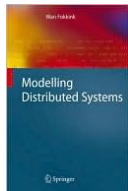


# Protocol Validation with $\mu$ CRL



Wan Fokkink  
Modelling Distributed Systems  
2nd edition

# Goals

- ▶ Formal modeling of real-life protocols
- ▶ Automated analysis of protocols by state space exploration (i.e. simulation or model checking)
- ▶ Give an impression of the difficulty in analysis (state space explosion), and how to overcome this
- ▶ Algorithms for automated protocol analysis
- ▶ Symbolic verification of protocols using equational logic and theorem provers
- ▶ Insight into protocol design
- ▶ Insight into concurrency

# Algebraic Specification

An **algebraic specification** of data types consists of

- ▶ a **signature**, i.e. function symbols from which one can build **terms**
- ▶ **axioms**, i.e. equations between terms, inducing an equality relation on terms (closed under:  
(1) **equivalence**, (2) **substitution**, and (3) **context**)

## Algebraic Specification - Example

The signature of the **natural numbers** consists of constant **0**, unary successor **S**, and binary addition **plus** and multiplication **mul**.

The axioms are:

$$\text{plus}(n, 0) = n$$

$$\text{plus}(n, S(m)) = S(\text{plus}(n, m))$$

$$\text{mul}(n, 0) = 0$$

$$\text{mul}(n, S(m)) = \text{plus}(\text{mul}(n, m), n)$$

The axioms are directed from left to right, and must constitute a *terminating* rewrite system.

There is an MSc course *Term Rewriting Systems*.

# Constructors

$\mu$ CRL uses algebraic specification of data, with explicit recognition of **constructor** symbols, which can't be eliminated from data terms.

**Example:** For the natural numbers,  $0$  and  $S$  are constructors, while  $plus$  and  $mul$  aren't.

```
sort    $Nat$ 
func    $0 : \rightarrow Nat$ 
          $S : Nat \rightarrow Nat$ 
map     $plus, mul : Nat \times Nat \rightarrow Nat$ 
var     $n, m : Nat$ 
rew     $plus(n, 0) = n$ 
          $plus(n, S(m)) = S(plus(n, m))$ 
          $mul(n, 0) = 0$ 
          $mul(n, S(m)) = plus(mul(n, m), n)$ 
```

# Innermost Rewriting

Rewriting of data terms is performed according to the **innermost** strategy, meaning that a term  $f(d_1, \dots, d_n)$  can only be rewritten if  $d_1, \dots, d_n$  are **normal forms**.

A normal form only consists of constructor symbols.

# Booleans

'true' and 'false', together with conjunction, disjunction and negation must be declared in each  $\mu$ CRL specification.

```
sort   Bool
func    $\top, \text{F} : \rightarrow \text{Bool}$ 
map     $\wedge, \vee : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$ 
          $\neg : \text{Bool} \rightarrow \text{Bool}$ 
var     $b : \text{Bool}$ 
rew     $b \wedge \top = b$ 
          $b \wedge \text{F} = \text{F}$ 
          $b \vee \top = \top$ 
          $b \vee \text{F} = b$ 
          $\neg \top = \text{F}$ 
          $\neg \text{F} = \top$ 
```

# If-then-else Function

A help function

$$if : Bool \times D \times D \rightarrow D$$

with as equations

$$if(T, d, e) = d$$

$$if(F, d, e) = e$$

is often convenient in the specification of data types.

# Equality Function

One needs to define an equality function  $eq : D \times D \rightarrow Bool$  for data types  $D$ , where  $eq(d, e) = T$  if and only if  $d = e$ .

Example:

**map**  $eq : Bool \times Bool \rightarrow Bool$

**rew**  $eq(T, T) = T$

$eq(F, F) = T$

$eq(T, F) = F$

$eq(F, T) = F$

**map**  $eq : Nat \times Nat \rightarrow Bool$

**var**  $n, m : Nat$

**rew**  $eq(0, 0) = T$

$eq(S(n), S(m)) = eq(n, m)$

$eq(0, S(n)) = F$

$eq(S(n), 0) = F$

# Equality Function

A shorter specification of the equality function on booleans is:

$$eq(b, b) = T$$

$$eq(T, F) = F$$

$$eq(F, T) = F$$

The following specification of an equality function doesn't work in  $\mu$ CRL:

$$eq(x, x) = T$$

$$eq(x, y) = F$$

That is, rewrite rules aren't necessarily 'executed' from top to bottom.

# Induction

One can prove properties of data terms by **induction** on **constructors**.

**Example:** We prove by induction that  $\neg\neg b = b$  for all booleans  $b$ .

$$[b \text{ is T}] \neg\neg T = \neg F = T$$

$$[b \text{ is F}] \neg\neg F = \neg T = F$$

**Example:** We prove by induction that  $plus(0, n) = n$  for all natural numbers  $n$ .

$$[\text{Base case, } n \text{ is } 0] plus(0, 0) = 0$$

$$[\text{Inductive case, } n \text{ is } S(m)] plus(0, S(m)) = S(plus(0, m)) = S(m)$$

# Basic Process Terms

**Basic process terms** are built from **parametrized actions** in a set  $\text{Act}$ , **alternative composition** and **sequential composition**.

- ▶ An action name  $a \in \text{Act}$  represents indivisible behavior.  
It can carry data parameters:  $a(d_1, \dots, d_n)$ .
- ▶ The process term  $p + q$  executes the behavior of either  $p$  or  $q$ .
- ▶ The process term  $p \cdot q$  first executes  $p$ , and upon termination proceeds to execute  $q$ .

# Basic Process Terms - Example

$((a + b) \cdot c) \cdot d$  represents the state space

$((a + b) \cdot c) \cdot d$



$c \cdot d$



$d$



$\checkmark$

# Basic Process Terms - Transition Rules

The link between a process term  $p$  and its transitions  $p \xrightarrow{a} p'$  and  $p \xrightarrow{a} \surd$  can be formally defined by **transition rules**.

$$\overline{a(\vec{d}) \xrightarrow{\quad} \surd}$$

$$\frac{x_1 \xrightarrow{a(\vec{d})} \surd}{x_1 + x_2 \xrightarrow{a(\vec{d})} \surd} \quad \frac{x_1 \xrightarrow{a(\vec{d})} y}{x_1 + x_2 \xrightarrow{a(\vec{d})} y} \quad \frac{x_2 \xrightarrow{a(\vec{d})} \surd}{x_1 + x_2 \xrightarrow{a(\vec{d})} \surd} \quad \frac{x_2 \xrightarrow{a(\vec{d})} y}{x_1 + x_2 \xrightarrow{a(\vec{d})} y}$$

$$\frac{x_1 \xrightarrow{a(\vec{d})} \surd}{x_1 \cdot x_2 \xrightarrow{a(\vec{d})} x_2} \quad \frac{x_1 \xrightarrow{a(\vec{d})} y}{x_1 \cdot x_2 \xrightarrow{a(\vec{d})} y \cdot x_2}$$

## Basic Process Algebra - Axioms

$$x + y = y + x$$

$$(x + y) + z = x + (y + z)$$

$$x + x = x$$

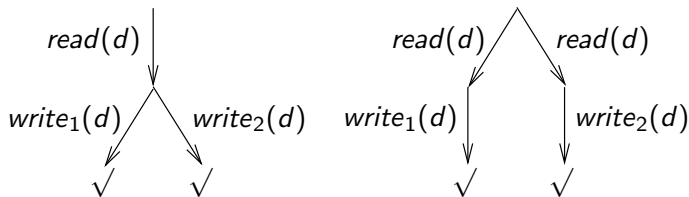
$$(x + y) \cdot z = (x \cdot z) + (y \cdot z)$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

# No Left Distributivity

$x \cdot (y + z) \neq (x \cdot y) + (x \cdot z)$  doesn't hold.

Example:



The left process reads  $d$ , and then decides whether it writes  $d$  on disc 1 or 2.

The right process makes a choice for disc 1 or 2 before it reads  $d$ .

If disc 1 crashes, the left process saves datum  $d$  on disc 2, while the right process may get stuck.

# Bisimulation Equivalence

$\downarrow$  is a special predicate on states, expressing **successful termination**.  
That is,  $\checkmark$  is the only state where  $\downarrow$  holds.

Assume a state space. A **bisimulation** is a binary relation  $\mathcal{B}$  on states such that if  $s_1 \mathcal{B} s_2$ , then:

1.  $s_1 \xrightarrow{a} s'_1$  implies that there is a transition  $s_2 \xrightarrow{a} s'_2$  with  $s'_1 \mathcal{B} s'_2$
2.  $s_2 \xrightarrow{a} s'_2$  implies that there is a transition  $s_1 \xrightarrow{a} s'_1$  with  $s'_1 \mathcal{B} s'_2$
3.  $s_1 \downarrow$  implies  $s_2 \downarrow$
4.  $s_2 \downarrow$  implies  $s_1 \downarrow$

Two states  $s_1$  and  $s_2$  are **bisimilar**, denoted  $s_1 \leftrightarrow s_2$ , if there is a bisimulation relation  $\mathcal{B}$  such that  $s_1 \mathcal{B} s_2$ .

**Example:**  $a \cdot (b + c) \not\leftrightarrow (a \cdot b) + (a \cdot c)$

# Soundness and Completeness of the Axioms

**Theorem:** For basic process algebra terms  $p$  and  $q$ :

$$p = q \Leftrightarrow p \underline{\leftrightarrow} q$$

To specify that two action names can **communicate** (or **synchronize**):

$$\mathbf{comm} \quad a|b = c$$

Communication is **commutative** and **associative**.

$$\begin{aligned} a|b &= b|a \\ (a|b)|c &= a|(b|c) \end{aligned}$$

Actions  $a(d_1, \dots, d_n)$  and  $b(e_1, \dots, e_m)$  can only communicate if they carry exactly the same data parameters.

In  $\mu\text{CRL}$ , the equality function only needs to be defined for data types that are used in parameters of actions that can communicate.

# Parallelism

The **merge**  $\parallel$  executes the two process terms in its arguments in parallel.

For example, if action names  $a$  and  $b$  don't communicate,

$$a \parallel b = a \cdot b + b \cdot a$$

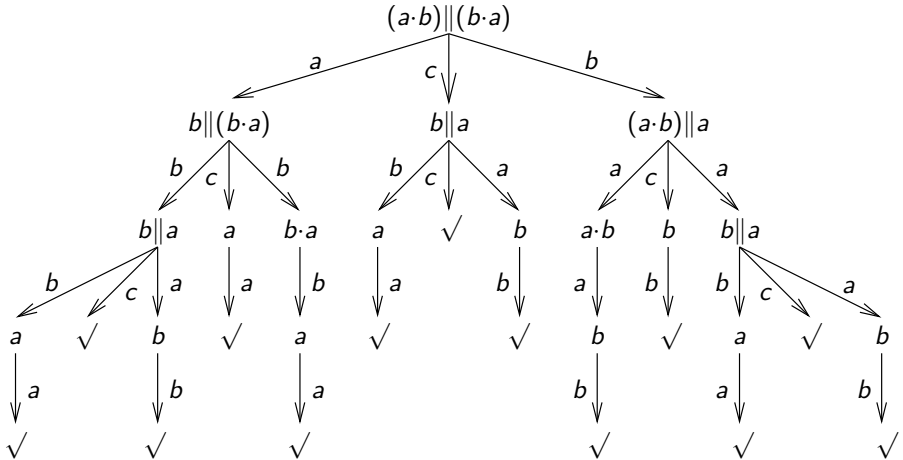
The merge can also execute a communication between actions of its arguments.

For example, if  $a|b = c$ ,

$$a \parallel b = (a \cdot b + b \cdot a) + c$$

# Parallelism - Example

If all communications between action names result to  $c$ , then



# Left Merge and Communication Merge

The **left merge**  $\ll$  executes an action of its first argument and then behaves as the merge.

The **communication merge**  $|$  executes a communication of actions of its two arguments and then behaves as the merge.

**Example:** If  $a|b = c$ , then

$$a \ll b = a \cdot b$$

$$a|b = c$$

These operators are needed to axiomatize the merge:

$$p \parallel q = (p \ll q + q \ll p) + p|q$$

## Left Merge and Communication Merge

$(x + y) \ll z = (x \ll z) + (y \ll z)$  holds.

$x \ll (y + z) = (x \ll y) + (x \ll z)$  doesn't hold.

$(x + y) | z = (x | z) + (y | z)$  holds.

$x | (y + z) = (x | y) + (x | z)$  holds.

## Parallelism - Axioms

$$x \parallel y = (x \perp\!\!\!\perp y + y \perp\!\!\!\perp x) + x | y$$

$$a(\vec{d}) \perp\!\!\!\perp y = a(\vec{d}) \cdot y \quad (\vec{d} \text{ denotes } d_1, \dots, d_n)$$

$$(a(\vec{d}) \cdot x) \perp\!\!\!\perp y = a(\vec{d}) \cdot (x \parallel y)$$

$$(x + y) \perp\!\!\!\perp z = x \perp\!\!\!\perp z + y \perp\!\!\!\perp z$$

$$a(\vec{d}) | b(\vec{d}) = c(\vec{d}) \quad \text{if } a | b = c$$

$$a(\vec{d}) | b(\vec{e}) = \delta \quad \text{if } \vec{d} \neq \vec{e} \text{ or } a | b \text{ is undefined}$$

$$(a(\vec{d}) \cdot x) | b(\vec{e}) = (a(\vec{d}) | b(\vec{e})) \cdot x$$

$$a(\vec{d}) | (b(\vec{e}) \cdot y) = (a(\vec{d}) | b(\vec{e})) \cdot y$$

$$(a(\vec{d}) \cdot x) | (b(\vec{e}) \cdot y) = (a(\vec{d}) | b(\vec{e})) \cdot (x \parallel y)$$

$$(x + y) | z = x | z + y | z$$

$$x | (y + z) = x | y + x | z$$

# Deadlock and Encapsulation

- \* The **deadlock**  $\delta$  doesn't display any behavior.
- \* The **encapsulation** operators  $\partial_H$ , for sets of actions  $H$ , rename all actions of  $H$  in their argument into  $\delta$ .

Encapsulation operators enable to enforce actions into communication, meaning that send and read actions can't occur in isolation.

**Example:** Let  $s|r = c$ .

$$\begin{aligned} s \parallel r &= (s \cdot r + r \cdot s) + c \\ \partial_{\{s,r\}}(s \parallel r) &= c \end{aligned}$$

# Deadlock and Encapsulation - Axioms

$$x + \delta = x$$

$$\delta \cdot x = \delta$$

$$\partial_H(\delta) = \delta$$

$$\partial_H(a(\vec{d})) = a(\vec{d}) \quad \text{if } a \notin H$$

$$\partial_H(a(\vec{d})) = \delta \quad \text{if } a \in H$$

$$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$$

$$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$$

## Example - Bits Through a Channel

A bit 0 or 1 is sent into a channel:  $s(0) + s(1)$

The bit is received at the other side of the channel:  $r(0) + r(1)$

The communication of  $s$  and  $r$  is  $c$ .

The behavior of the channel is described by

$$\partial_{\{s,r\}}((s(0) + s(1)) \parallel (r(0) + r(1)))$$

The encapsulation operator enforces that  $s(d)$  and  $r(d)$  can only occur in communication.

