\langle l, S \rangle \models (WF_l \Rightarrow P)$ .
    - Property  $P$  is only proved for runs that are weakly fair to  $l$ .

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



# Example

$State = \mathbb{N} \times \mathbb{N}; Label = \{p, q\}.$

$I(x, y) :\Leftrightarrow x = 0 \wedge y = 0.$

$R(I, \langle x, y \rangle, \langle x', y' \rangle) :\Leftrightarrow$

$(I = p \wedge x' = x + 1 \wedge y' = y) \vee (I = q \wedge x' = x \wedge y' = y + 1).$

- $\langle I, R \rangle \models WF_p \Rightarrow \diamond x = 1.$ 
  - $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \dots$
  - 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

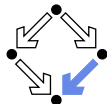
- A run  $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \dots$  is **strongly fair** to a transition  $l$ , if
  - if  $l$  is **infinitely often** enabled in the run,
  - then  $l$  is also infinitely often executed the run.

$$(\forall i : \exists j \geq i : Enabled_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l).$$

- 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  $SF_l : \Leftrightarrow (\Box \Diamond E_l) \Rightarrow (\Box \Diamond X_l)$ .
    - If  $l$  is enabled infinitely often, it is executed infinitely often.
  - Prove  $\langle l, S \rangle \models (SF_l \Rightarrow P)$ .
    - Property  $P$  is only proved for runs that are strongly fair to  $l$ .

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

# Example



```
var x=0
loop
  a : x := -x
  b : choose x := 0 [] x := 1
```

$State := \{a, b\} \times \mathbb{Z}; Label = \{A, B_0, B_1\}.$

$I(p, x) :\Leftrightarrow p = a \wedge x = 0.$

$R(I, \langle p, x \rangle, \langle p', x' \rangle) :\Leftrightarrow$

$(I = A \wedge (p = a \wedge p' = b \wedge x' = -x)) \vee$

$(I = B_0 \wedge (p = b \wedge p' = a \wedge x' = 0)) \vee$

$(I = B_1 \wedge (p = b \wedge p' = a \wedge x' = 1)).$

■  $\langle I, R \rangle \models SF_{B_1} \Rightarrow \diamond x = 1.$

■  $[a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_0} [a, 0] \xrightarrow{A} [b, 0] \xrightarrow{B_0} [a, 0] \xrightarrow{A} \dots$

■ 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.



# 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 of fairness exist.



---

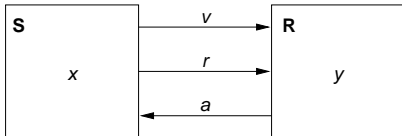
## 1. The Basics of Temporal Logic

## 2. Specifying with Linear Time Logic

## 3. Verifying Safety Properties by Computer-Supported Proving



# A Bit Transmission Protocol



```
var x, y
var v := 0, r := 0, a := 0
```

S: loop

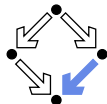
```
  choose x ∈ {0, 1}    ||
  1 : v, r := x, 1
  2 : wait a = 1
     r := 0
  3 : wait a = 0
```

R: loop

```
  1 : wait r = 1
     y, a := v, 1
  2 : wait r = 0
     a := 0
```

Transmit a sequence of bits through a wire.





# A (Simplified) Model of the Protocol

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

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

$$R(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ S1(\dots) \vee S2(\dots) \vee S3(\dots) \vee R1(\dots) \vee R2(\dots).$$

$$S1(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ p = 1 \wedge p' = 2 \wedge v' = x \wedge r' = 1 \wedge \\ q' = q \wedge x' = x \wedge y' = y \wedge a' = a.$$

$$S2(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ p = 2 \wedge p' = 3 \wedge a = 1 \wedge r' = 0 \wedge \\ q' = q \wedge x' = x \wedge y' = y \wedge v' = v \wedge a' = a.$$

$$S3(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ p = 3 \wedge p' = 1 \wedge a = 0 \wedge x' \in \mathbb{N}_2 \wedge \\ q' = q \wedge y' = y \wedge v' = v \wedge r' = r \wedge a' = a.$$

