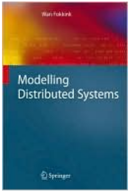
Protocol Validation with μCRL



Wan Fokkink
Modelling Distributed Systems
Springer, 2007

Goals

- Formal modeling of real-life protocols
- Automated analysis of protocols by means of state space exploration (i.e. simulation or model checking)
- To get an impression of the difficulties in analysis (state space explosion)
- Symbolic verification of protocols using equational logic and theorem provers

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)

Natural Numbers - Example

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

The axioms are:

$$\begin{aligned} plus(n, 0) &= n \\ plus(n, S(m)) &= S(plus(n, m)) \\ mul(n, 0) &= 0 \\ mul(n, S(m)) &= plus(mul(n, m), n) \end{aligned}$$

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

Question: Derive $plus(S(0), S(0)) = S(S(0))$.

Constructors

μCRL uses algebraic specification of data, with explicit recognition of constructor symbols, which cannot be eliminated from data terms.

Example: For the natural numbers, 0 and S are constructors, while *plus* and *mul* are not.

$$
\begin{array}{ll}
\textbf{sort} & Nat \\
\textbf{func} & 0 :\to Nat \\
 & S : Nat \to Nat \\
\textbf{map} & plus, mul : Nat \times Nat \to Nat \\
\textbf{var} & n, m : Nat \\
\textbf{rew} & plus(n, 0) \; = \; n \\
 & plus(n, S(m)) \; = \; S(plus(n, m)) \\
 & mul(n, 0) \; = \; 0 \\
 & mul(n, S(m)) \; = \; plus(mul(n, m), n)
\end{array}
$$

Question: Specify $power(n, m)$, denoting n^m.

Booleans

'true' and 'false', together with conjunction, disjunction and negation must be declared in each μCRL specification.

$$
\begin{array}{ll}
\textbf{sort} & \mathit{Bool} \\
\textbf{func} & \mathsf{T}, \mathsf{F} :\to \mathit{Bool} \\
\textbf{map} & \wedge, \vee : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool} \\
 & \neg : \mathit{Bool} \to \mathit{Bool} \\
\textbf{var} & b : \mathit{Bool} \\
\textbf{rew} & b \wedge \mathsf{T} = b \\
 & b \wedge \mathsf{F} = \mathsf{F} \\
 & b \vee \mathsf{T} = \mathsf{T} \\
 & b \vee \mathsf{F} = b \\
 & \neg \mathsf{T} = \mathsf{F} \\
 & \neg \mathsf{F} = \mathsf{T}
\end{array}
$$

Innermost Rewriting

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

A normal form only consists of constructor symbols.

Equality Function

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

Example:

$$
\begin{array}{ll}
\textbf{map} & eq : Bool \times Bool \rightarrow Bool \\
\textbf{rew} & eq(\mathrm{T}, \mathrm{T}) = \mathrm{T} \\
 & eq(\mathrm{F}, \mathrm{F}) = \mathrm{T} \\
 & eq(\mathrm{T}, \mathrm{F}) = \mathrm{F} \\
 & eq(\mathrm{F}, \mathrm{T}) = \mathrm{F}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{map} & eq : Nat \times Nat \rightarrow Bool \\
\textbf{var} & n, m : Nat \\
\textbf{rew} & eq(0, 0) = \mathrm{T} \\
 & eq(S(n), S(m)) = eq(n, m) \\
 & eq(0, S(n)) = \mathrm{F} \\
 & eq(S(n), 0) = \mathrm{F}
\end{array}
$$

Equality Function

A shorter specification of the equality function on booleans is:

$$\begin{array}{rcl} eq(b, b) & = & \mathsf{T} \\ eq(\mathsf{T}, \mathsf{F}) & = & \mathsf{F} \\ eq(\mathsf{F}, \mathsf{T}) & = & \mathsf{F} \end{array}$$

The following specification of an equality function does *not* work in μCRL:

$$\begin{array}{rcl} eq(x, x) & = & \mathsf{T} \\ eq(x, y) & = & \mathsf{F} \end{array}$$

That is, rewrite rules are not always '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 is T] $\neg\neg \mathsf{T} = \neg \mathsf{F} = \mathsf{T}$

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

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

[Base case, n is 0] $plus(0, 0) = 0$

[Inductive case, n is $S(m)$] $plus(0, S(m)) = S(plus(0, m)) = S(m)$

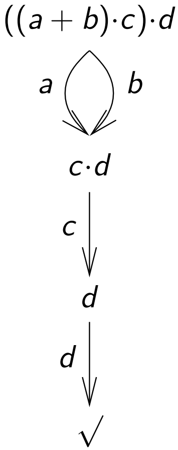
Basic Process Terms

Basic process terms are built from parametrized actions in a set Act, alternative composition and sequential composition.