We use the axioms to equate the process term above to

$$c(0) + c(1)$$

## Example - Bits Through a Channel

$$\begin{aligned} & (s(0) + s(1)) \parallel (r(0) + r(1)) \\ = & (s(0) + s(1)) \perp (r(0) + r(1)) + (r(0) + r(1)) \perp (s(0) + s(1)) + \\ & (s(0) + s(1)) | (r(0) + r(1)) \\ = & s(0) \perp (r(0) + r(1)) + s(1) \perp (r(0) + r(1)) + \\ & r(0) \perp (s(0) + s(1)) + r(1) \perp (s(0) + s(1)) + \\ & s(0) | r(0) + s(0) | r(1) + s(1) | r(0) + s(1) | r(1) \\ = & s(0) \cdot (r(0) + r(1)) + s(1) \cdot (r(0) + r(1)) + \\ & r(0) \cdot (s(0) + s(1)) + r(1) \cdot (s(0) + s(1)) + c(0) + \delta + \delta + c(1) \\ = & s(0) \cdot (r(0) + r(1)) + s(1) \cdot (r(0) + r(1)) + \\ & r(0) \cdot (s(0) + s(1)) + r(1) \cdot (s(0) + s(1)) + c(0) + c(1) \end{aligned}$$

## Example - Bits Through a Channel

Let  $H$  denote  $\{s, r\}$ .

$$\begin{aligned} & \partial_H((s(0) + s(1)) \parallel (r(0) + r(1))) \\ = & \partial_H(s(0) \cdot (r(0) + r(1)) + s(1) \cdot (r(0) + r(1)) + \\ & r(0) \cdot (s(0) + s(1)) + r(1) \cdot (s(0) + s(1)) + c(0) + c(1)) \\ = & \partial_H(s(0)) \cdot \partial_H(r(0) + r(1)) + \partial_H(s(1)) \cdot \partial_H(r(0) + r(1)) + \\ & \partial_H(r(0)) \cdot \partial_H(s(0) + s(1)) + \partial_H(r(1)) \cdot \partial_H(s(0) + s(1)) + \\ & \partial_H(c(0)) + \partial_H(c(1)) \\ = & \delta \cdot \partial_H(r(0) + r(1)) + \delta \cdot \partial_H(r(0) + r(1)) + \\ & \delta \cdot \partial_H(s(0) + s(1)) + \delta \cdot \partial_H(s(0) + s(1)) + c(0) + c(1) \\ = & \delta + \delta + \delta + \delta + c(0) + c(1) \\ = & c(0) + c(1) \end{aligned}$$

# Process Declaration



This process can be captured by means of:  $X = a \cdot Y$   
 $Y = b \cdot X$

$X$  and  $Y$  represent the two states of the process.

A **process declaration** **proc** consists of **recursive equations**

$$X(d_1:D_1, \dots, d_n:D_n) = p$$

where the process term  $p$  may contain expressions  $Y(e_1, \dots, e_m)$ .

The **initial declaration** **init**, i.e. the initial state of the specification, is an expression  $X(t_1, \dots, t_n)$  with  $t_1, \dots, t_n$  *closed* data terms.

## Process Declaration - Example

The process *Clock* repeatedly performs action *tick* or displays the current time.

**act**    *tick*  
          *display* : *Nat*

**proc**     $Clock(n:Nat) = tick \cdot Clock(S(n)) + display(n) \cdot Clock(n)$

**init**     $Clock(0)$

'Unguarded' process declarations such as  $X = X$  and  $Y = Y \cdot a$  are illegal.

# Conditional

The process term  $p \triangleleft b \triangleright q$ , where  $p$  and  $q$  are process terms, and  $b$  a data term of sort *Bool*, behaves as  $p$  if  $b = T$  and as  $q$  if  $b = F$ .

$$x \triangleleft T \triangleright y = x$$

$$x \triangleleft F \triangleright y = y$$

**Example:** The process *Counter* counts the number of *a*-actions, resetting the internal counter after three *a*'s:

**act**  $a, \text{reset}$

**proc**  $\text{Counter}(n:\text{Nat}) =$   
 $a \cdot \text{Counter}(S(n)) \triangleleft n < S(S(S(0))) \triangleright \text{reset} \cdot \text{Counter}(0)$

**init**  $\text{Counter}(0)$

# Summation over a Data Type

The **sum** operator  $\sum_{d:D} P(d)$  behaves as

$$P(d_1) + P(d_2) + \dots$$

i.e. as the (possibly infinite) choice between process terms  $P(d)$  for data terms  $d$  that can be built from the *constructors* of  $D$ .

In  $\mu$ CRL, the distinction between **func** and **map** is used to build the set of constructor terms for summation over a data type.

## Summation - Axioms

$$\sum_{d:D} x = x$$

$$\sum_{d:D} P(d) = \sum_{d:D} P(d) + P(d_0) \quad (d_0 \in D)$$

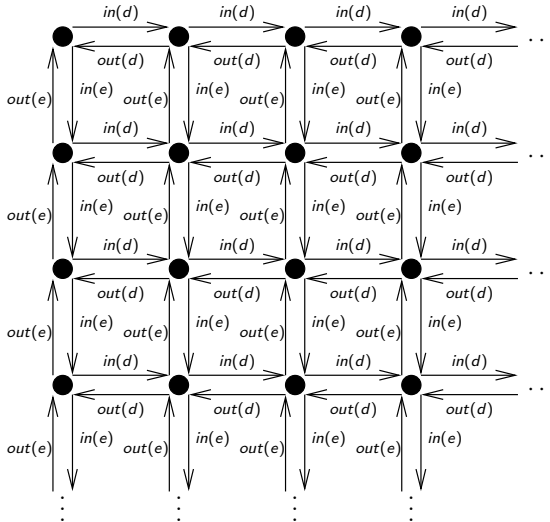
$$\sum_{d:D} (P(d) + Q(d)) = \sum_{d:D} P(d) + \sum_{d:D} Q(d)$$

$$(\sum_{d:D} P(d)) \cdot x = \sum_{d:D} (P(d) \cdot x)$$

$$(\forall d:D P(d) = Q(d)) \Rightarrow \sum_{d:D} P(d) = \sum_{d:D} Q(d)$$

# Example - Unbounded Bag

We can put elements of sort  $D$  into a bag, and collect these elements from the bag in arbitrary order. For example, if  $D$  is  $\{d, e\}$ :



## Example - Unbounded Bag

If  $D$  is  $\{d, e\}$ , then a  $\mu$ CRL specification of the bag is:

```
act    $in, out : D$   
proc   $Y(n: Nat, m: Nat) = in(d) \cdot Y(S(n), m)$   
       $+ in(e) \cdot Y(n, S(m))$   
       $+ (out(d) \cdot Y(P(n), m) \triangleleft n > 0 \triangleright \delta)$   
       $+ (out(e) \cdot Y(n, P(m)) \triangleleft m > 0 \triangleright \delta)$   
  
init   $Y(0, 0)$ 
```

where  $P(S(n)) = n$  (and  $P(0)$  is undefined!).

An alternative  $\mu$ CRL specification that works for general  $D$  is:

```
act    $in, out : D$   
proc   $X = \sum_{d:D} in(d) \cdot (X \parallel out(d))$   
init   $X$ 
```

# Hidden Action and Hiding

The **hidden action**  $\tau$  represents an internal computation step.

**Example:**  $a \cdot \tau \cdot b = a \cdot b$

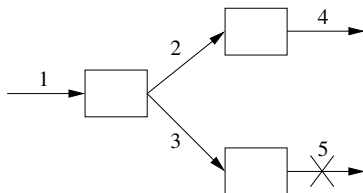
$\tau$  doesn't communicate with any action names.

A **hiding** operator  $\tau_I$ , with  $I \subseteq \text{Act}$ , renames all actions from  $I$  in its argument into  $\tau$ .

**Example:**  $\tau_{\{c\}}(a \cdot c \cdot b) = a \cdot \tau \cdot b = a \cdot b$

# Not All Hidden Actions Are Inert

**Example:** A malfunctioning channel.



$$\tau_{\{c_2, c_3\}}(\partial_{\{s_5\}}(r_1 \cdot (c_2 \cdot s_4 + c_3 \cdot s_5))) = r_1 \cdot (\tau \cdot s_4 + \tau \cdot \delta) \neq r_1 \cdot s_4$$

## Which Hidden Actions Are Inert? - Part I

$$a \cdot (b + \tau \cdot \delta) \neq a \cdot b$$

$$\partial_{\{c\}}(a \cdot (b + \tau \cdot c)) \neq \partial_{\{c\}}(a \cdot (b + c))$$

$$a \cdot (b + \tau \cdot c) \neq a \cdot (b + c)$$

**Solution:** A hidden action is inert if it **doesn't lose possible behavior**.

**Example:**  $a \cdot (b + \tau \cdot (b + c)) = a \cdot (b + c)$

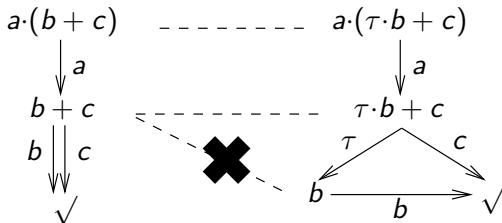
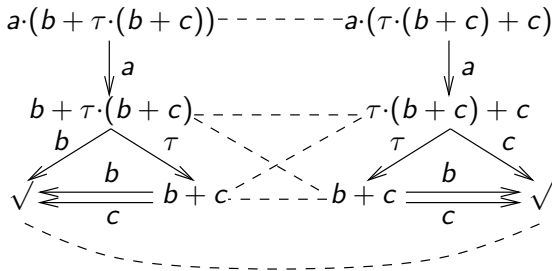
# Branching Bisimulation Equivalence

$s_1$  and  $s_2$  are **branching bisimilar** states, denoted by  $s_1 \leftrightarrow_b s_2$ , if for each transition  $s_1 \xrightarrow{a} s'_1$  (or  $s_1 \downarrow$ ):

- \* either  $a = \tau$  and  $s'_1 \leftrightarrow_b s_2$  (i.e.,  $s_1 \xrightarrow{\tau} s'_1$  is inert);
- \* or there exist transitions  $s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{s}_2$ , where  $s_1 \leftrightarrow_b \hat{s}_2$  and  $\hat{s}_2 \xrightarrow{a} s'_2$  with  $s'_1 \leftrightarrow_b s'_2$  (or  $\hat{s}_2 \downarrow$ )

and **vice versa**.

# Branching Bisimulation - Examples



## Which Hidden Actions Are Inert? - Part II

Initial  $\tau$ 's aren't inert.

$$a \cdot (b + \tau \cdot c) \neq a \cdot (b + c)$$

$$\tau \cdot c \neq c$$

**Solution:** A hidden action is inert if it doesn't lose possible behavior and isn't initial.

# Rooted Branching Bisimulation Equivalence

$s_1$  and  $s_2$  are **rooted branching bisimilar**, denoted  $s_1 \leftrightarrow_{rb} s_2$ , if:

1.  $s_1 \xrightarrow{a} s'_1$  implies that there is a transition  $s_2 \xrightarrow{a} s'_2$  with  $s'_1 \leftrightarrow_b s'_2$
2.  $s_1 \downarrow$  implies  $s_2 \downarrow$

and **vice versa**.

## Hidden Action and Hiding - Axioms

$$x \cdot \tau = x$$

$$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$$

$$\tau_I(\delta) = \delta$$

$$\tau_I(\tau) = \tau$$

$$\tau_I(a(\vec{d})) = a(\vec{d}) \quad \text{if } a \notin I$$

$$\tau_I(a(\vec{d})) = \tau \quad \text{if } a \in I$$

$$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$$

$$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$$

# Soundness and Completeness of the Axioms

A **sound** and **complete axiomatization** is given in the textbook.

**Theorem:** For process algebra terms  $p$  and  $q$ :

$$p = q \Leftrightarrow p \leftrightarrow_{rb} q$$

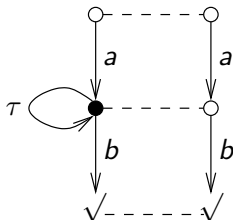
# Fair Abstraction

$\tau$ -loops can be eliminated.

Example: The process  $X$  with

$$\begin{aligned} X &= a \cdot Y \\ Y &= \tau \cdot Y + b \end{aligned}$$

is rooted branching bisimilar to  $a \cdot b$ .



In the black state there is a fixed chance  $\alpha > 0$  that the  $b$ -transition is taken. So the chance that  $b$  is eventually executed is 100%.

# Overview

- \* data types: (**sort func map var rew**)
- \* action declaration: (**act**  $a : D_1 \times \dots \times D_n$ , **comm**  $a \mid b = c$ )
- \* basic operators: ( $+$   $\cdot$ )
- \* data-dependent operators: ( $\langle b \rangle \sum_{d:D}$ )
- \* process declaration: (**proc**  $X(d_1, \dots, d_n)$ )
- \* parallel operators: ( $\parallel \ll \mid$ )
- \* deadlock and encapsulation: ( $\delta \partial_H$ )
- \* hidden action and hiding: ( $\tau \tau_I$ )

In general, the **init** declaration of a  $\mu$ CRL specification is of the form

$$\tau_I(\partial_H(X_1(d_1, \dots, d_n) \parallel \dots \parallel X_k(e_1, \dots, e_m)))$$

where the recursive equations for  $X_1, \dots, X_k$  use only data, actions, basic operators, and data-dependent operators.

# Model Checking Versus Symbolic Correctness Proofs

In *model checking*, the state space is generated, and logical properties are checked automatically.

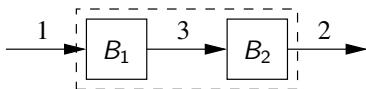
- ▶ automated, so convenient to use
- ▶ expressive logic for specifying properties
- ▶ entire state space is searched
- ▶ suffers from state explosion
- ▶ works only for fixed data sets and topologies

Symbolic correctness proofs can be supported by a *theorem prover* (as taught in the MSc course *Logical Verification*).

- ▶ laborious
- ▶ no generation of the state space
- ▶ provides general correctness proof

## Example - One-bit Buffers in Sequence

**act**  $r_1, s_2, r_3, s_3, c_3 : D$   
**comm**  $r_3 \mid s_3 = c_3$   
**proc**  $B_1 = \sum_{d:D} r_1(d) \cdot s_3(d) \cdot B_1$   
 $B_2 = \sum_{d:D} r_3(d) \cdot s_2(d) \cdot B_2$   
**init**  $\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2))$



Buffers  $B_1$  and  $B_2$  of capacity one in sequence behave as a buffer of capacity two:

**proc**  $X = \sum_{d:D} r_1(d) \cdot Y(d)$   
 $Y(d:D) = \sum_{d':D} r_1(d') \cdot Z(d, d') + s_2(d) \cdot X$   
 $Z(d:D, d':D) = s_2(d) \cdot Y(d')$   
**init**  $X$

## Example - One-bit Buffers in Sequence

In the  $\mu$ CRL specification of two one-bit buffers in sequence, the following data types are needed:

- \*  $Bool$  with constructors  $T$  and  $F$ , and with mappings  $and$ ,  $or$ ,  $not$ ,  $eq$ .
- \*  $D$  with constructors  $d_1$  and  $d_2$ , and with mapping  $eq$ .

Furthermore, the following action declarations are needed:

- \*  $r_1, s_2, r_3, s_3, c_3 : D$
- \*  $r_3 | s_3 = c_3$

## Symbolic Proof Example: One-bit Buffers in Sequence

$$\begin{aligned} & B_1 \parallel B_2 && \text{(Summations } \Sigma_{d:D} \text{ and data parameters } d \text{ are omitted)} \\ = & B_1 \ll B_2 + B_2 \ll B_1 + B_1 | B_2 \\ = & (r_1 \cdot s_3 \cdot B_1) \ll B_2 + (r_3 \cdot s_2 \cdot B_2) \ll B_1 + (r_1 \cdot s_3 \cdot B_1) | (r_3 \cdot s_2 \cdot B_2) \\ = & r_1 \cdot ((s_3 \cdot B_1) \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel B_1) + \delta \cdot ((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) \\ = & r_1 \cdot ((s_3 \cdot B_1) \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel B_1) \end{aligned}$$

$$\begin{aligned} & \partial_{\{s_3, r_3\}}(B_1 \parallel B_2) \\ = & \partial_{\{s_3, r_3\}}(r_1 \cdot ((s_3 \cdot B_1) \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel B_1)) \\ = & \partial_{\{s_3, r_3\}}(r_1 \cdot ((s_3 \cdot B_1) \parallel B_2)) + \partial_{\{s_3, r_3\}}(r_3 \cdot ((s_2 \cdot B_2) \parallel B_1)) \\ = & \partial_{\{s_3, r_3\}}(r_1) \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2) + \partial_{\{s_3, r_3\}}(r_3) \cdot \partial_{\{s_3, r_3\}}((s_2 \cdot B_2) \parallel B_1) \\ = & r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2) + \delta \cdot \partial_{\{s_3, r_3\}}((s_2 \cdot B_2) \parallel B_1) \\ = & r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2) \end{aligned}$$

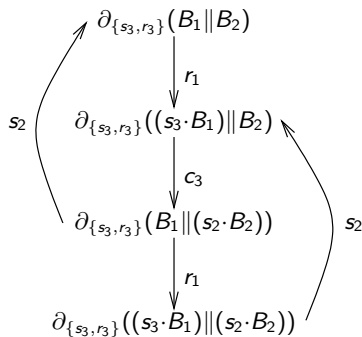
# One-bit Buffers in Sequence

Likewise we can derive:

$$\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2) = c_3 \cdot \partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2))$$

$$\partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2)) = s_2 \cdot \partial_{\{s_3, r_3\}}(B_1 \parallel B_2) + r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))$$

$$\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) = s_2 \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2)$$



## One-bit Buffers in Sequence

$$\begin{aligned}\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2)) &= \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\ &= r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\ &= r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\ &= r_1 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\ &= r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2)))\end{aligned}$$

Likewise we can derive:

$$\begin{aligned}\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2))) &= s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2)) \\ &+ r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))) \\ \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))) &= s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\ &= s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel (s_2 \cdot B_2)))\end{aligned}$$

# Renaming

The **renaming** operators  $\rho_f$ , for mappings  $f : \text{Act} \rightarrow \text{Act}$ , rename all action names  $a$  in their argument into  $f(a)$ .