$$R1(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ q = 1 \wedge q' = 2 \wedge r = 1 \wedge y' = v \wedge a' = 1 \wedge \\ p' = p \wedge x' = x \wedge v' = v \wedge r' = r.$$

$$R2(\langle p, q, x, y, v, r, a \rangle, \langle p', q', x', y', v', r', a' \rangle) :\Leftrightarrow \\ q = 2 \wedge q' = 1 \wedge r = 0 \wedge a' = 0 \wedge \\ p' = p \wedge x' = x \wedge y' = y \wedge v' = v \wedge r' = r.$$



# A Verification Task

$$\langle I, R \rangle \models \Box(q = 2 \Rightarrow y = x)$$

$$\text{Invariant}(p, \dots) \Rightarrow (q = 2 \Rightarrow y = x)$$

$$I(p, \dots) \Rightarrow \text{Invariant}(p, \dots)$$

$$R(\langle p, \dots \rangle, \langle p', \dots \rangle) \wedge \text{Invariant}(p, \dots) \Rightarrow \text{Invariant}(p', \dots)$$

$$\text{Invariant}(p, q, x, y, v, r, a) :\Leftrightarrow$$

$$(p = 1 \vee p = 2 \vee p = 3) \wedge (q = 1 \vee q = 2) \wedge$$

$$(x = 0 \vee x = 1) \wedge (v = 0 \vee v = 1) \wedge (r = 0 \vee r = 1) \wedge (a = 0 \vee a = 1) \wedge$$

$$(p = 1 \Rightarrow q = 1 \wedge r = 0 \wedge a = 0) \wedge$$

$$(p = 2 \Rightarrow r = 1 \wedge v = x) \wedge$$

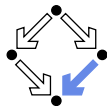
$$(p = 3 \Rightarrow r = 0) \wedge$$

$$(q = 1 \Rightarrow a = 0) \wedge$$

$$(q = 2 \Rightarrow (p = 2 \vee p = 3) \wedge a = 1 \wedge y = x)$$

The invariant captures the essence of the protocol.

# The RISC ProofNavigator Theory



```
newcontext "protocol";
```

```
p: NAT; q: NAT; x: NAT; y: NAT; v: NAT; r: NAT; a: NAT;  
p0: NAT; q0: NAT; x0: NAT; y0: NAT; v0: NAT; r0: NAT; a0: NAT;
```

```
S1: BOOLEAN =
```

```
  p = 1 AND p0 = 2 AND v0 = x AND r0 = 1 AND  
  q0 = q AND x0 = x AND y0 = y AND a0 = a;
```

```
S2: BOOLEAN =
```

```
  p = 2 AND p0 = 3 AND a = 1 AND r0 = 0 AND  
  q0 = q AND x0 = x AND y0 = y AND v0 = v AND a0 = a;
```

```
S3: BOOLEAN =
```

```
  p = 3 AND p0 = 1 AND a = 0 AND (x0 = 0 OR x0 = 1) AND  
  q0 = q AND y0 = y AND v0 = v AND r0 = r AND a0 = a;
```

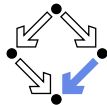
```
R1: BOOLEAN =
```

```
  q = 1 AND q0 = 2 AND r = 1 AND y0 = v AND a0 = 1 AND  
  p0 = p AND x0 = x AND v0 = v AND r0 = r;
```

```
R2: BOOLEAN =
```

```
  q = 2 AND q0 = 1 AND r = 0 AND a0 = 0 AND  
  p0 = p AND x0 = x AND y0 = y AND v0 = v AND r0 = r;
```

# The RISC ProofNavigator Theory



Init: BOOLEAN =

```
p = 1 AND q = 1 AND (x = 0 OR x = 1) AND  
v = 0 AND r = 0 AND a = 0;
```

Step: BOOLEAN =

```
S1 OR S2 OR S3 OR R1 OR R2;
```

Invariant: (NAT, NAT, NAT, NAT, NAT, NAT, NAT)->BOOLEAN =

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

```
(p = 1 OR p = 2 OR p = 3) AND  
(q = 1 OR q = 2) AND  
(x = 0 OR x = 1) AND  
(v = 0 OR v = 1) AND  
(r = 0 OR r = 1) AND  
(a = 0 OR a = 1) AND  
(p = 1 => q = 1 AND r = 0 AND a = 0) AND  
(p = 2 => r = 1 AND v = x) AND  
(p = 3 => r = 0) AND  
(q = 1 => a = 0) AND  
(q = 2 => (p = 2 OR p = 3) AND a = 1 AND 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



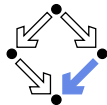
```
[vd2]: expand Invariant, Property in m2v
      [rle]: proved (CVCL)
```

```
[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  $C_i = \langle IC_i, RC_i \rangle$ .

State :=  $PC \times \mathbb{N}_2 \times \mathbb{N}_2$ .

Int :=  $\{R_i, S_i, C_i\}$ .

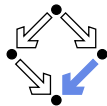
$IC_i(pc, request, answer) :\Leftrightarrow$   
 $pc = R \wedge request = 0 \wedge answer = 0$ .

$RC_i(I, \langle pc, request, answer \rangle,$   
 $\langle pc', request', answer' \rangle) :\Leftrightarrow$   
 $(I = R_i \wedge pc = R \wedge request = 0 \wedge$   
 $pc' = S \wedge request' = 1 \wedge answer' = answer) \vee$   
 $(I = S_i \wedge pc = S \wedge answer \neq 0 \wedge$   
 $pc' = C \wedge request' = request \wedge answer' = 0) \vee$   
 $(I = C_i \wedge pc = C \wedge request = 0 \wedge$   
 $pc' = R \wedge request' = 1 \wedge answer' = answer) \vee$

---

$(I = \overline{REQ}_i \wedge request \neq 0 \wedge$   
 $pc' = pc \wedge request' = 0 \wedge answer' = answer) \vee$   
 $(I = ANS_i \wedge$   
 $pc' = pc \wedge request' = request \wedge answer' = 1).$

```
Client(ident):  
  param ident  
  begin  
    loop  
      ...  
    R: sendRequest()  
    S: receiveAnswer()  
    C: // critical region  
      ...  
      sendRequest()  
    endloop  
  end Client
```



# A Client/Server System (Contd)

Server system  $S = \langle IS, RS \rangle$ .

$State := (\mathbb{N}_3)^3 \times (\{1, 2\} \rightarrow \mathbb{N}_2)^2$ .

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

$IS(given, waiting, sender, rbuffer, sbuffer) :\Leftrightarrow$   
 $given = waiting = sender = 0 \wedge$   
 $rbuffer(1) = rbuffer(2) = sbuffer(1) = sbuffer(2) = 0$ .

$RS(I, \langle given, waiting, sender, rbuffer, sbuffer \rangle,$   
 $\langle given', waiting', sender', rbuffer', sbuffer' \rangle) :\Leftrightarrow$   
 $\exists i \in \{1, 2\} :$   
 $(I = D_i \wedge sender = 0 \wedge rbuffer(i) \neq 0 \wedge$   
 $sender' = i \wedge rbuffer'(i) = 0 \wedge$   
 $U(given, waiting, sbuffer) \wedge$   
 $\forall j \in \{1, 2\} \setminus \{i\} : U_j(rbuffer)) \vee$   
 $\dots$

$U(x_1, \dots, x_n) :\Leftrightarrow x'_1 = x_1 \wedge \dots \wedge x'_n = x_n$ .

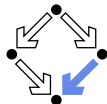
$U_j(x_1, \dots, x_n) :\Leftrightarrow x'_1(j) = x_1(j) \wedge \dots \wedge x'_n(j) = x_n(j)$ .