$a$  and  $f(a)$  must carry the same data types.

$$\rho_f(\delta) = \delta$$

$$\rho_f(\tau) = \tau$$

$$\rho_f(a(\vec{d})) = f(a)(\vec{d})$$

$$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$$

$$\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y)$$

## Four One-bit Buffers in Sequence

**act**  $r_1, s_2, r_3, s_3, c_3 : D$

**comm**  $r_3 \mid s_3 = c_3$

**proc**  $B_1 = \sum_{d:D} r_1(d) \cdot s_3(d) \cdot B_1$

$B_2 = \sum_{d:D} r_3(d) \cdot s_2(d) \cdot B_2$

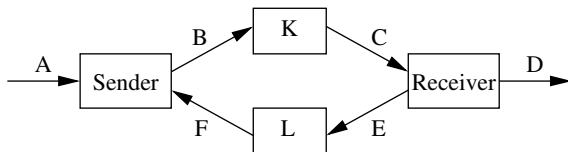
$C = \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2))$

**act**  $r_4, s_4, c_4 : D$

**comm**  $s_4 \mid r_4 = c_4$

**init**  $\tau_{\{c_4\}}(\partial_{\{s_4, r_4\}}(\rho_{s_2 \rightarrow s_4}(C) \parallel \rho_{r_1 \rightarrow r_4}(C)))$

# Alternating Bit Protocol



Data elements are sent from *Sender* to *Receiver* via a corrupted channel. *Sender* alternately attaches bit 0 or 1 to data elements.

If *Receiver* receives a datum, it sends the attached bit to *Sender* via a corrupted channel, to acknowledge reception. If *Receiver* receives an error message, then it resends the preceding acknowledgement.

*Sender* keeps sending a datum with attached bit  $b$  until it receives acknowledgement  $b$ . Then it starts sending the next datum with attached bit  $1-b$  until it receives acknowledgement  $1-b$ , etc.

# Alternating Bit Protocol - Sender and Receiver

$S_b$  sends a datum with bit  $b$  attached:

$$S_b = \sum_{d:\Delta} r_A(d) \cdot s_B(d, b) \cdot T_{db}$$

$$\begin{aligned} T_{db} &= r_F(b) \cdot S_{1-b} \\ &+ (r_F(1-b) + r_F(\perp)) \cdot s_B(d, b) \cdot T_{db} \end{aligned}$$

$R_b$  expects to receive a datum with  $b$  attached:

$$\begin{aligned} R_b &= \sum_{d:\Delta} r_C(d, b) \cdot s_D(d) \cdot s_E(b) \cdot R_{1-b} \\ &+ \sum_{d:\Delta} r_C(d, 1-b) \cdot s_E(1-b) \cdot R_b \\ &+ r_C(\perp) \cdot s_E(1-b) \cdot R_b \end{aligned}$$

# Alternating Bit Protocol - Channels

$K$  and  $L$  represent the corrupted channels, to model *asynchronous* communication. (The action  $j$  represents an internal choice.)

$$K = \sum_{d:\Delta} \sum_{b:\{0,1\}} r_B(d, b) \cdot (j \cdot s_C(d, b) + j \cdot s_C(\perp)) \cdot K$$

$$L = \sum_{b:\{0,1\}} r_E(b) \cdot (j \cdot s_F(b) + j \cdot s_F(\perp)) \cdot L$$

A send and a read action of the same message over the same internal channel communicate:

$$s_B \mid r_B = c_B \quad s_C \mid r_C = c_C \quad s_E \mid r_E = c_E \quad s_F \mid r_F = c_F$$

## Motivation for Internal Choice $j$

The internal choice  $j$  is included in  $K$  and  $L$  because else deadlocks could disappear in the abstract  $\mu\text{CRL}$  specification.

For instance, if the  $j$ 's were omitted from  $K$  and  $L$ , and  $T_{db}$  and  $R_b$  would only consist of their first summand, the protocol specification would work correctly.

# Alternating Bit Protocol - Initial State

The **initial state** of the alternating bit protocol is specified by

$$\tau_I(\partial_H(R_0 \parallel S_0 \parallel K \parallel L))$$

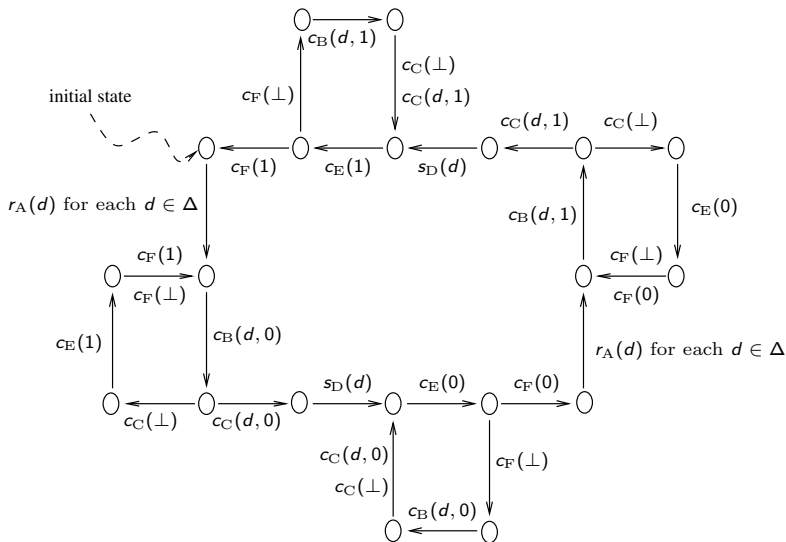
with  $H$  the set of read and send actions over channels B, C, E and F, and  $I$  the set of communication actions together with  $j$ .

$\tau_I(\partial_H(R_0 \parallel S_0 \parallel K \parallel L))$  exhibits the desired **external behavior**

$$X = \sum_{d:\Delta} r_A(d) \cdot s_D(d) \cdot X$$

# Alternating Bit Protocol - State Space

State space of  $\partial_H(R_0 \parallel S_0 \parallel K \parallel L)$  (without  $j$ )



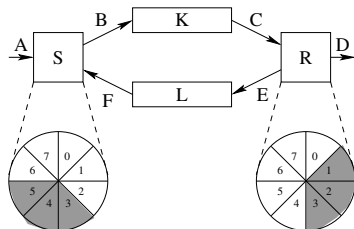
# Alternating Bit Protocol - Unrealistic Assumptions

The alternating bit protocol makes three unrealistic assumptions:

- ▶ Unbounded number of retries
- ▶ Messages are never lost (and error messages can be recognized)
- ▶ Poor use of available bandwidth

The first (and second) assumption will be dropped in the forthcoming bounded retransmission protocol.

# Sliding Window Protocol



The sliding window protocol doesn't include error messages, and has better use of available bandwidth.

A. Tanenbaum, *Computer Networks* (Chapter 4.2), Prentice Hall, 1981

Bahareh Badban, Wan Fokkink, Jan Friso Groote, Jun Pang, Jaco vd Pol  
Verification of a sliding window protocol in  $\mu$ CRL and PVS  
*Formal Aspects of Computing*, 17:342-388, 2005

## Alternating Bit Protocol - $\mu$ CRL specification

In the  $\mu$ CRL specification of the alternating bit protocol, the following data types and action declarations are needed:

- \* *Bool* with constructors T, F and mappings *and*, *or*, *not*, *eq*
- \*  $\Delta$  with constructors  $d_1$ ,  $d_2$  and with mapping *eq*
- \* *Error* with constructor  $\perp$  and with mapping *eq*
- \* *Bit* with constructors 0, 1 and with mappings *invert*, *eq*
- \*  $r_A, s_D : \Delta$   
 $s_B, r_B, c_B, s_C, r_C, c_C : \Delta \times \textit{Bit}$   
 $s_E, r_E, c_E, s_F, r_F, c_F : \textit{Bit}$   
 $s_C, r_C, c_C, s_F, r_F, c_F : \textit{Error}$   
 $j$
- \*  $r_B | s_B = c_B$      $r_C | s_C = c_C$      $r_E | s_E = c_E$      $r_F | s_F = c_F$

# Linear Process Equation

A **linear process equation (LPE)** is a symbolic representation of a state space:

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

with

- ▶  $a_i \in \text{Act} \cup \{\tau\}$
- ▶  $f_i : D \times E \rightarrow D_i$
- ▶  $g_i : D \times E \rightarrow D$
- ▶  $h_i : D \times E \rightarrow \text{Bool}$

A  $\mu\text{CRL}$  specification is turned into an LPE.  
Next a state space can be generated.

# Linearization - Type I and II Equations

Two types of recursive equations  $X(x_1:D_1, \dots, x_n:D_n) = p$  are distinguished:

- I  $p$  contains only  $\cdot, +, \langle b \rangle, \sum_{d:D}$
- II  $p$  also contains  $\parallel, \partial_H, \tau_I, \rho_f$

The  $\mu$ CRL lineariser requires that recursion variables with an equation of **type II** can be eliminated from right-hand sides of recursive equations and from the initial declaration, by repeatedly replacing such variables by the right-hand side of their equation.

**Example:**  $X = a \cdot (X \parallel X)$  can't be linearized.

# Linearization - Type I and II Equations

**Example:** Four one-bit buffers in sequence.

$$\begin{aligned} \mathbf{proc} \quad B_1 &= \sum_{d:D} r_1(d) \cdot s_3(d) \cdot B_1 \\ B_2 &= \sum_{d:D} r_3(d) \cdot s_2(d) \cdot B_2 \\ C &= \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2)) \\ \mathbf{init} \quad \tau_{\{c_4\}}(\partial_{\{s_4, r_4\}}(\rho_{s_2 \rightarrow s_4}(C) \parallel \rho_{r_1 \rightarrow r_4}(C))) \end{aligned}$$

The recursive equations for  $B_1$  and  $B_2$  are of type I.

The recursive equation for  $C$  is of type II.

The occurrences of  $C$  in the initial declaration can be eliminated: replace them by the right-hand side of  $C$ 's recursive equation.

# Linearization

First the type I recursive equations are linearized, in two steps:

- ▶ Turn them into **Greibach Normal Form**, by replacing “non-initial” actions in right-hand sides of recursive equations into fresh recursion variables.
- ▶ Linearize the resulting recursive equations using a **stack**.

(An alternative method uses **pattern matching**.)

Then all (type I and II) equations are transformed into a single LPE, by eliminating parallel, encapsulation, hiding and renaming operators from right-hand sides of recursive equations and the initial declaration.

# Linearization of Type I Equations - Example

First we explain, by an example, the linearization method for type I equations invoked by `mcr1`.

**Example:**  $Y = a \cdot Y \cdot b + c$

$Y$  performs  $k$   $a$ 's, then a  $c$ , and then  $k$   $b$ 's, for any  $k \geq 0$ .

*Step 1:* Make a **Greibach Normal Form**.

$$\begin{aligned} Y &= a \cdot Y \cdot Z + c \\ Z &= b \end{aligned}$$

# Linearization of Type I Equations - Example

*Step 2:* Linearization using a stack.

*Lists* can contain *recursion variables* and their *data parameters* (i.e., function symbols, brackets, comma's).

Empty list  $[]$  and  $in : D \times List \rightarrow List$  are the constructors of *List*.

$empty : List \rightarrow Bool$  and  $head : List \rightarrow D$  and  $tail : List \rightarrow List$  are standard operations on lists.

# Linearization of Type I Equations - Example

$$\begin{aligned} Y &= a \cdot Y \cdot Z + c \\ Z &= b \end{aligned}$$

is transformed into

$$\begin{aligned} X(\lambda:List) &= a \cdot X(in(Y, in(Z, tail(\lambda)))) \triangleleft eq(head(\lambda), Y) \triangleright \delta \\ &+ (c \triangleleft empty(tail(\lambda)) \triangleright c \cdot X(tail(\lambda))) \triangleleft eq(head(\lambda), Y) \triangleright \delta \\ &+ (b \triangleleft empty(tail(\lambda)) \triangleright b \cdot X(tail(\lambda))) \triangleleft eq(head(\lambda), Z) \triangleright \delta \end{aligned}$$

**Question:** How is the initial state  $Y$  represented?

Here  $Y, Z$  don't carry data parameters, so  $D$  contains only  $Y, Z$ .

If recursion variables carry data parameters, then function symbols, brackets and comma's are also pushed on the stack.

# Linearization with Pattern Matching - Example 1

The stack employed in `mcr1` gives a lot of overhead.

`mcr1 -regular` invokes another linearization algorithm for type I equations, based on **pattern matching**.

When the state space is finite, `mcr1 -regular` usually works much more efficiently than `mcr1`.

But `mcr1 -regular` doesn't always terminate.

**Example:**

$$Y = a \cdot Z \cdot Y$$

$$Z = b \cdot Z + b$$

$Y$  repeatedly performs an  $a$  followed by one or more  $b$ 's.

# Linearization with Pattern Matching - Example 1

*Step 1:* Replace  $Z \cdot Y$  by a fresh recursion variable  $X$ .

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= Z \cdot Y \end{aligned}$$

*Step 2:* Expand  $Z$  in the right-hand side of  $X$ . (Store that  $X = Z \cdot Y$ .)

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= b \cdot Z \cdot Y + b \cdot Y \end{aligned}$$

*Step 3:* Replace  $Z \cdot Y$  by  $X$  in the right-hand side of  $X$ .

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= b \cdot X + b \cdot Y \end{aligned}$$

## Linearization with Pattern Matching - Example 2

$$X(n:\text{Nat}) = a(n) \cdot b(S(n)) \cdot c(S(S(n))) \cdot X(S(S(S(n))))$$

$$X(n:\text{Nat}) = a(n) \cdot Y(n)$$

$$Y(n:\text{Nat}) = b(S(n)) \cdot c(S(S(n))) \cdot X(S(S(S(n))))$$

$$X(n:\text{Nat}) = a(n) \cdot Y(n)$$

$$Y(n:\text{Nat}) = b(S(n)) \cdot Z(n)$$

$$Z(n:\text{Nat}) = c(S(S(n))) \cdot X(S(S(S(n))))$$

# Linearization with Pattern Matching - Non-termination

`mcr1 -regular` doesn't always terminate.

Example:

$$\begin{array}{l} Y = a \cdot Y \cdot b + c \\ \hline Y = a \cdot X_1 + c \\ X_1 = Y \cdot b \\ \hline Y = a \cdot X_1 + c \\ X_1 = a \cdot X_1 \cdot b + c \cdot b \\ \hline Y = a \cdot X_1 + c \\ X_1 = a \cdot X_2 + c \cdot Z_1 \\ X_2 = X_1 \cdot b \\ Z_1 = b \\ \hline Y = a \cdot X_1 + c \\ X_1 = a \cdot X_2 + c \cdot Z_1 \\ X_2 = a \cdot X_2 \cdot b + c \cdot Z_1 \cdot b \\ Z_1 = b \\ \hline \vdots \end{array}$$

## Linearization of Type II Equations - Example

We show, by an example, how to reduce the parallel composition of LPEs to an LPE.

**Example:** Let  $a|b = c$ , and

$$X(n:\text{Nat}) = a(n) \cdot X(S(n)) \triangleleft n < 10 \triangleright \delta + b(n) \cdot X(S(S(n))) \triangleleft n > 5 \triangleright \delta$$

$Y(m:\text{Nat}, n:\text{Nat}) = X(m) \parallel X(n)$  can be linearized to:

$$\begin{aligned} Y(m:\text{Nat}, n:\text{Nat}) = & a(m) \cdot Y(S(m), n) \triangleleft m < 10 \triangleright \delta \\ & + b(m) \cdot Y(S(S(m)), n) \triangleleft m > 5 \triangleright \delta \\ & + a(n) \cdot Y(m, S(n)) \triangleleft n < 10 \triangleright \delta \\ & + b(n) \cdot Y(m, S(S(n))) \triangleleft n > 5 \triangleright \delta \\ & + c(m) \cdot Y(S(m), S(S(n))) \triangleleft m < 10 \wedge n > 5 \wedge \text{eq}(m, n) \triangleright \delta \\ & + c(n) \cdot Y(S(S(m)), S(n)) \triangleleft m > 5 \wedge n < 10 \wedge \text{eq}(m, n) \triangleright \delta \end{aligned}$$

# State Space Generation

From an LPE  $X(d:D)$  and initial state  $d_0$ , a state space is generated (with a tool called *instantiator*).