Server:

```

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





# A Client/Server System (Contd'2)

...

$$(I = F \wedge sender \neq 0 \wedge sender = given \wedge waiting = 0 \wedge given' = 0 \wedge sender' = 0 \wedge U(waiting, rbuffer, sbuffer)) \vee$$

$$(I = A1 \wedge sender \neq 0 \wedge sbuffer(waiting) = 0 \wedge sender = given \wedge waiting \neq 0 \wedge given' = waiting \wedge waiting' = 0 \wedge sbuffer'(waiting) = 1 \wedge sender' = 0 \wedge U(rbuffer) \wedge \forall j \in \{1, 2\} \setminus \{waiting\} : U_j(sbuffer)) \vee$$

$$(I = A2 \wedge sender \neq 0 \wedge sbuffer(sender) = 0 \wedge sender \neq given \wedge given = 0 \wedge given' = sender \wedge sbuffer'(sender) = 1 \wedge sender' = 0 \wedge U(waiting, rbuffer) \wedge \forall j \in \{1, 2\} \setminus \{sender\} : U_j(sbuffer)) \vee$$

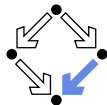
...

Server:

```

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

# A Client/Server System (Contd'3)



...  
 $(I = W \wedge sender \neq 0 \wedge sender \neq given \wedge given \neq 0 \wedge$   
 $waiting' := sender \wedge sender' = 0 \wedge$   
 $U(given, rbuffer, sbuffer)) \vee$

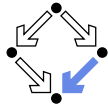
$\exists i \in \{1, 2\} :$

$(I = REQ_i \wedge rbuffer'(i) = 1 \wedge$   
 $U(given, waiting, sender, sbuffer) \wedge$   
 $\forall j \in \{1, 2\} \setminus \{i\} : U_j(rbuffer)) \vee$

$(I = \overline{ANS}_i \wedge sbuffer(i) \neq 0 \wedge$   
 $sbuffer'(i) = 0 \wedge$   
 $U(given, waiting, sender, rbuffer) \wedge$   
 $\forall j \in \{1, 2\} \setminus \{i\} : U_j(sbuffer)).$

Server:

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



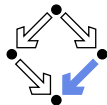
# A Client/Server System (Contd'4)

$$\text{State} := (\{1, 2\} \rightarrow PC) \times (\{1, 2\} \rightarrow \mathbb{N}_2)^2 \times (\mathbb{N}_3)^2 \times (\{1, 2\} \rightarrow \mathbb{N}_2)^2$$

$$I(\langle pc, request, answer, given, waiting, sender, rbuffer, sbuffer \rangle) :\Leftrightarrow \\ \forall i \in \{1, 2\} : IC(pc_i, request_i, answer_i) \wedge \\ IS(given, waiting, sender, rbuffer, sbuffer)$$

$$R(\langle pc, request, answer, given, waiting, sender, rbuffer, sbuffer \rangle, \\ \langle pc', request', answer', given', waiting', sender', rbuffer', sbuffer' \rangle) :\Leftrightarrow \\ (\exists i \in \{1, 2\} : RC_{local}(\langle pc_i, request_i, answer_i \rangle, \langle pc'_i, request'_i, answer'_i \rangle) \wedge \\ \langle given, waiting, sender, rbuffer, sbuffer \rangle = \\ \langle given', waiting', sender', rbuffer', sbuffer' \rangle) \vee \\ (RS_{local}(\langle given, waiting, sender, rbuffer, sbuffer \rangle, \\ \langle given', waiting', sender', rbuffer', sbuffer' \rangle) \wedge \\ \forall i \in \{1, 2\} : \langle pc_i, request_i, answer_i \rangle = \langle pc'_i, request'_i, answer'_i \rangle) \vee \\ (\exists i \in \{1, 2\} : External(i, \langle request_i, answer_i, rbuffer, sbuffer \rangle, \\ \langle request'_i, answer'_i, rbuffer', sbuffer' \rangle) \wedge \\ pc = pc' \wedge \langle sender, waiting, given \rangle = \langle sender', waiting', given' \rangle)$$