*Closed* contains all “explored” states, and *Open* the generated states that still need to be explored. (For simplicity, transitions are ignored.)

Initially,  $Open = \{d_0\}$  and  $Closed = \emptyset$ .

**while**  $Open \neq \emptyset$  **do**

    select  $d \in Open$ ;  $Open := Open \setminus \{d\}$ ;  $Closed := Closed \cup \{d\}$

    from LPE  $X$ , compute each transition  $X(d) \xrightarrow{a} X(d')$

**if**  $d' \notin Open \cup Closed$  **then**  $Open := Open \cup \{d'\}$

**Challenges:** Store large state spaces in memory.

    Check efficiently whether  $d' \notin Open \cup Closed$ .

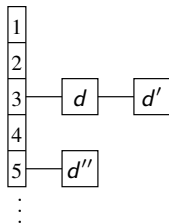
# Hash Tables

A (random) **hash function**  $h$  maps a large domain to a small one, to allow fast lookups:

$$h : D \rightarrow \text{Hash values}$$

**Problem:** Different states may map to the same hash value.

**Solution:** A **chained** hash table.



When the hash table gets full, blocks of states from the hash table are swapped to disk (e.g. based on “age”).

# Bloom Filter

If a generated state  $d'$  isn't in the hash table, the check  $d' \notin Open \cup Closed$  requires an expensive disk lookup.

A **Bloom filter** gives an inexpensive check whether  $d' \in Open \cup Closed$ , allowing for **false positives**.

For some (smartly chosen)  $k$  and  $m$ , fix different hash functions  $h_1, \dots, h_k : D \rightarrow \{1, \dots, m\}$ .

A Bloom filter is a **bit array** of length  $m$ .

Initially, all bits are set to 0.

For each generated state  $d$ , the bits in the Bloom filter at positions  $h_1(d), \dots, h_k(d)$  are set to 1.

# Bloom Filter

If a state  $d'$  is generated, and doesn't occur at entry  $h(d')$  in the hash table, then check if positions  $h_i(d')$  for  $i = 1, \dots, k$  in the Bloom filter all contain 1.

If not, then  $d' \notin Open \cup Closed$ .

Else, still an expensive disk lookup is required.

# Bloom Filter - Analysis

When  $n$  elements have been inserted in  $Open \cup Closed$ , the possibility that a certain position in the Bloom filter contains 0 is

$$\left(\frac{m-1}{m}\right)^{kn}$$

So the probability that  $k$  positions in the Bloom filter all contain 1 is

$$\left(1 - \left(\frac{m-1}{m}\right)^{kn}\right)^k$$

For given  $m, n$ , the number of false positives are minimal for  $k \approx 0.7 \cdot \frac{m}{n}$ .  
(Typically, 256 MB is given to the Bloom filter and  $k = 4$ .)

# Bitstate Hashing

In **bitstate hashing**, a non-chained hash table is maintained.

No extra disk space is used.

If two generated states happen to have the same hash value, the old entry is overwritten by the new entry.

Bitstate hashing approximates an *exhaustive* search for small systems, and slowly changes into a *partial* search for large systems.

## Sorted Lists to Fight State Explosion

Storing messages in non-FIFO channels in a **sorted** list reduces the number of states considerably.

Empty list  $[]$  and  $in : D \times List \rightarrow List$  are the constructors of the data type  $List$ .

Impose a **total order**  $<$  on the data type  $D$ .

$add : D \times List \rightarrow List$  produces *sorted* lists:

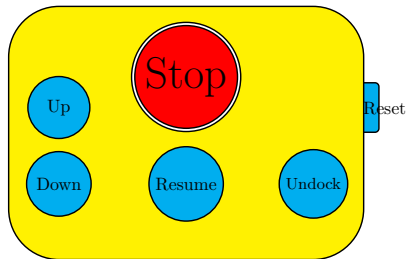
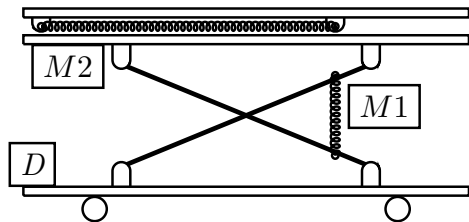
$$add(d, []) = in(d, [])$$

$$add(d, in(e, \ell)) = if(d < e, in(d, in(e, \ell)), in(e, add(d, \ell)))$$

where  $if : Bool \times List \times List \rightarrow List$  acts as “if-then-else”:

$$if(T, \ell_1, \ell_2) = \ell_1 \text{ and } if(F, \ell_1, \ell_2) = \ell_2.$$

# Patient Support System



## System states:

- ▶ calibrated versus uncalibrated
- ▶ docked versus undocked
- ▶ emergency

# Homework Assignment

1. Identify the requirements of the system in natural language.
2. List the external events of the system, which are visible for the outside world. Describe clearly but compactly the meaning of each external event in words.
3. Translate the requirements in terms of these external events.
4. Describe an architecture for the system.
5. Describe the behavior of all components in the architecture in terms of  $\mu$ CRL.
6. Verify using  $\mu$ CRL and CADP that all requirements given in item 3 are valid.
7. If not, modify the  $\mu$ CRL specification (or the requirements), and verify the requirements again.

# Pitfalls for Requirements

Requirements must be formulated in terms of **external events** (input/output).

**Example:** *“the controllers must communicate asynchronously”*

(Implementation detail, can't be verified using model checking.)

Beware not to formulate requirements that are **too general**.

**Example:** *“the bed can move up, down, left or right”*

(In which states of the system, under which inputs?)

# Types of Requirements

- ▶ *Safety*: something bad will never happen.

(E.g., when motor M1 is on, brake B1 is never applied.)

- ▶ *Liveness*: something good will eventually happen.

(E.g., if the system is in uncalibrated mode, the bed isn't in the uppermost position, and the Up button is pressed, then the bed must go up.)

# Minimization Algorithm

Assume a finite state space.

(For simplicity, we disregard termination states.)

Let the set  $S$  of states be partitioned into  $P_1 \cup \dots \cup P_k$ , such that  
(\*) *branching bisimilar states reside in the same set of the partition.*

For  $a \in \text{Act} \cup \{\tau\}$ ,  $s_0 \in \text{split}_a(P_i, P_j)$  if

$s_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$  with  $s_0, \dots, s_{n-1} \in P_i$  and  $s_n \in P_j$ .

If  $s \in \text{split}_a(P_i, P_j)$  and  $s' \in P_i \setminus \text{split}_a(P_i, P_j)$  with  $a \neq \tau$  or  $i \neq j$ ,  
then  $s \not\leftrightarrow_b s'$ .

So after performing a split, (\*) remains satisfied.

# Minimization Algorithm

Initially,  $S$  is partitioned into  $S$ . (So  $(*)$  is trivially satisfied.)

Suppose that at some point  $S$  is partitioned into  $P_1 \cup \dots \cup P_k$ .

If  $a \neq \tau$  or  $i \neq j$ , and

$$\emptyset \subset \text{split}_a(P_i, P_j) \subset P_i$$

then in the partition,  $P_i$  can be replaced by  $\text{split}_a(P_i, P_j)$  and  $P_i \setminus \text{split}_a(P_i, P_j)$ .

Splitting continues until no further split is possible.

$(*)$  is satisfied by the final partition.

# Minimization Algorithm

Let  $s \mathcal{B} s'$  if  $s$  and  $s'$  are in the same set of the final partition.

$\mathcal{B}$  is a branching bisimulation relation.

**Theorem:** Let  $P_1 \cup \dots \cup P_k$  be the final partition of  $S$ .

Two states  $s$  and  $s'$  are in the same set  $P_i$  if and only if  $s \leftrightarrow_b s'$ .

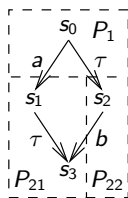
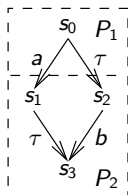
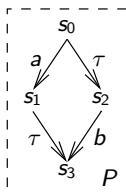
The states in each set  $P_i$  can be collapsed to a single state.

$\tau$ -transitions within a set  $P_i$  can be eliminated.

$P_i \xrightarrow{a} P_j$  if  $s \xrightarrow{a} s'$  for some  $s \in P_i$  and  $s' \in P_j$ , with  $a \neq \tau$  or  $i \neq j$ .

# Minimization Algorithm - Example

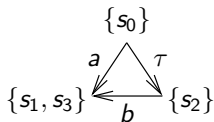
Consider  $(a \cdot \tau + \tau \cdot b) \cdot \delta$ .  $P$  contains all four states in the state space.



$split_a(P, P)$  separates  $s_0$  from  $\{s_1, s_2, s_3\}$ .

$split_b(P_2, P_2)$  separates  $s_2$  from  $\{s_1, s_3\}$ .

$P_1$ ,  $P_{21}$  and  $P_{22}$  can't be split further. The minimized state space is



# Minimization Algorithm - Complexity

**Worst-case time complexity:**  $O(mn)$ , where  $m$  is the number of transitions and  $n$  the number of states in the original state space.

Calculating a split takes  $O(m)$ , and there are no more than  $n$  splits.

# Model Checking

We define some basic **modal** logic operators to express properties of states.

$$\phi ::= \mathbf{T} \mid \mathbf{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a] \phi$$

where  $a$  ranges over  $\text{Act} \cup \{\tau\}$ .

- ▶  $\mathbf{T}$  holds in all states, and  $\mathbf{F}$  in no state
- ▶  $\wedge$  denotes conjunction, and  $\vee$  disjunction
- ▶  $\langle a \rangle \phi$  holds in state  $s$  if there is a transition  $s \xrightarrow{a} s'$  such that  $\phi$  holds in state  $s'$
- ▶  $[a] \phi$  holds in state  $s$  if for each transition  $s \xrightarrow{a} s'$ ,  $\phi$  holds in state  $s'$

# Model Checking

The states  $s$  that satisfy a formula  $\phi$ , denoted  $s \models \phi$ , are defined inductively by:

$$s \models \top$$

$$s \not\models \text{F}$$

$$s \models \phi \wedge \phi' \quad \text{if } s \models \phi \text{ and } s \models \phi'$$

$$s \models \phi \vee \phi' \quad \text{if } s \models \phi \text{ or } s \models \phi'$$

$$s \models \langle a \rangle \phi \quad \text{if for some state } s', s \xrightarrow{a} s' \text{ with } s' \models \phi$$

$$s \models [a] \phi \quad \text{if for all states } s', s \xrightarrow{a} s' \text{ implies } s' \models \phi$$

**Example:** If  $s \xrightarrow{a}$ , then  $s \models [a] \text{F}$  and  $s \not\models \langle a \rangle \top$ .

# Fixpoints

Let  $D$  be a *finite* set with **partial ordering**  $\leq$ ,  
with a *least* and a *greatest* element.

$S : D \rightarrow D$  is **monotonic** if  $d \leq e$  implies  $S(d) \leq S(e)$ .

$d \in D$  is a **fixpoint** of  $S : D \rightarrow D$  if  $S(d) = d$ .

If  $S$  is monotonic, then it has a **minimal fixpoint**  $\mu X.S(X)$   
and a **maximal fixpoint**  $\nu X.S(X)$ .

**Question:** How can  $\mu X.S(X)$  and  $\nu X.S(X)$  be computed?

The  $\mu$ -calculus is a temporal logic.

$$\phi ::= \text{T} \mid \text{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a] \phi \mid X \mid \mu X. \phi \mid \nu X. \phi$$

where the  $X$  are recursion variables.

We restrict to closed formulas, meaning that each occurrence of a recursion variable  $X$  is within the scope of a  $\mu X$  or  $\nu X$ .

We need to explain how a formula  $\phi$  with  $X$  as only free variable is interpreted as a (monotonic) mapping between sets of states.

Consider a finite state space.

Let  $X$  be the *only free variable* in  $\mu$ -calculus formula  $\phi$ .

$\phi$  maps each set  $P$  of states to the set of states that satisfy  $\phi$ , under the assumption that  $P$  is the set of states in which  $X$  holds.

**Example:** Consider the state space  $s_0 \xrightarrow{a} s_1$ .

$\langle a \rangle X$  maps sets containing  $s_1$  to  $\{s_0\}$ , and all other sets to  $\emptyset$ .

$[a] X$  maps sets containing  $s_1$  to  $\{s_0, s_1\}$ , and all other sets to  $\{s_1\}$ .

As partial order we take **set inclusion**.

**Theorem:** For each  $\phi$  with one free variable  $X$ , the corresponding mapping is **monotonic**.

So the closed formulas  $\mu X.\phi$  and  $\nu X.\phi$  are well-defined.

They are satisfied by the states in the minimal and maximal fixpoint of  $\phi$ , respectively.

# $\mu$ -calculus - Negation Violates Monotonicity

Absence of negation in the  $\mu$ -calculus is needed for monotonicity.

Example:



$\mu X. \neg \langle a \rangle X$  has no fixpoint.

## $\mu$ -calculus - Examples

$\mu X.(\langle a \rangle X \vee \langle b \rangle T)$  represents those states that can execute  $a^k b$  for some  $k \geq 0$ .

$\nu X.(\langle a \rangle X \vee \langle b \rangle T)$  represents those states that can execute  $a^\infty$  or  $a^k b$  for some  $k \geq 0$ .

$\nu X.(\langle a \rangle X \vee \langle b \rangle X)$  represents those states that can execute an infinite trace of  $a$ 's and  $b$ 's.

Question: How about  $\mu X.(\langle a \rangle X \vee \langle b \rangle X)$ ?

# $\mu$ -calculus - Complexity

**Worst-case time complexity:**  $O(|\phi| \cdot m \cdot n^{N(\phi)})$

where the state space contains  $n$  states and  $m$  transitions, and  $N(\phi)$  is the longest chain of nested fixpoints in  $\phi$ .

**Example:** Consider  $\nu X. \langle b \rangle (\mu Y. (\langle a \rangle X \vee \langle a \rangle Y))$ .



Y	X
$\emptyset$	$\{s\}$
$\{s\}$	$\emptyset$
$\emptyset$	$\emptyset$

In the second iteration, recomputation of  $Y$  *must* start at  $\emptyset$  (instead of  $\{s\}$ ).

**Conclusion:** If a minimal fixpoint  $\mu Y$  is within the scope of a maximal fixpoint  $\nu X$ , the successive values of  $Y$  must be recomputed starting at  $\emptyset$  every time.

# Alternation-free $\mu$ -calculus

For two nested minimal (or maximal) fixpoints, recomputing a fixpoint isn't so expensive.

**Example:**  $s_0 \xrightarrow{a} s_1 \xrightarrow{b} \dots \xrightarrow{a} s_{2n-3} \xrightarrow{b} s_{2n-2} \xrightarrow{a} s_{2n-1} \xrightarrow{b} s_{2n}$

Consider  $\nu X. \nu Y. (\langle a \rangle X \vee \langle b \rangle Y)$ .

Y	X
$\{s_0, \dots, s_{2n}\}$	$\{s_0, \dots, s_{2n}\}$
$\{s_0, \dots, s_{2n-1}\}$	$\{s_0, \dots, s_{2n-2}\}$
$\{s_0, \dots, s_{2n-3}\}$	$\{s_0, \dots, s_{2n-4}\}$
$\vdots$	$\vdots$
$\emptyset$	$\emptyset$

Note that the successive values of  $X$  and  $Y$  decrease.

This is always true for two nested maximal fixpoints. Likewise, for two nested minimal fixpoints, the successive values always increase.

# Alternation-free $\mu$ -calculus

Worst-case time complexity:  $O(|\phi| \cdot m \cdot n^{A(\phi)})$

where  $A(\phi)$  is the longest chain of nested *alternating* fixpoints in  $\phi$  (i.e., minimal within maximal, or maximal within minimal fixpoint).

Worst-case time complexity:  $O(|\phi| \cdot m \cdot n)$

for model checking the **alternation-free**  $\mu$ -calculus.

Model checking the *full*  $\mu$ -calculus is in **NP**  $\cap$  **co-NP**.

It is an open question whether it is in **P**.

# Regular $\mu$ -calculus

$\alpha ::= \top \mid a \mid \neg\alpha \mid \alpha \wedge \alpha' \quad (a \in \text{Act} \cup \{\tau\})$

$\beta ::= \alpha \mid \beta \cdot \beta' \mid \beta \mid \beta' \mid \beta^*$

$\phi ::= \top \mid \text{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle \beta \rangle \phi \mid [\beta] \phi \mid X \mid \mu X. \phi \mid \nu X. \phi$

$\alpha$  represents a *set of actions*:  $\top$  denotes all actions,  $a$  the set  $\{a\}$ ,  $\neg$  complement, and  $\wedge$  intersection.

$\beta$  represents a *set of traces*:  $\cdot$  is concatenation,  $\mid$  union, and  $*$  iteration.

# Regular $\mu$ -calculus - Examples

Deadlock freeness:  $[T^*] \langle T \rangle T$

Absence of *error*:  $[T^* \cdot \text{error}] F$

After an occurrence of *send*, **fair** reachability of *read* is guaranteed:

$$[T^* \cdot \text{send} \cdot (\neg \text{read})^*] \langle T^* \cdot \text{read} \rangle T$$

**Question:** Specify the properties:

- there is an execution sequence to a deadlock state
- *read* can't be executed before an occurrence of *send*

## Regular $\mu$ -calculus - Examples

There is an infinite execution sequence:

$$\nu X.(\langle T \rangle X)$$

No reachable state exhibits an infinite  $\tau$ -sequence:

$$[T^*] \mu X. [\tau] X$$

Each *send* is eventually followed by a *read*:

$$[T^* \cdot \text{send}] \mu X. (\langle T \rangle T \wedge [\neg \text{read}] X)$$

# CADP Model Checker

CADP supports model checking of **alternation-free**, **regular**  $\mu$ -calculus formulas.

Classes of actions can be expressed with the use of **UNIX regular expressions**, e.g. `'send(.*)'`.

If a property is violated, an *error trace* is produced.

To analyse the error trace, omit the hiding operator from the initial state before state space generation.

# CADP Syntax for Formulas

Examples: `[(not "send(d)")*. "recv(d)"] false`

`[(not 'send(.*)')*. 'recv(.*)'] false`

`[(true)*. "send(d)"] mu X. (<true> true and [not "read(d)"] X)`

Beware that text between quotes ("`...`") is interpreted literally (even "`a(d,e)`" and "`a(d, e)`" are taken to be syntactically different!).

A small typo in an action name may therefore mean you verify a property for a non-existent action.

# Self-loops to Verify State Properties

To take values of variables into account in a model checking analysis, include artificial **self-loops** carrying these variables as action variables.

**Example:** To the recursive equation of a recursion variable  $X(x_1:D_1, x_2:D_2, x_3:D_3)$  one can add a summand

$$+ \text{test}(x_1, x_3) \cdot X(x_1, x_2, x_3)$$

where *test* is a “fresh” action name.

Such a self-loop doesn't increase the number of reachable states (but can influence the validity of properties that ignore fairness).

# Bounded Retransmission Protocol

Data **packets** are sent from RC to TV.

The **last** datum of a packet is labeled.

A datum may get **lost**.

$T_1$  and  $T_2$  send **time-out** messages.

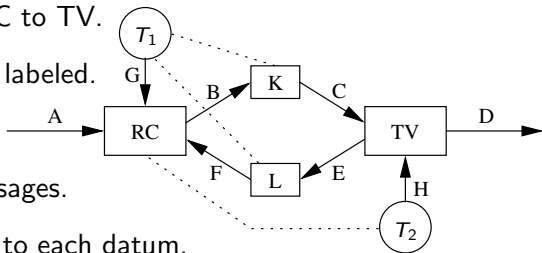
An alternating bit is attached to each datum.

Only **one** kind of acknowledgement.

If RC doesn't receive an acknowledgement within a certain **time**, it resends the datum.

A datum is resent a **limited** number of times.

If TV doesn't receive a next datum within a certain **time**, RC has given up transmission.



# Bounded Retransmission Protocol - External Behavior



## Messages into channel A:

$s_A(I_{OK})$ : transmission was successful

$s_A(I_{NOK})$ : transmission was unsuccessful

$s_A(I_{DK})$ : transmission *may* have been (un)successful

# Bounded Retransmission Protocol - Remote Control

Let  $\Lambda$  consist of lists over  $\Delta$ . (Only lists of length  $\geq 2$  can be transmitted.)

$$X = \sum_{\lambda:\Lambda} r_A(\lambda) \cdot Y(\lambda, 0, S(0)) \triangleleft \text{length}(\lambda) \triangleright S(0) \triangleright \delta$$

$$Y(\lambda:\Lambda, b:\text{Bit}, n:\text{Nat}) =$$

$$(s_B(\text{head}(\lambda), b) \triangleleft \text{length}(\lambda) \triangleright S(0) \triangleright s_B(\text{head}(\lambda), b, \text{last})) \cdot Z(\lambda, b, n)$$

$$Z(\lambda:\Lambda, b:\text{Bit}, n:\text{Nat}) =$$

$$r_F(\text{ack}) \cdot (Y(\text{tail}(\lambda), 1-b, S(0)) \triangleleft \text{length}(\lambda) \triangleright S(0) \triangleright s_A(I_{OK}) \cdot X)$$

$$+ r_G(\text{to}) \cdot (Y(\lambda, b, S(n)) \triangleleft n < \text{max} \triangleright$$

$$(s_A(I_{NOK}) \triangleleft \text{length}(\lambda) \triangleright S(0) \triangleright s_A(I_{DK})) \cdot s_H(\text{to}) \cdot X)$$

## Bounded Retransmission Protocol - Television

$$\begin{aligned}V &= \sum_{d:\Delta} r_C(d, 0) \cdot s_D(d, \text{first}) \cdot s_E(\text{ack}) \cdot W(1) \\ &+ \sum_{d:\Delta} \sum_{b:\text{Bit}} r_C(d, b, \text{last}) \cdot s_E(\text{ack}) \cdot V \\ &+ r_H(\text{to}) \cdot V\end{aligned}$$

$$\begin{aligned}W(b:\text{Bit}) &= \sum_{d:\Delta} r_C(d, b) \cdot s_D(d) \cdot s_E(\text{ack}) \cdot W(1-b) \\ &+ \sum_{d:\Delta} r_C(d, b, \text{last}) \cdot s_D(d, \text{last}) \cdot s_E(\text{ack}) \cdot V \\ &+ \sum_{d:\Delta} r_C(d, 1-b) \cdot s_E(\text{ack}) \cdot W(b) \\ &+ r_H(\text{to}) \cdot V\end{aligned}$$

## Bounded Retransmission Protocol - Medium

$$K = \sum_{d:\Delta} \sum_{b:\{0,1\}} (r_B(d, b) \cdot (j \cdot s_C(d, b) + j \cdot s_G(to))) \cdot K \\ + r_B(d, b, last) \cdot (j \cdot s_C(d, b, last) + j \cdot s_G(to)) \cdot K$$

$$L = r_E(ack) \cdot (j \cdot s_F(ack) + j \cdot s_G(to)) \cdot L$$

# Bounded Retransmission Protocol - Initial State

The **initial state** is specified by

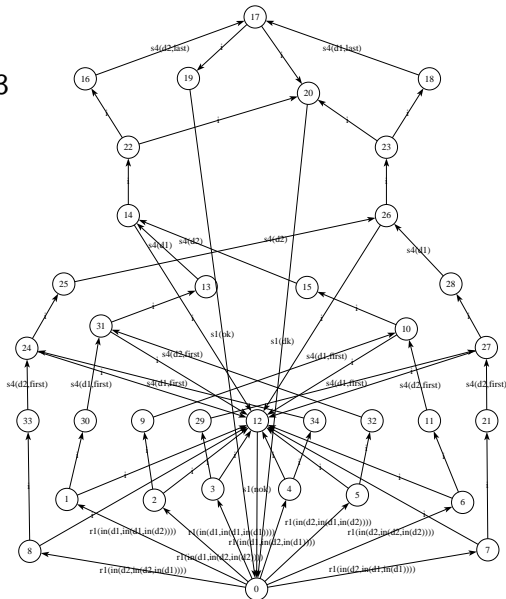
$$\tau_I(\partial_H(V \parallel X \parallel K \parallel L))$$

with  $H$  the internal read and send actions,  
and  $I$  the communication actions and  $j$ .

# Bounded Retransmission Protocol - External Behavior

$$\Delta = \{d_1, d_2\}$$

$\Lambda$  consists of lists of length 3



# Convergent Linear Process Equation

Recall that a **linear process equation (LPE)** is a symbolic representation of a state space:

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

with  $a_i \in \text{Act} \cup \{\tau\}$  and  $f_i : D \times E \rightarrow D_i$  and  $g_i : D \times E \rightarrow D$  and  $h_i : D \times E \rightarrow \text{Bool}$ .

An LPE is **convergent** if it doesn't give rise to infinite  $\tau$ -sequences.

**Example:**  $X = a \cdot X \triangleleft T \triangleright \delta$  is convergent.

$X = \tau \cdot X \triangleleft T \triangleright \delta$  isn't convergent.

The axiom **CL-RSP** says: Let  $P(d)$  for  $d \in D$  be process terms, with

$$P(d) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot P(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

for all  $d \in D$ , where the LPE

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

is *convergent*. Then  $P(d) = X(d)$  for all  $d \in D$ .

**Theorem:** CL-RSP is **sound** modulo  $\leftrightarrow_{rb}$ .

Convergence is essential for the soundness of CL-RSP.

**Example:** Consider  $X = \tau \cdot X$ .

$\tau \cdot a = \tau \cdot \tau \cdot a$  and  $\tau \cdot b = \tau \cdot \tau \cdot b$ .

However,  $\tau \cdot a \not\rightarrow_{rb} \tau \cdot b$ .

Example:  $X(m:\text{Nat}) = a(m + m) \cdot X(S(m))$

$$Y(n:\text{Nat}) = a(n) \cdot Y(S(S(n)))$$

Substituting  $Y(m + m)$  for  $X(m)$  in the first LPE, for  $m:\text{Nat}$ , yields

$$Y(m + m) = a(m + m) \cdot Y(S(m) + S(m))$$

which follows from the second LPE by substituting  $m + m$  for  $n$ , and deriving  $S(S(m + m)) = S(m) + S(m)$ .

So by CL-RSP,  $Y(m + m) = X(m)$  for all  $m \in \text{Nat}$ .

# Invariants

Assume an LPE

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

A mapping  $\mathcal{I} : D \rightarrow \text{Bool}$  is an **invariant** for this LPE if for all  $i \in I$ ,  $d \in D$  and  $e \in E$

$$\mathcal{I}(d) \wedge h_i(d, e) \Rightarrow \mathcal{I}(g_i(d, e))$$

An invariant is used to determine the **reachable** states of the LPE in question (given an initial state  $d_0$  with  $\mathcal{I}(d_0)$ ).

Namely, if  $\mathcal{I}(d)$ , then from  $X(d)$  one can only reach states  $X(d')$  with  $\mathcal{I}(d')$ .

## Invariants - Example

Let  $X(n:\text{Nat}) = a(n) \cdot X(S(S(n)))$ .

Invariants for this LPE are

$$\mathcal{I}(n) = \begin{cases} \text{T} & \text{if } n \text{ is even} \\ \text{F} & \text{if } n \text{ is odd} \end{cases}$$

and

$$\mathcal{I}(n) = \begin{cases} \text{F} & \text{if } n \text{ is even} \\ \text{T} & \text{if } n \text{ is odd} \end{cases}$$

# CL-RSP with Invariants

Let  $P(d)$  for  $d \in D$  be process terms, with

$$P(d) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot P(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

for all  $d \in D$  with  $\mathcal{I}(d)$ , with  $\mathcal{I}$  an invariant for the convergent LPE

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

Then  $P(d) = X(d)$  for all  $d \in D$  with  $\mathcal{I}(d)$ .

## CL-RSP with Invariants - Example

$even : Nat \rightarrow Bool$  is defined by  $even(0) = T$ ,  $even(S(n)) = \neg even(n)$ .

$$X(n: Nat) = a(even(n)) \cdot X(S(S(n)))$$

$$Y = a(T) \cdot Y$$

Substituting  $Y$  for  $X(n)$  for even numbers  $n$  in the first LPE yields

$$Y = a(T) \cdot Y$$

which follows from the second LPE. Since

$$\mathcal{I}(n) = \begin{cases} T & \text{if } n \text{ is even} \\ F & \text{if } n \text{ is odd} \end{cases}$$

is an invariant for the first LPE, by CL-RSP

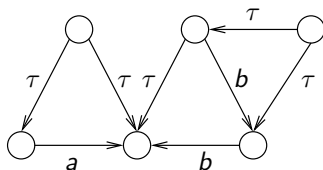
$$X(n) = Y \quad \text{for even numbers } n$$

# Cones and Foci

A state in a state space is a **focus point** if it doesn't have any outgoing  $\tau$ -transitions.

A state is in the **cone** of a focus point if it can reach this focus point by  $\tau$ -transitions only.

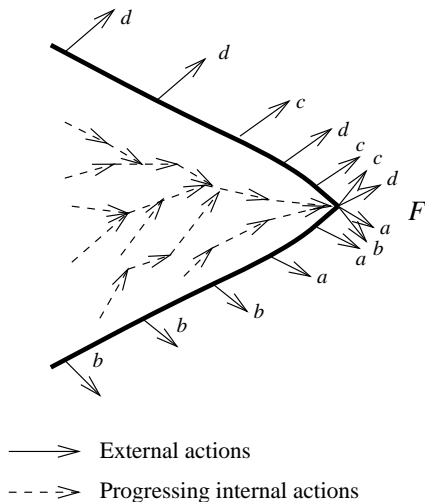
Question:



What are the focus points and the cones in this state space?

# Cones and Foci

We require that each state belongs to the cone of a focus point.



# Matching Criteria

Assume a state space  $G_1$  in which each state belongs to the cone of a focus point, and a state space  $G_2$  without  $\tau$ 's.

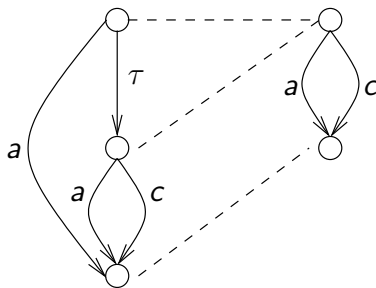
A mapping  $\phi$  from the states in  $G_1$  to the states in  $G_2$  satisfies the matching criteria if for states  $s$  and  $s'$  in  $G_1$  and  $a \neq \tau$ :

- $s \xrightarrow{\tau} s'$  implies  $\phi(s) = \phi(s')$
- if  $s \xrightarrow{a(\vec{d})} s'$ , then  $\phi(s) \xrightarrow{a(\vec{d})} \phi(s')$
- if  $s$  is a focus point of  $G_1$  and  $\phi(s) \xrightarrow{a(\vec{d})} s''$ , then  $s \xrightarrow{a(\vec{d})} s'$  with  $\phi(s') = s''$

If  $\phi$  satisfies the matching criteria, then it establishes a branching bisimulation relation.

# Cones and Foci - Example

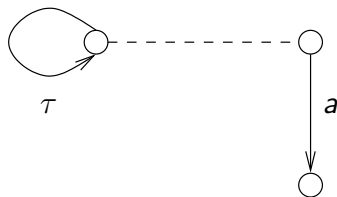
$G_1$     $\phi$     $G_2$



# Cones and Foci

It is essential that each state belongs to the cone of a focus point.

Example:



# Cones and Foci

Assume an LPE

$$X(d:D) = \sum_{a:\text{Act} \cup \{\tau\}} \sum_{e:E} a(f_a(d, e)) \cdot X(g_a(d, e)) \triangleleft h_a(d, e) \triangleright \delta$$

Assume an LPE  $Y$  **without  $\tau$ 's**:

$$Y(d':D') = \sum_{b:\text{Act}} \sum_{e:E} b(f'_b(d', e)) \cdot Y(g'_b(d', e)) \triangleleft h'_b(d', e) \triangleright \delta$$

$\phi : D \rightarrow D'$  satisfies the **matching criteria** at some  $d \in D$

if for all  $b \in \text{Act}$  and  $e \in E$ :

- $h_\tau(d, e) \Rightarrow \phi(d) = \phi(g_\tau(d, e))$
- $h_b(d, e) \Rightarrow h'_b(\phi(d), e)$
- $h_b(d, e) \Rightarrow \phi(g_b(d, e)) = g'_b(\phi(d), e)$
- $h_b(d, e) \Rightarrow f_b(d, e) = f'_b(\phi(d), e)$
- $(FC_X(d) \wedge h'_b(\phi(d), e)) \Rightarrow h_b(d, e)$

where the **focus condition**  $FC_X(d)$  denotes  $\forall e:E \neg h_\tau(d, e)$ .