# The Verification Task



$$\langle I, R \rangle \models \Box \neg (pc_1 = C \wedge pc_2 = C)$$

*Invariant*(*pc*, *request*, *answer*, *sender*, *given*, *waiting*, *rbuffer*, *sbuffer*) : $\Leftrightarrow$

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

$$(pc(i) = C \vee sbuffer(i) = 1 \vee answer(i) = 1 \Rightarrow$$

$$given = i \wedge$$

$$\forall j : j \neq i \Rightarrow pc(j) \neq C \wedge sbuffer(j) = 0 \wedge answer(j) = 0) \wedge$$

$$(pc(i) = R \Rightarrow$$

$$sbuffer(i) = 0 \wedge answer(i) = 0 \wedge$$

$$(i = given \Leftrightarrow request(i) = 1 \vee rbuffer(i) = 1 \vee sender = i) \wedge$$

$$(request(i) = 0 \vee rbuffer(i) = 0)) \wedge$$

$$(pc(i) = S \Rightarrow$$

$$(sbuffer(i) = 1 \vee answer(i) = 1 \Rightarrow$$

$$request(i) = 0 \wedge rbuffer(i) = 0 \wedge sender \neq i) \wedge$$

$$(i \neq given \Rightarrow$$

$$request(i) = 0 \vee rbuffer(i) = 0)) \wedge$$

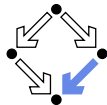
$$(pc(i) = C \Rightarrow$$

$$request(i) = 0 \wedge rbuffer(i) = 0 \wedge sender \neq i \wedge$$

$$sbuffer(i) = 0 \wedge answer(i) = 0) \wedge$$

...

# The Verification Task (Contd)

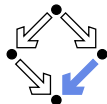


...

$$(sender = 0 \wedge (request(i) = 1 \vee rbuffer(i) = 1) \Rightarrow$$
$$sbuffer(i) = 0 \wedge answer(i) = 0) \wedge$$
$$(sender = i \Rightarrow$$
$$(waiting \neq i) \wedge$$
$$(sender = given \wedge pc(i) = R \Rightarrow$$
$$request(i) = 0 \wedge rbuffer(i) = 0) \wedge$$
$$(pc(i) = S \wedge i \neq given \Rightarrow$$
$$request(i) = 0 \wedge rbuffer(i) = 0) \wedge$$
$$(pc(i) = S \wedge i = given \Rightarrow$$
$$request(i) = 0 \vee rbuffer(i) = 0)) \wedge$$
$$(waiting = i \Rightarrow$$
$$given \neq i \wedge pc_i = S \wedge request_i = 0 \wedge rbuffer(i) = 0 \wedge$$
$$sbuffer_i = 0 \wedge answer(i) = 0) \wedge$$
$$(sbuffer(i) = 1 \Rightarrow$$
$$answer(i) = 0 \wedge request(i) = 0 \wedge rbuffer(i) = 0)$$

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);
```

```
PCs: AXIOM R /= S AND R /= C AND S /= C;
```

```
% 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;
```

```
sender: Index0; sender0: Index0;
```

```
rbuffer: Index -> BOOLEAN; rbuffer0: Index -> BOOLEAN;
```

```
sbuffer: Index -> BOOLEAN; sbuffer0: Index -> BOOLEAN;
```

# The RISC ProofNavigator Theory (Contd)



```
% -----  
% 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, 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'2)