## Cones and Foci + Invariants - Correctness

Let each datum in  $D$  belong to the cone of a focus point of the LPE  $X(d:D)$ , and let  $Y(d':D')$  be an LPE without  $\tau$ 's.

Let  $\mathcal{I} : D \rightarrow Bool$  be an **invariant** for  $X$ .

Suppose  $\phi : D \rightarrow D'$  satisfies the **matching criteria** at all  $d \in D$  with  $\mathcal{I}(d)$ .

Then for all  $d \in D$  with  $\mathcal{I}(d)$ ,

$$X(d) \xleftrightarrow{b} Y(\phi(d))$$

If moreover  $FC_X(d)$ , then

$$X(d) \xleftrightarrow{rb} Y(\phi(d))$$

## Cones and Foci - Example

$State = \{1, 2, 3\}$

$$\begin{aligned}X(k:State) &= \tau \cdot X(2) \triangleleft k = 1 \triangleright \delta \\ &+ a \cdot X(3) \triangleleft k = 1 \vee k = 2 \triangleright \delta \\ &+ c \cdot X(3) \triangleleft k = 2 \triangleright \delta\end{aligned}$$

$$\begin{aligned}Y(b:Bool) &= a \cdot Y(F) \triangleleft b \triangleright \delta \\ &+ c \cdot Y(F) \triangleleft b \triangleright \delta\end{aligned}$$

**Questions:** What is the **focus condition** ?

How is the **state mapping** defined ?

What are the **matching criteria** ?

Which **equivalences between states** have been proved ?

# Cones and Foci - Example

Focus condition  $FC_X(k)$ :  $k \neq 1$

State mapping:

$$\phi(1) = T$$
$$\phi(2) = T$$
$$\phi(3) = F$$

Matching criteria:

$$k = 1 \Rightarrow \phi(k) = \phi(2)$$
$$k = 1 \vee k = 2 \Rightarrow \phi(k)$$
$$k = 2 \Rightarrow \phi(k)$$
$$k = 1 \vee k = 2 \Rightarrow \neg\phi(3)$$
$$k = 2 \Rightarrow \neg\phi(3)$$
$$k \neq 1 \wedge \phi(k) \Rightarrow k = 1 \vee k = 2$$
$$k \neq 1 \wedge \phi(k) \Rightarrow k = 2$$

IEEE standard 1394, called “FireWire”, is a high performance serial multimedia bus. It connects digital equipment, and is “hot-pluggable”: devices can be added and removed dynamically.

IEEE Computer Society

*IEEE standard for a high performance serial bus*

Std 1394–1995, August 1996

For the sake of performance, identities of nodes aren’t communicated, so the network is **anonymous**.

The size of the network is unknown to its nodes.

The network topology must be **connected** and **free of cycles**.

# IEEE 1394 Leader Election Algorithm

We must determine a **root** of the network,  
by establishing parent-child relations.

A node can send a **parent request** to one of its neighbors.

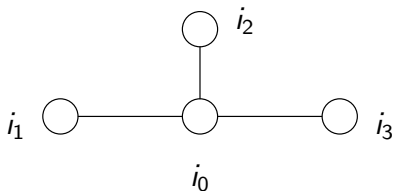
We assume that communication is **synchronous**.

$X(i:Node, p:Nodelist, s:State)$  represents node  $i$   
with possible parents  $p$ , in state  $s$ :

- in state 0 a node is looking for a parent
- in state 1 a node has a parent, or is the root

$$\begin{aligned} X(i:Node, p:Nodelist, s:State) &= \\ &= \sum_{j:Node} r(j, i) \cdot X(i, p \setminus \{j\}, s) \triangleleft j \in p \wedge s = 0 \triangleright \delta \\ &+ \sum_{j:Node} s(i, j) \cdot X(i, p, 1) \triangleleft p = \{j\} \wedge s = 0 \triangleright \delta \\ &+ leader(i) \cdot X(i, p, 1) \triangleleft p = [] \wedge s = 0 \triangleright \delta \end{aligned}$$

# IEEE 1394 Leader Election Algorithm - Example

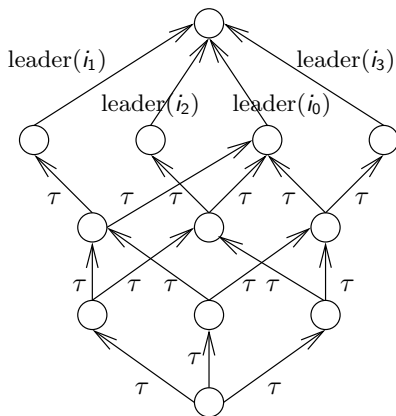


This network is captured by

$$\tau_I(\partial_H(X(i_0, \{i_1, i_2, i_3\}, 0) \parallel X(i_1, \{i_0\}, 0) \parallel X(i_2, \{i_0\}, 0) \parallel X(i_3, \{i_0\}, 0)))$$

# IEEE 1394 Leader Election Algorithm - External Behavior

The **external behavior** of the network on the previous slide:

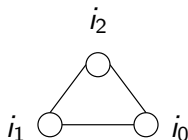


This state space is minimal modulo  $\leftrightarrow_b$ .

# IEEE 1394 Leader Election Algorithm - Correctness

A network with a **cycle** won't produce a root.

Example:



Cycles are detected using a *time-out*.

A **non-connected** network will produce *multiple* roots.

**Theorem:** In a finite network that is **connected** and **free of cycles**, one root will be elected.

# IEEE 1394 Leader Election Algorithm - Verification

ASSUMPTION: the finite network is connected and free of cycles.

GOAL: prove that one root is selected, i.e., the external behavior is  $\tau \cdot \text{leader} \cdot \delta$ .

We specify the node processes without parametrization of *leader*:

$$\begin{aligned} X(i:\text{Node}, p:\text{Nodelist}, s:\text{State}) & \\ &= \sum_{j:\text{Node}} r(j, i) \cdot X(i, p \setminus \{j\}, s) \triangleleft j \in p \wedge s = 0 \triangleright \delta \\ &+ \sum_{j:\text{Node}} s(i, j) \cdot X(i, p, 1) \triangleleft p = \{j\} \wedge s = 0 \triangleright \delta \\ &+ \text{leader} \cdot X(i, p, 1) \triangleleft p = [] \wedge s = 0 \triangleright \delta \end{aligned}$$

The initial state is specified by

$$\tau_1(\partial_H(X(i_0, p_0[i_0], 0) \parallel \dots \parallel X(i_k, p_0[i_k], 0))) \quad (k \geq 1)$$

where  $p_0[i]$  consists of the neighbors of node  $i$ .

# Linearization

$$\begin{aligned} X(i:\text{Node}, p:\text{Nodelist}, s:\text{State}) &= \sum_{j:\text{Node}} r(j, i) \cdot X(i, p \setminus \{j\}, s) \triangleleft j \in p \wedge s = 0 \triangleright \delta \\ &+ \sum_{j:\text{Node}} s(i, j) \cdot X(i, p, 1) \triangleleft p = \{j\} \wedge s = 0 \triangleright \delta \\ &+ \text{leader} \cdot X(i, p, 1) \triangleleft p = [] \wedge s = 0 \triangleright \delta \end{aligned}$$

By **CL-RSP**, one can prove that

$$\tau_I(\partial_H(X(i_0, p[i_0], s[i_0]) \parallel \cdots \parallel X(i_k, p[i_k], s[i_k])))$$

is equal to  $Y(p, s)$ , defined by the LPE

$$\begin{aligned} Y(p:\text{Nodelistlist}, s:\text{Statelist}) &= \sum_{i,j:\text{Node}} \tau \cdot Y(p[i] := p[i] \setminus \{j\}, s[j] := 1) \\ &\quad \triangleleft j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0 \triangleright \delta \\ &+ \sum_{i:\text{Node}} \text{leader} \cdot Y(p, s[i] := 1) \triangleleft \text{empty}(p[i]) \wedge s[i] = 0 \triangleright \delta \end{aligned}$$

$Y(p, s)$  is **convergent**, because each execution of  $\tau$  reduces the number of nodes  $j$  with  $s[j] = 0$ .

# Cones and Foci

**Focus points** of  $Y$  are states  $(p, s)$  that satisfy:

$$\forall i, j: \text{Node} (j \notin p[i] \vee p[j] \neq \{i\} \vee s[i] = 1 \vee s[j] = 1)$$

The LPE for the **external behavior** is

$$Z(b: \text{Bool}) = \text{leader} \cdot Z(F) \triangleleft b \triangleright \delta$$

The **state mapping**  $\phi$  from pairs  $(p, s)$  to  $\text{Bool}$  is defined by

$$\phi(p, s) = \begin{cases} \text{T} & \text{if } s[i] = 0 \text{ for some node } i \\ \text{F} & \text{if } s[i] = 1 \text{ for all nodes } i \end{cases}$$

# Matching Criteria

$Y(p: Nodelistlist, s: Statelist)$

$$= \sum_{i,j: Node} \tau \cdot Y(p[i] := p[i] \setminus \{j\}, s[j] := 1) \triangleleft j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0 \triangleright \delta$$
$$+ \sum_{i: Node} leader \cdot Y(p, s[i] := 1) \triangleleft empty(p[i]) \wedge s[i] = 0 \triangleright \delta$$

## Matching criteria

$$\forall i, j: Node ((j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0) \Rightarrow$$
$$\phi(p, s) = \phi(p[i] := p[i] \setminus \{j\}, s[j] := 1))$$

$$\forall i: Node ((empty(p[i]) \wedge s[i] = 0) \Rightarrow \phi(p, s))$$

$$\forall i: Node ((empty(p[i]) \wedge s[i] = 0) \Rightarrow \neg \phi(p, s[i] := 1))$$

$$(\forall i, j: Node (j \notin p[i] \vee p[j] \neq \{i\} \vee s[i] = 1 \vee s[j] = 1) \wedge \phi(p, s)) \Rightarrow$$
$$\exists i': Node (empty(p[i']) \wedge s[i'] = 0)$$

# Validity of the Matching Criteria

$$\forall i, j: \text{Node} ((j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0) \Rightarrow \\ \phi(p, s) = \phi(p[i] := p[i] \setminus \{j\}, s[j] := 1))$$

$\phi(p, s) = \top = \phi(p[i] \setminus \{j\}, s[j] := 1)$ , because  $s[i]$  remains 0.

$$\forall i: \text{Node} ((\text{empty}(p[i]) \wedge s[i] = 0) \Rightarrow \phi(p, s))$$

$\phi(p, s)$ , because  $s[i] = 0$ .

# Invariants

$Y(p: \text{Nodelistlist}, s: \text{Statelist})$

$$= \sum_{i,j: \text{Node}} \tau \cdot Y(p[i] := p[i] \setminus \{j\}, s[j] := 1) \triangleleft j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0 \triangleright \delta \\ + \sum_{i: \text{Node}} \text{leader} \cdot Y(p, s[i] := 1) \triangleleft \text{empty}(p[i]) \wedge s[i] = 0 \triangleright \delta$$

We list three invariants for this LPE.

$$\mathcal{I}_1(i: \text{Node}, j: \text{Node}) : j \in p[i] \vee i \in p[j]$$

$$\mathcal{I}_2 : \forall i, j: \text{Node} ((j \notin p[i] \wedge i \in p[j]) \Rightarrow s[j] = 1)$$

$$\mathcal{I}_3 : \forall j: \text{Node} (s[j] = 1 \Rightarrow (\text{empty}(p[j]) \vee \text{singleton}(p[j])))$$

$\mathcal{I}_1(i, j)$  holds in the initial state of  $Y$  for all neighbors  $i, j$ .

$\mathcal{I}_2$  and  $\mathcal{I}_3$  hold in the initial state of  $Y$ .

# Uniqueness of the Root

**Lemma:**  $\forall i, j: \text{Node} ((\text{empty}(p[i]) \wedge j \neq i) \Rightarrow (s[j] = 1 \wedge \text{singleton}(p[j])))$

**Proof:** By connectedness, there are distinct nodes

$i = i_0, i_1, \dots, i_m = j$  with  $i_{k+1} \in p_0[i_k]$  for  $k = 0, \dots, m - 1$ .

We prove  $s[i_{k+1}] = 1$  and  $p[i_{k+1}] = \{i_k\}$  for  $k = 0, \dots, m - 1$ .

$$\begin{aligned} k = 0: \quad \mathcal{I}_1 &\Rightarrow (i_1 \in p[i_0] \vee i_0 \in p[i_1]) \\ &\quad \text{empty}(p[i_0]) \Rightarrow (i_1 \notin p[i_0] \wedge i_0 \in p[i_1]) \\ \mathcal{I}_2 &\Rightarrow s[i_1] = 1 \\ \mathcal{I}_3 \wedge i_0 \in p[i_1] &\Rightarrow p[i_1] = \{i_0\} \end{aligned}$$

$$\begin{aligned} k \geq 1: \quad \mathcal{I}_1 &\Rightarrow (i_{k+1} \in p[i_k] \vee i_k \in p[i_{k+1}]) \\ p[i_k] = \{i_{k-1}\} &\Rightarrow (i_{k+1} \notin p[i_k] \wedge i_k \in p[i_{k+1}]) \\ \mathcal{I}_2 &\Rightarrow s[i_{k+1}] = 1 \\ \mathcal{I}_3 \wedge i_k \in p[i_{k+1}] &\Rightarrow p[i_{k+1}] = \{i_k\} \end{aligned}$$

Hence,  $s[i_m] = 1$  and  $p[i_m] = \{i_{m-1}\}$ .

# Validity of the Matching Criteria

$$\forall i: \text{Node} ((\text{empty}(p[i]) \wedge s[i] = 0) \Rightarrow \neg\phi(p, s[i] := 1))$$

By **uniqueness of the root**,  $\text{empty}(p[i])$  implies  $s[j] = 1$  for each  $j \neq i$ .

Hence,  $\neg\phi(p, s[i] := 1)$ .

We need a fourth invariant for LPE  $Y$ .

$$\mathcal{I}_4 : \forall i, j: \text{Node} ((j \in p[i] \wedge s[i] = 0) \Rightarrow (i \in p[j] \wedge s[j] = 0))$$

$\mathcal{I}_4$  holds in the initial state of  $Y$ .

# Validity of the Matching Criteria

$$(\forall i, j: \text{Node} (j \notin p[i] \vee p[j] \neq \{i\} \vee s[i] = 1 \vee s[j] = 1) \wedge \phi(p, s)) \Rightarrow \\ \exists i': \text{Node} (\text{empty}(p[i']) \wedge s[i'] = 0)$$

Since  $\phi(p, s)$ , there is a node  $i'$  with  $s[i'] = 0$ .

Suppose  $p[i']$  isn't empty; we derive a contradiction.

Let  $j \in p[i']$  for some  $j$ . By  $\mathcal{I}_4$ ,  $i' \in p[j]$  and  $s[j] = 0$ .

So by the condition of the matching criterion,  $p[j] \neq \{i'\}$ .

Then there is a  $k \neq i'$  with  $k \in p[j]$ .

By  $\mathcal{I}_4$ ,  $j \in p[k]$  and  $s[k] = 0$ .

So by the condition of the matching criterion,  $p[k] \neq \{j\}$ .

Then there is an  $\ell \neq j$  with  $\ell \in p[k]$ .

Et cetera. This contradicts the fact that there is no cycle.

# Conclusion

If  $p_0$  establishes a connected network without cycles, and  $s_0[i] = 0$  for all nodes  $i$ , then

$$Y(p_0, s_0) \xleftrightarrow{b} Z(\phi(p_0, s_0))$$

Hence,

$$Y(p_0, s_0) \xleftrightarrow{rb} \tau \cdot leader \cdot \delta$$

This implies

$$\begin{aligned} \tau_I(\partial_H(X(i_0, p_0[i_0], 0) \parallel \dots \parallel X(i_k, p_0[i_k], 0))) &\xleftrightarrow{rb} Y(p_0, s_0) \\ &\xleftrightarrow{rb} \tau \cdot leader \cdot \delta \end{aligned}$$

So in a finite network that is **connected** and **free of cycles**, one root will be elected.

# Asynchronous IEEE 1394 Leader Election Algorithm

Now suppose communication is **asynchronous**.

(This can be modeled by unidirectional one-bit buffers between nodes.)

Two nodes can send parent requests to each other simultaneously.

This is called **root contention**.

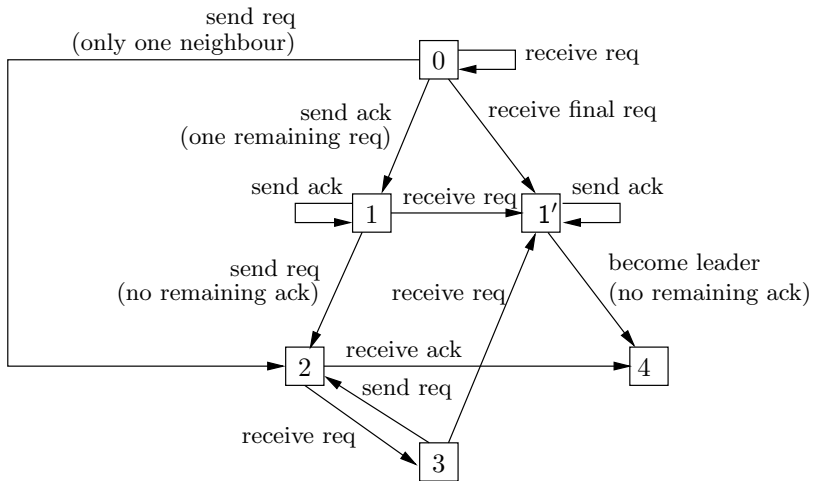
An **acknowledgement** is introduced to confirm that a node accepts a neighbor as child.

# Asynchronous IEEE 1394 Leader Election Algorithm

Each node can be in five states.

- 0: receiving parent requests
- 1: sending acknowledgements, followed by sending a parent request or *leader* action
- 2: waiting for an acknowledgement of a parent request
- 3: root contention (received a parent request instead of an acknowledgement)
- 4: finished

# Asynchronous IEEE 1394 Leader Election Algorithm



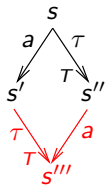
# Confluence

**Confluence** arises when two concurrent components can execute independent actions.

Assume a state space  $G$  with a finite set  $S$  of states.

Fix a set  $T$  of  $\tau$ -transitions in  $G$ . Let  $s \xrightarrow{\tau}_T s'$  denote  $s \xrightarrow{\tau} s' \in T$ .

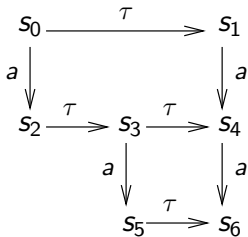
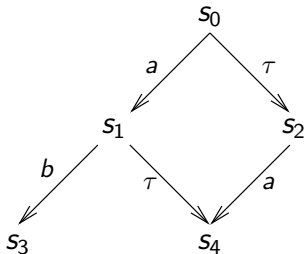
$T$  is **confluent** if for each pair of *distinct* transitions  $s \xrightarrow{a} s'$  ( $a \in \text{Act} \cup \{\tau\}$ ) and  $s \xrightarrow{\tau}_T s''$  in  $G$ ,



**Theorem:** If  $T$  is confluent and  $s \xrightarrow{\tau}_T s'$ , then  $s \xleftrightarrow{b} s'$  in  $G$ .

# Question

Give the maximal set of confluent  $\tau$ -transitions for the following two state spaces.



What are the state spaces after  $\tau$ -priorization? (See next slide.)

# $\tau$ -priorization

Let  $T$  be confluent.

Let  $G$  be free of  $\tau$ -loops.

If  $s \xrightarrow{\tau}_T s'$ , then

- ▶ all other outgoing transitions of  $s$  can be eliminated from  $G$
- ▶ and  $s$  and  $s'$  can be collapsed

without changing the branching bisimulation class of states in  $G$ .

# $\tau$ -prioritization - Absence of $\tau$ -loops is Essential

In case of  $\tau$ -loops, prioritization of confluent  $\tau$ 's may be unsound.

Example:



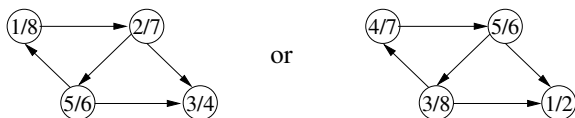
The  $\tau$ -transition is confluent. But eliminating the  $a$ -transition changes the branching bisimulation class of the state.

All states on a  $\tau$ -loop are branching bisimilar, so they can be collapsed to a single state.

$\tau$ -loops can be detected using *Kosaraju's Algorithm* for finding strongly connected components.

# Kosaraju's Algorithm

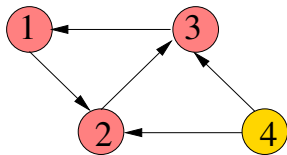
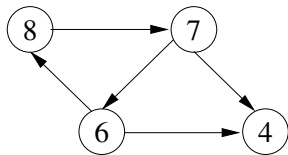
Let **depth-first search** provide “time stamps”  $D_u$  and  $F_u$  when it reaches and deserts a node  $u$ , respectively. For instance,



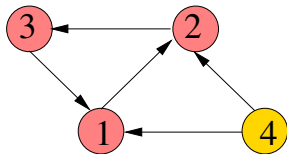
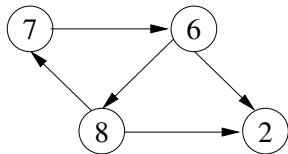
An algorithm for detecting strongly connected components in a directed graph  $G$ :

1. Apply depth-first search to  $G$ ; each new exploration starts in an unvisited node.
2. Reverse all edges in  $G$ , and apply depth-first search to the resulting graph  $G^R$ ; each new exploration starts in the unvisited node with the highest  $F$ -value. Each exploration in  $G^R$  determines a strongly connected component in  $G$ .

# Kosaraju's Algorithm - Example



or



# Kosaraju's Algorithm - Correctness & Complexity

**Correctness:** Let  $u$  have the largest  $F$ -value after DFS1.

If  $v$  isn't discovered from  $u$  in DFS2, there is no path from  $v$  to  $u$  in  $G$ . So then they aren't in the same strongly connected component.

If  $v$  is discovered from  $u$  in DFS2, there is a path from  $v$  to  $u$  in  $G$ .

Suppose, towards a contradiction, there is no path from  $u$  to  $v$  in  $G$ .

- If DFS1 visited  $v$  before  $u$ , then  $F_v > F_u$   
(because there is a path from  $v$  to  $u$ ).
- If DFS1 visited  $u$  before  $v$ , then  $F_v > F_u$   
(because there is no path from  $u$  to  $v$ ).

This contradicts the fact that  $F_u > F_v$ .

**Time complexity:**  $O(m + n)$ , with  $m/n$  the number of edges/nodes.

# Computation of Maximal Confluent Set

**Step 1:** Collapse all states on a  $\tau$ -loop.

**Step 2:** Initially,  $T$  contains all  $\tau$ -transitions, and all transitions are placed on a stack  $\mathcal{S}$ .

**Step 3:** While  $\mathcal{S} \neq \emptyset$ , take an  $s \xrightarrow{a} s'$  from  $\mathcal{S}$ .

Verify for each  $s \xrightarrow{\tau} s'' \in T$  whether the confluence property holds.

If not, then  $s \xrightarrow{\tau} s''$  is eliminated from  $T$ , and **all transitions  $s''' \xrightarrow{b} s$  are placed back on  $\mathcal{S}$ .**

(Namely, confluence of  $s''' \xrightarrow{b} s$  and some  $s''' \xrightarrow{\tau} s'''' \in T$  may have depended on the fact that  $s \xrightarrow{\tau} s'' \in T$ .)

**Step 4:** If  $\mathcal{S} = \emptyset$ , then output  $T$ .

**Worst-case time complexity:**  $O(m + n)$

# Partial Order Reduction

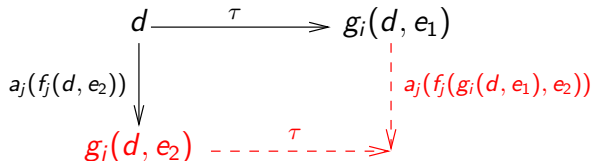
A main obstacle for protocol verification is state explosion. We are after algorithms that generate a reduced, but equivalent, state space.

Assume an LPE

$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

We want to establish which  $\tau$ -summands give rise to only confluent  $\tau$ -transitions.

Let  $a_i = \tau$ . We need to check that  $h_i(d, e_1)$  and  $h_j(d, e_2)$  implies



or  $a_j = \tau$  and  $g_i(d, e_1) = g_j(d, e_2)$ .

# Partial Order Reduction

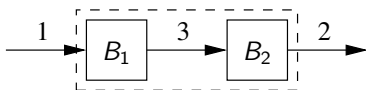
$$X(d:D) = \sum_{i:I} \sum_{e:E} a_i(f_i(d, e)) \cdot X(g_i(d, e)) \triangleleft h_i(d, e) \triangleright \delta$$

Let  $a_i = \tau$ . If  $h_i(d, e_1) \wedge h_j(d, e_2)$  always implies

$$\begin{aligned} & h_j(g_i(d, e_1), e_2) \\ \wedge & h_i(g_j(d, e_2), e_1) \\ \wedge & f_j(d, e_2) = f_j(g_i(d, e_1), e_2) \\ \wedge & g_j(g_i(d, e_1), e_2) = g_i(g_j(d, e_2), e_1) \\ \vee & \\ & a_j = \tau \\ \wedge & g_i(d, e_1) = g_j(d, e_2) \end{aligned}$$

then summand  $i$  of the LPE is **confluent**.

## Two Unbounded Queues



$$\begin{aligned} B_1(\lambda_1:List) &= \sum_{d:\Delta} r_1(d) \cdot B_1(in(d, \lambda_1)) \\ &+ s_3(toe(\lambda_1)) \cdot B_1(untoe(\lambda_1)) \triangleleft \neg empty(\lambda_1) \triangleright \delta \end{aligned}$$

$$\begin{aligned} B_2(\lambda_2:List) &= \sum_{d:\Delta} r_3(d) \cdot B_2(in(d, \lambda_2)) \\ &+ s_2(toe(\lambda_2)) \cdot B_2(untoe(\lambda_2)) \triangleleft \neg empty(\lambda_2) \triangleright \delta \end{aligned}$$

The initial state is  $\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1(\square) \parallel B_2(\square)))$ .

The external behavior of the two unbounded queues in sequence is again an unbounded queue. The resulting state space is infinite.

## Two Unbounded Queues - Data Types

**sort**  $List$

**func**  $[] : \rightarrow List$

$in : \Delta \times List \rightarrow List$

**map**  $empty : List \rightarrow Bool$

$if : Bool \times List \times List \rightarrow List$

$toe : List \rightarrow \Delta$

$untoe : List \rightarrow List$

**var**  $d : \Delta$

$\lambda, \lambda_1, \lambda_2 : List$

**rew**  $empty([]) = T$

$empty(in(d, \lambda)) = F$

$if(T, \lambda_1, \lambda_2) = \lambda_1$

$if(F, \lambda_1, \lambda_2) = \lambda_2$

$toe(in(d, \lambda)) = if(\neg empty(\lambda), toe(\lambda), d)$

$untoe(in(d, \lambda)) = if(\neg empty(\lambda), in(d, untoe(\lambda)), [])$

## Two Unbounded Queues - Confluence Formulas

$\tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1(\lambda_1) \parallel B_2(\lambda_2)))$  linearizes to

$$\begin{aligned} X(\lambda_1:List, \lambda_2:List) &= \sum_{d:\Delta} r_1(d) \cdot X(in(d, \lambda_1), \lambda_2) \\ &+ \tau \cdot X(untoe(\lambda_1), in(toe(\lambda_1), \lambda_2)) \triangleleft \neg empty(\lambda_1) \triangleright \delta \\ &+ s_2(toe(\lambda_2)) \cdot X(\lambda_1, untoe(\lambda_2)) \triangleleft \neg empty(\lambda_2) \triangleright \delta \end{aligned}$$

We compute the confluence formulas.

(1) Commutation of  $\tau$  and  $r_1(d)$ , for any  $d \in D$ :

$$\begin{aligned} \neg empty(\lambda_1) &\Rightarrow \neg empty(in(d, \lambda_1)) \\ &\wedge in(d, untoe(\lambda_1)) = untoe(in(d, \lambda_1)) \\ &\wedge in(toe(\lambda_1), \lambda_2) = in(toe(in(d, \lambda_1)), \lambda_2) \end{aligned}$$

(2) Commutation of  $\tau$  and  $s_2(toe(\lambda_2))$ :

$$\begin{aligned} \neg empty(\lambda_1) \wedge \neg empty(\lambda_2) &\Rightarrow \neg empty(in(toe(\lambda_1), \lambda_2)) \\ &\wedge toe(in(toe(\lambda_1), \lambda_2)) = toe(\lambda_2) \\ &\wedge untoe(in(toe(\lambda_1), \lambda_2)) = in(toe(\lambda_1), untoe(\lambda_2)) \end{aligned}$$

## Two Unbounded Queues - Symbolic $\tau$ -prioritization

If a  $\tau$ -summand in a **convergent** LPE is confluent, then it can be given priority over other summands.

This is done by adding the negation of the condition of the  $\tau$ -summand as a conjunct to the conditions of the other summands.

**Example:** Since the  $\tau$ -summand of the LPE for two unbounded buffers in sequence is confluent, this LPE can be transformed to

$$\begin{aligned} & X(\lambda_1:List, \lambda_2:List) \\ = & \sum_{d:\Delta} r_1(d) \cdot X(in(d, \lambda_1), \lambda_2) \triangleleft \mathbf{empty}(\lambda_1) \triangleright \delta \\ + & \tau \cdot X(untoe(\lambda_1), in(toe(\lambda_1), \lambda_2)) \triangleleft \neg \mathbf{empty}(\lambda_1) \triangleright \delta \\ + & s_2(toe(\lambda_2)) \cdot X(\lambda_1, untoe(\lambda_2)) \triangleleft \mathbf{empty}(\lambda_1) \wedge \neg \mathbf{empty}(\lambda_2) \triangleright \delta \end{aligned}$$

So if  $\lambda_1$  isn't empty, then only the  $\tau$ -summand can be executed.

# Representative States

Consider a *finite* state space, possibly containing  $\tau$ -loops.

For each reachable state  $d$  we compute a **representative** state  $repr(d)$ , such that:

- ▶ if  $d \xrightarrow{\tau} d'$  is confluent, then  $repr(d) = repr(d')$ ; and
- ▶ each state  $d$  can evolve to  $repr(d)$  by confluent  $\tau$ -transitions.

To compute the representative of a state, a depth-first search traversal via the confluent  $\tau$ -transitions is made, until a state with a known representative is encountered, or a 'terminal' strongly connected component of confluent  $\tau$ -transitions.

In the first case the **representative** is returned, and in the second case the **least state** in the terminal strongly connected component (assuming a total order on states).

# State Space Generation Modulo Confluence

Consider an LPE (with a finite state space).

In the state space generation algorithm,  
only representatives of states need to be generated.

State space generation is started from  $\text{repr}(d_0)$ ,  
where  $d_0$  is the initial state.

If the representative of a state  $d$  is added to the state space,  
then for each encountered transition  $d \xrightarrow{a} d'$ , add the state  $\text{repr}(d')$   
and the transition  $\text{repr}(d) \xrightarrow{a} \text{repr}(d')$  to the generated state space  
(except for transitions with  $a = \tau$  and  $\text{repr}(d) = \text{repr}(d')$ ).

# State Space Generation Modulo Confluence - Case Studies

	standard state space		reduced state space	
system	states	transitions	states	transitions
abp(2)	97	122	29	54
brp(2)	1,952	2,387	1,420	1,855
tip(10)	72,020	389,460	6,171	22,668
tip(12)	446,648	2,853,960	27,219	123,888
tip(14)	2,416,632	17,605,592	105,122	544,483

```
confcheck -mark file.tbf
```

renames confluent tau-summands in the input file `file.tbf` into `ctau`.

The flag `-invariant` allows to attach an invariant.

```
confelm file.tbf
```

symbolically reduces `file.tbf`, exploiting `ctau`-summands.

```
instantiator -confluent ctau file.tbf
```

generates the state space from `file.tbf`,  
while prioritizing `ctau`-transitions.

Constructor terms are stored compactly as **ATerms**, consisting of a head symbol and some parameters, which are again ATerms.

ATerms are stored by means of **maximal sharing**:

An ATerm is stored only once, but may be referenced multiple times.