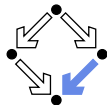
```
% -----  
% transition relation  
% -----
```

```
RC: (PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN)->BOOLEAN =  
  LAMBDA(pc: PC, request: BOOLEAN, answer: BOOLEAN,  
    pc0: PC, request0: BOOLEAN, answer0: BOOLEAN):  
    (pc = R AND (request <=> FALSE) AND  
      pc0 = S AND (request0 <=> TRUE) AND (answer0 <=> answer)) OR  
    (pc = S AND (answer <=> TRUE) AND  
      pc0 = C AND (request0 <=> request) AND (answer0 <=> FALSE)) OR  
    (pc = C AND (request <=> FALSE) AND  
      pc0 = R AND (request0 <=> TRUE) AND (answer0 <=> answer));
```

```
RS: (Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN,  
  Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN)->BOOLEAN =  
  LAMBDA(given: Index0, waiting: Index0, sender: Index0,  
    rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN,  
    given0: Index0, waiting0: Index0, sender0: Index0,  
    rbuffer0: Index->BOOLEAN, sbuffer0: Index->BOOLEAN):
```

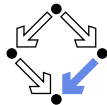


# The RISC ProofNavigator Theory (Contd'3)



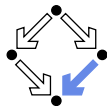
```
(EXISTS(i:Index):
  sender = 0 AND (rbuffer(i) <=> TRUE) AND
  sender0 = i AND (rbuffer0(i) <=> FALSE) AND
  given = given0 AND waiting = waiting0 AND sbuffer = sbuffer0 AND
  (FORALL(j:Index): j /= i => (rbuffer(j) <=> rbuffer0(j)))) OR
(sender /= 0 AND sender = given AND waiting = 0 AND
  given0 = 0 AND sender0 = 0 AND
  waiting = waiting0 AND rbuffer = rbuffer0 AND sbuffer = sbuffer0) OR
(sender /= 0 AND
  sender = given AND waiting /= 0 AND
  (sbuffer(waiting) <=> FALSE) AND
  given0 = waiting AND waiting0 = 0 AND
  (sbuffer0(waiting)<=>TRUE) AND (sender0 = 0) AND
  (rbuffer = rbuffer0) AND
  (FORALL(j:Index): j /= waiting => (sbuffer(j) <=> sbuffer0(j)))) OR
(sender /= 0 AND (sbuffer(sender) <=> FALSE) AND
  sender /= given AND given = 0 AND given0 = sender AND
  (sbuffer0(sender)<=>TRUE) AND sender0=0 AND
  (waiting=waiting0) AND (rbuffer=rbuffer0) AND
  (FORALL(j:Index): j/= sender => (sbuffer(j) <=> sbuffer0(j)))) OR
(sender /= 0 AND sender /= given AND given /= 0 AND
  waiting0 = sender AND sender0 = 0 AND
  given = given0 AND rbuffer = rbuffer0 AND sbuffer = sbuffer0);
```

# The RISC ProofNavigator Theory (Contd'4)



```
External: (Index, PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN,
          Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN,
          Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN)->BOOLEAN =
LAMBDA(i:Index,
  pc: PC, request: BOOLEAN, answer: BOOLEAN,
  pc0: PC, request0: BOOLEAN, answer0: BOOLEAN,
  given: Index0, waiting: Index0, sender: Index0,
  rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN,
  given0: Index0, waiting0: Index0, sender0: Index0,
  rbuffer0: Index->BOOLEAN, sbuffer0: Index->BOOLEAN):
((request <=> TRUE) AND
  pc0 = pc AND (request0 <=> FALSE) AND (answer0 <=> answer) AND
  (rbuffer0(i) <=> TRUE) AND given = given0 AND waiting = waiting0
  AND sender = sender0 AND sbuffer = sbuffer0 AND
  (FORALL (j: Index): j /= i => (rbuffer(j) <=> rbuffer0(j)))) OR
(pc0 = pc AND (request0 <=> request) AND (answer0 <=> TRUE) AND
  (sbuffer(i) <=> TRUE) AND (sbuffer0(i) <=> FALSE) AND
  given = given0 AND waiting = waiting0 AND sender = sender0 AND
  rbuffer = rbuffer0 AND
  (FORALL (j: Index): j /= i => (sbuffer(j) <=> sbuffer0(j))));
```

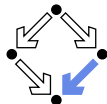
# The RISC ProofNavigator Theory (Contd'5)