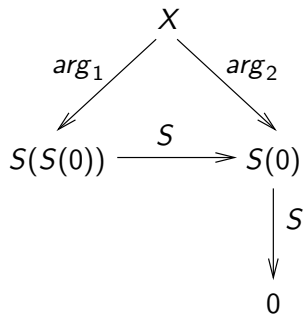
Equality checking reduces to a single comparison of references.

Unreferenced ATerms are reclaimed by a garbage collector.

# ATerms

LPEs are also stored using ATerms.

**Example:** The process term  $X(S(S(0)), S(0))$  is stored as follows.



# Distributed State Space Generation - Hash Function

Store the state space on a cluster of computers (e.g. DAS-4).

Let there be a **globally known hash function**.

States are divided over processors on the basis of their hash values.

When a state is generated at a processor, its hash value is calculated, and the state is forwarded to the appropriate processor.

There it is determined whether the state was generated before.

The outgoing transitions of a state are stored separately at the processor that owns this state.

# Distributed State Space Generation - Complications

In  $\mu$ CRL, a state is represented as a long list of data terms.

Storage as ATerms hampers the computation of hash values.

The long representations of states as a list of data terms makes the computation of hash values and sending states expensive.

We discuss a database approach to distributed state space generation to overcome these two complications.

## Distributed State Space Generation - Database

Data terms occurring in states are provided with an **index** in  $\mathbb{N}$ , maintained in a central database.

A new data term encountered during state space generation gets index  $max + 1$ , with  $max$  the largest index in use.

A state  $(d_1, \dots, d_k)$  becomes a list of indices  $(i_1, \dots, i_k)$ .

Computation of hash values, and sending states, becomes cheaper.

The database usually remains small, compared to the state space.

## Distributed State Space Generation - Database

To compute the successors of a state  $(i_1, \dots, i_k)$ , it needs to be expanded into  $(d_1, \dots, d_k)$ .

The data terms in a successor  $(e_1, \dots, e_k)$  are transformed back to a list of indices  $(j_1, \dots, j_k)$ .

Data terms that aren't yet in the database are added to it.

The hash value of  $(j_1, \dots, j_k)$  is computed, and  $(j_1, \dots, j_k)$  is sent to the responsible processor.

The state space generator must continuously consult the database, to move between the two representations of states.

# Distributed State Space Generation - Database

The central database is replicated at all processors.

Each local database at a processor  $P$  always contains all indices from 0 up to some  $max_P$ .

If indices of some data terms  $e_1, \dots, e_\ell$  are absent in  $P$ 's database, it sends these terms together with  $max_P$  to the central database.

At the central database, data terms that aren't yet present are included with fresh indices.

Data terms with indices from  $max_P + 1$  up to  $max$  are sent to  $P$ .

Thus  $P$  is also provided with data terms it hasn't yet encountered, to avoid future requests and thus reduce communication overhead.

# Distributed State Space Generation - Binary Tree

Lists of indices are stored in a **binary tree**, exploiting the fact that in a transition generally most data terms in the state remain unchanged.

A list of indices is split into two halves, stored in separate tables; each half is associated to an index number.

A root table only contains pairs of indices, pointing to the corresponding halves in those two tables.

This split is applied recursively, producing a binary tree with:

- ▶ Pairs of indices in each **non-leaf**;  
each index points to such a pair in a child.
- ▶ Parts of length 2 from the original lists of indices in each **leaf**.

# Distributed State Space Generation - Binary Tree

A state is added to the binary tree in a bottom-up fashion.

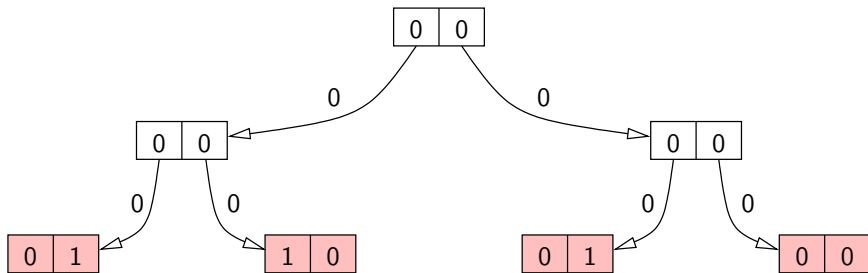
The list of indices is chopped into sublists of length 2; each sublist is added to the corresponding leaf, if not yet present.

At each non-leaf, the pair of indices representing the part of the original list “stored” at this node is added, if not yet present.

The state was already present if the tree remains unchanged.

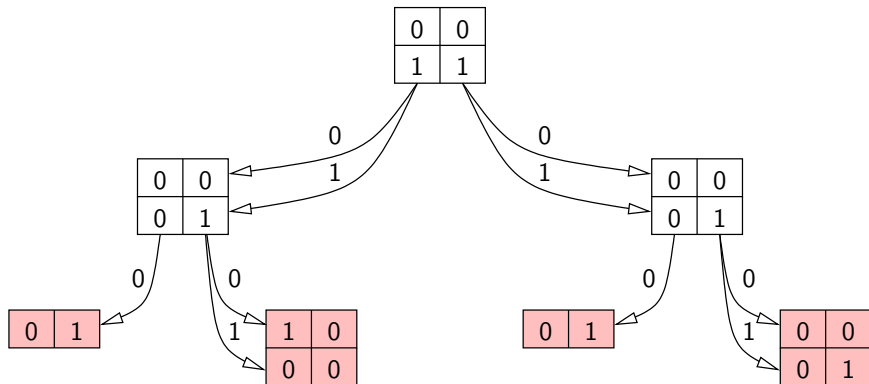
# Distributed State Space Generation - Example

A processor  $P$  first stores the state 01100100.



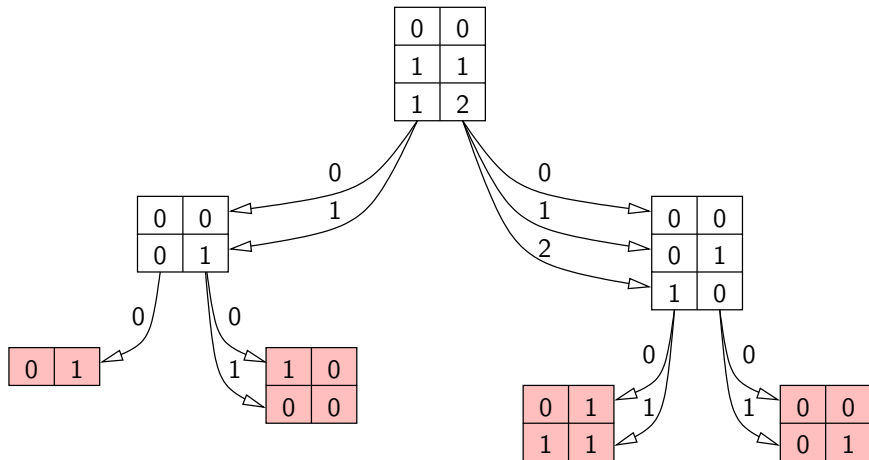
# Distributed State Space Generation - Example

$P$  next stores the state 01000101.



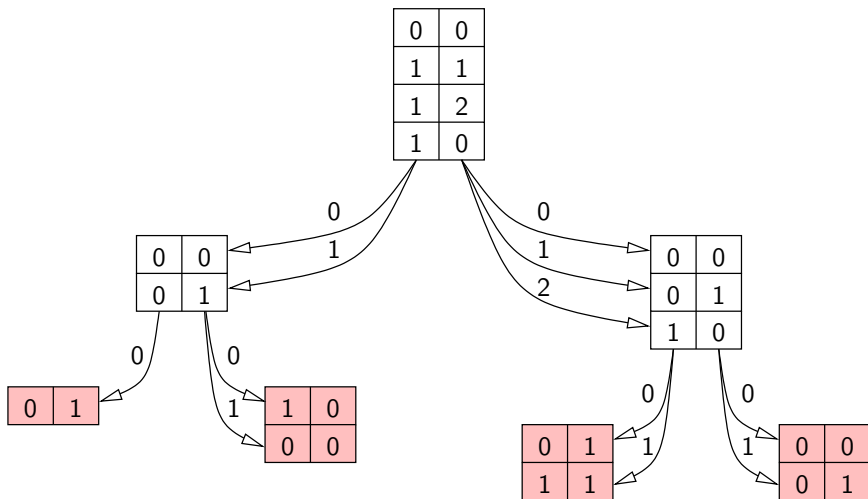
# Distributed State Space Generation - Example

$P$  next stores the state 01001100.



# Distributed State Space Generation - Example

$P$  finally stores the state 01000100.



# Distributed State Space Generation - Binary Tree

We assumed states are represented by lists of indices of length  $2^k$ .

If this isn't the case, some nodes have only one child, being a leaf, and contain pairs of:

- ▶ an index pointing to this leaf, and
- ▶ a single element from an original list of indices.

# Distributed State Space Generation - Moral

Optimizations for a uniprocessor setting  
may become a stumbling block in a distributed setting.

Special data structures and algorithmic solutions  
can come to the rescue.

# Distributed Verification

Distributed versions exist of:

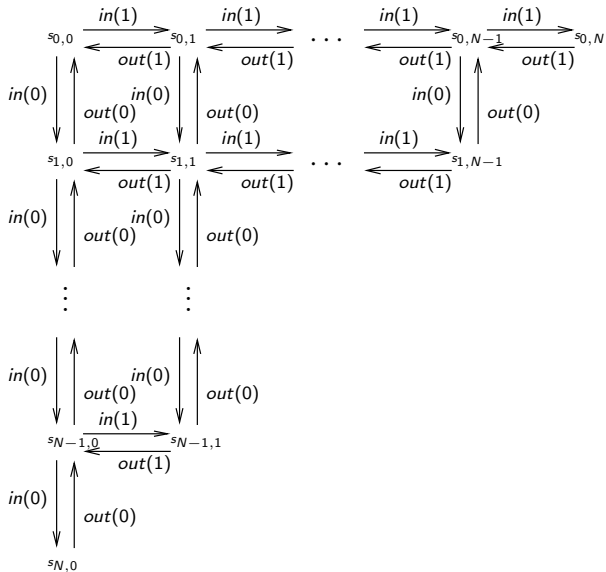
- ▶ minimization modulo  $\leftrightarrow_b$
- ▶ model checking

**Challenge:** Perform these tasks efficiently,  
with as little communication overhead as possible.

# Abstraction

Define mappings  $\pi : S \rightarrow \hat{S}$  and  $\theta : A \rightarrow \hat{A}$ ,  
where  $\hat{S}$  and  $\hat{A}$  contain **abstracted** states and actions, respectively.

# Abstraction - Bag of Size $N$



# Abstraction - Bag of Size $N$

Let  $N \geq 3$ .

Let  $\hat{S} = \{\text{empty}, \text{middle}, \text{full}\}$  and  $\hat{A} = \{\hat{\iota}, \hat{\delta}\}$ .

$$\begin{aligned}\pi(s_{0,0}) &= \text{empty} \\ \pi(s_{i,j}) &= \text{middle} && 0 < i + j < N \\ \pi(s_{i,j}) &= \text{full} && i + j = N \\ \\ \theta(\text{in}(b)) &= \hat{\iota} && b \in \{0, 1\} \\ \theta(\text{out}(b)) &= \hat{\delta} && b \in \{0, 1\}\end{aligned}$$

# Abstraction

## Must transitions

$\hat{s} \xrightarrow{\hat{a}} \square \hat{s}'$  if for all  $s \in S$  with  $\pi(s) = \hat{s}$  there is a transition  $s \xrightarrow{a} s'$  with  $\theta(a) = \hat{a}$  and  $\pi(s') = \hat{s}'$ .

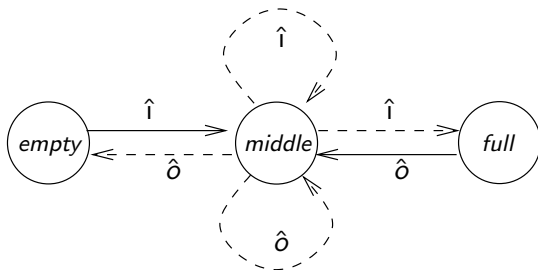
## May transitions

$\hat{s} \xrightarrow{\hat{a}} \diamond \hat{s}'$  if there is a transition  $s \xrightarrow{a} s'$  with  $\pi(s) = \hat{s}$ ,  $\theta(a) = \hat{a}$  and  $\pi(s') = \hat{s}'$ .

Must transitions are depicted as **solid** arrows,  
and may transitions as **dashed** arrows.

# Abstraction - Bag of Size $N$

The abstracted state space for the bag of size  $N \geq 3$  is



# Model Checking with Abstraction

$$C(T) = \hat{S}$$

$$C(F) = \emptyset$$

$$C(\phi \wedge \phi') = C(\phi) \cap C(\phi')$$

$$C(\phi \vee \phi') = C(\phi) \cup C(\phi')$$

$$C(\langle \hat{a} \rangle \phi) = \{\hat{s} \mid \exists \hat{s}' \in C(\phi) (\hat{s} \xrightarrow{\hat{a}}_{\square} \hat{s}')\}$$

$$C([\hat{a}] \phi) = \{\hat{s} \mid \hat{s} \xrightarrow{\hat{a}}_{\diamond} \hat{s}' \Rightarrow \hat{s}' \in C(\phi)\}$$

If  $\pi(s) \in C(\phi)$ , then in the original state space,  $s$  satisfies the formula obtained by replacing expressions  $\langle \hat{a} \rangle$  and  $[\hat{a}]$  in  $\phi$  by  $\langle \alpha \rangle$  and  $[\alpha]$ , where  $\alpha$  represents the union of all actions in  $\theta^{-1}(\hat{a})$ .

# Model Checking with Abstraction

As before,  $\mu X.\phi$  and  $\nu X.\phi$  denote minimal and maximal fixpoints.

And the logic can be extended with  $\langle \hat{\beta} \rangle \phi$  and  $[\hat{\beta}] \phi$ ,  
where  $\hat{\beta}$  is a regular expression representing a set of traces  
over the abstracted actions

Model checking on an abstracted state space works well  
for **safety** properties (something bad will never happen)

The safety property is then checked on an **overapproximation** of the  
original system, as all must and may transitions are taken into account.

The safety property can therefore be lifted to the original state space.

Lifting **liveness** properties (something good will eventually happen)  
from an abstracted to the concrete state space is much more difficult.

# Model Checking with Abstraction - Bag of Size $N$

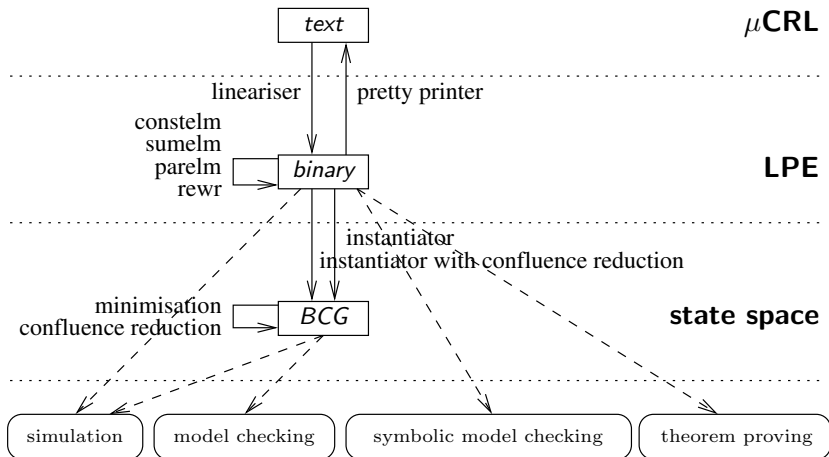
The initial state of the *abstracted* state space of the bag of size  $N$  satisfies

$$[(\neg(\hat{i}))^* \cdot \hat{o}] F$$

So the initial state of the *original* state space satisfies

$$[(\neg(in(0)|in(1)))^* \cdot (out(0)|out(1))] F$$

# Overview of the $\mu$ CRL Toolset



# Automated Theorem Prover

A **theorem prover** within the  $\mu$ CRL toolset provides (semi-)automated support for proving (large) formulas.

This theorem prover isn't complete!  
(Equalities over an abstract data type are in general undecidable.)

If the theorem prover can't prove validity of a formula, diagnostics are provided. The user can add equations to the data specification.

In some cases, the formula isn't valid in all states of the system, but does hold in all reachable states.

The user may supply **invariants**.

Formulas can be proved under the assumption of invariants.

Such invariants must be proved separately, using the theorem prover.