Next: BOOLEAN =

```
((EXISTS (i: Index):
  RC(pc(i), request(i), answer(i),
    pc0(i), request0(i), answer0(i)) AND
  (FORALL (j: Index): j /= i =>
    pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
    (answer(j) <=> answer0(j)))) AND
  given = given0 AND waiting = waiting0 AND sender = sender0 AND
  rbuffer = rbuffer0 AND sbuffer = sbuffer0) OR
(RS(given, waiting, sender, rbuffer, sbuffer,
  given0, waiting0, sender0, rbuffer0, sbuffer0) AND
  (FORALL (j: Index): pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
    (answer(j) <=> answer0(j)))) OR
(EXISTS (i: Index):
  External(i, pc(i), request(i), answer(i),
    pc0(i), request0(i), answer0(i),
    given, waiting, sender, rbuffer, sbuffer,
    given0, waiting0, sender0, rbuffer0, sbuffer0) AND
  (FORALL (j: Index): j /= i =>
    pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
    (answer(j) <=> answer0(j)))));
```

# The RISC ProofNavigator Theory (Contd'6)



```
% -----  
% invariant  
% -----  
Invariant: (Index->PC, Index->BOOLEAN, Index->BOOLEAN,  
           Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN) -> BOOLEAN =  
LAMBDA(pc: Index->PC, request: Index->BOOLEAN, answer: Index->BOOLEAN,  
       given: Index0, waiting: Index0, sender: Index0,  
       rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN):  
FORALL (i: Index):  
  (pc(i) = C OR (sbuffer(i) <=> TRUE) OR (answer(i) <=> TRUE) =>  
   given = i AND  
   (FORALL (j: Index): j /= i =>  
    pc(j) /= C AND  
    (sbuffer(j) <=> FALSE) AND (answer(j) <=> FALSE))) AND  
  (pc(i) = R =>  
   (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE) AND  
   (i /= given =>  
    (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i)  
   AND  
   (i = given =>  
    (request(i) <=> TRUE) OR (rbuffer(i) <=> TRUE) OR sender = i) AND  
    ((request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND
```

# 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
```

# The RISC ProofNavigator Theory (Contd'8)

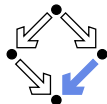


```
(waiting = i =>
  given /= i AND
  pc(waiting) = S AND
  (request(waiting) <=> FALSE) AND (rbuffer(waiting) <=> FALSE) AND
  (sbuffer(waiting) <=> FALSE) AND (answer(waiting) <=> FALSE)) AND
((sbuffer(i) <=> TRUE) =>
  (answer(i) <=> FALSE) AND (request(i) <=> FALSE) AND
  (rbuffer(i) <=> FALSE));
```

# The RISC ProofNavigator Theory (Contd'9)



```
% -----  
% mutual exclusion proof  
% -----  
Mutex: FORMULA  
  Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer) =>  
  NOT(pc(1) = C AND pc(2) = C);  
  
% -----  
% invariance proof  
% -----  
Inv1: FORMULA  
  Initial =>  
  Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer);  
  
Inv2: FORMULA  
  Invariant(pc, request, answer, given, waiting, sender,  
    rbuffer, sbuffer) AND Next =>  
  Invariant(pc0, request0, answer0, given0, waiting0, sender0,  
    rbuffer0, sbuffer0);
```



# The Proofs: MutEx and Inv1

[z3f]: expand Invariant, IC, IS  
[nhn]: scatter  
[znj]: auto  
[niu]: proved (CVCL)

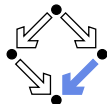
[oas]: expand Initial, Invariant, IC, IS  
[eij]: scatter  
[5ul]: auto  
[uvj]: proved (CVCL)  
[6ul]: auto  
[2u6]: proved (CVCL)  
[avl]: auto  
[cuv]: proved (CVCL)  
[bv1]: auto  
[jtl]: proved (CVCL)  
[cv1]: auto  
[qsb]: proved (CVCL)  
[dvl]: auto  
[xrx]: proved (CVCL)  
[ev1]: auto  
[5qn]: proved (CVCL)  
[fv1]: auto  
[fqd]: proved (CVCL)  
[gvl]: auto  
[mpz]: proved (CVCL)  
[hv1]: proved (CVCL)  
[h5h]: auto  
[p3z]: proved (CVCL)  
[i5h]: auto  
[gjb]: proved (CVCL)  
[j5h]: auto  
[4vi]: proved (CVCL)  
[k5h]: auto  
[ucq]: proved (CVCL)  
[l5h]: auto  
[lpx]: proved (CVCL)

[m5h]: proved (CVCL)  
[n5h]: proved (CVCL)  
[o5h]: proved (CVCL)  
[p5h]: proved (CVCL)  
[q5h]: proved (CVCL)  
[q5i]: proved (CVCL)  
[r5i]: proved (CVCL)  
[s5i]: proved (CVCL)  
[t5i]: proved (CVCL)  
[u5i]: auto  
[1br]: proved (CVCL)  
[v5i]: auto  
[roy]: proved (CVCL)  
[w5i]: auto  
[i26]: proved (CVCL)  
[x5i]: proved (CVCL)  
[y5i]: auto  
[wuo]: proved (CVCL)  
[z5i]: auto  
[nbw]: proved (CVCL)  
[z5j]: auto  
[nbn]: proved (CVCL)  
[15j]: auto  
[eou]: proved (CVCL)  
[25j]: proved (CVCL)  
[35j]: proved (CVCL)  
[45j]: proved (CVCL)  
[55j]: proved (CVCL)  
[65j]: proved (CVCL)

Single application  
of autostar.



# The Proofs: Inv2



```
[pas]: scatter
  [lbh]: expand Next
  [pzi]: split bfv
  [leh]: decompose
  [pkr]: expand RS
  [lpn]: split 5xv
  [pt6]: expand Invariant
  [lcw]: scatter
  [puh]: auto
  [l43]: proved (CVCL)
  ... (20 times)
  [tuh]: proved (CVCL)
  ... (15 times)
  [qt6]: expand Invariant
  [snq]: scatter
  [avi]: auto
  [cct]: proved (CVCL)[meh]: scatter
  ... (26 times)
  [gvi]: proved (CVCL)
  ... (6 times)
  [rt6]: scatter
  [zyk]: expand Invariant
  [rvj]: scatter
  [zgj]: auto
  [rhd]: proved (CVCL)
  ... (31 times)
  [2f3]: proved (CVCL)
  ... (1 times)

[st6]: scatter
  [aef]: expand Invariant
  [cwk]: scatter
  [ql6]: auto
  [seg]: proved (CVCL)
  ... (21 times)
  [wl6]: proved (CVCL)[neh]: scatter
  ... (12 times)
  [tt6]: scatter
  [hp6]: expand Invariant
  [twl]: scatter
  [hqv]: auto
  [tbj]: proved (CVCL)
  ... (27 times)
  [nqv]: proved (CVCL)
  ... (6 times)
  [w3z]: expand External
  [3rk]: split lhe
  [g4b]: scatter
  [mdh]: expand Invariant
  [wzf]: scatter
  [3ys]: auto
  [gsh]: proved (CVCL)
  ... (36 times)

[h4b]: scatter
  [tob]: expand Invariant
  [hlg]: scatter
  [t4i]: auto
  [hpk]: proved (CVCL)
  ... (36 times)
  [4oc]: expand RC
  [nuh]: split nwz
  [4ge]: scatter
  [ney]: expand Invariant
  [45d]: scatter
  [nui]: auto
  [4wr]: proved (CVCL)
  ... (36 times)
  [5ge]: scatter
  [ups]: expand Invariant
  [o6e]: scatter
  [ez5]: auto
  [5tu]: proved (CVCL)
  ... (36 times)
  [6ge]: scatter
  [21m]: expand Invariant
  [66f]: scatter
  [24u]: auto
  [6qx]: proved (CVCL)
  ... (36 times)
```

Ten main branches each requiring only single application of autostar.