- An action name $a \in \mathrm{Act}$ represents indivisible behavior.
  It can carry data parameters: $a(d_1, \ldots, d_n)$.
- 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



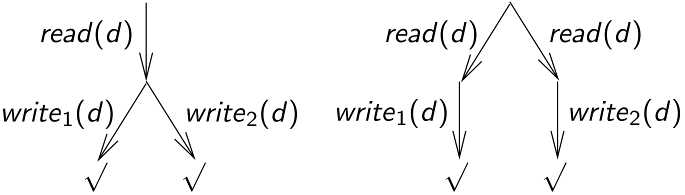
Basic Process Algebra - Axioms

$$
\begin{aligned}
x + y &= y + x \\
(x + y) + z &= x + (y + z) \\
x + x &= x \\
(x + y) \cdot z &= (x \cdot z) + (y \cdot z) \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z)
\end{aligned}
$$

No Left Distributivity

$x{\cdot}(y + z) \neq (x{\cdot}y) + (x{\cdot}z)$ does *not* hold.

Example:



Process one reads d, and then decides whether it writes d on disc 1 or 2. Process two makes a choice for disc 1 or 2 before it reads d. If disc 1 crashes, then process one saves datum d on disc 2, while process two may get stuck.

Bisimulation Equivalence

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

Assume a state space. A bisimulation is a binary relation $\mathcal{B}$ on states such that:

1. if $s_1 \, \mathcal{B} \, s_2$ and $s_1 \xrightarrow{a} s_1'$, then $s_2 \xrightarrow{a} s_2'$ with $s_1' \, \mathcal{B} \, s_2'$
2. if $s_1 \, \mathcal{B} \, s_2$ and $s_2 \xrightarrow{a} s_2'$, then $s_1 \xrightarrow{a} s_1'$ with $s_1' \, \mathcal{B} \, s_2'$
3. if $s_1 \, \mathcal{B} \, s_2$ and $s_1 \downarrow$, then $s_2 \downarrow$
4. if $s_1 \, \mathcal{B} \, s_2$ and $s_2 \downarrow$, then $s_1 \downarrow$

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

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

Soundness and Completeness of the Axioms

Theorem: For basic process algebra terms p and q:

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

Communication

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

$$\textbf{comm} \quad a\,|\,b \;=\; c$$

Communication is supposed to be commutative and associative.

$$\begin{array}{rcl} a\,|\,b & = & b\,|\,a \\ (a\,|\,b)\,|\,c & = & a\,|\,(b\,|\,c) \end{array}$$

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

In μCRL, 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 do not communicate,

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

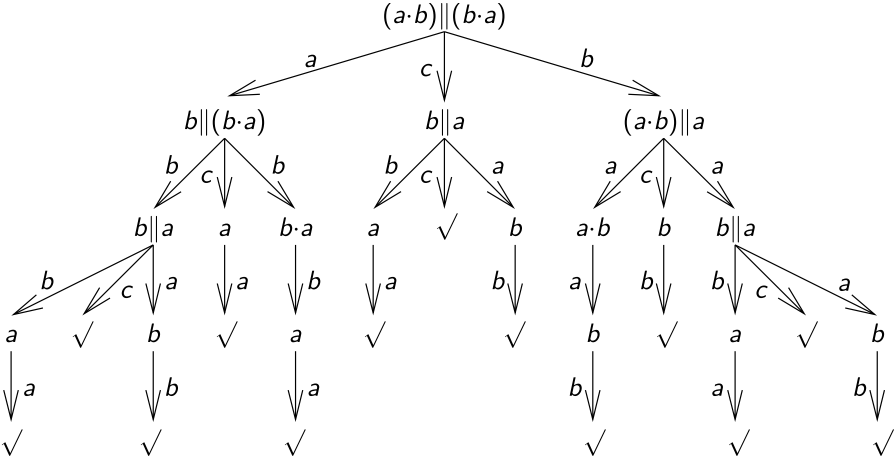
The merge can also execute a communication between actions of its arguments.

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



Question

Does $(x + y) \parallel z = (x \parallel z) + (y \parallel z)$ hold?

Left Merge and Communication Merge

The left merge $\mathbin{\underline{\|}}$ 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 $a \mid b = c$, then

$$a \mathbin{\underline{\|}} b = a \cdot b$$

$$a \mid b = c$$

These operators are needed to axiomatize the merge. In particular,

$$p \parallel q = (p \mathbin{\underline{\|}} q + q \mathbin{\underline{\|}} p) + p \mid q$$

Left Merge and Communication Merge

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

$x \llfloor (y + z) = (x \llfloor y) + (x \llfloor z)$ does not hold.

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

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

Parallelism - Axioms

$$x \parallel y = (x \mathbin{⌊\!⌊} y + y \mathbin{⌊\!⌊} x) + x \mid y$$

$$
\begin{array}{rcll}
a(\vec{d}) \mathbin{⌊\!⌊} x & = & a(\vec{d}) \cdot x & (\vec{d} \text{ denotes } d_1, \ldots, d_n) \\
(a(\vec{d}) \cdot x) \mathbin{⌊\!⌊} y & = & a(\vec{d}) \cdot (x \parallel y) & \\
(x + y) \mathbin{⌊\!⌊} z & = & x \mathbin{⌊\!⌊} z + y \mathbin{⌊\!⌊} z &
\end{array}
$$

$$
\begin{array}{rcll}
a(\vec{d}) \mid b(\vec{d}) & = & c(\vec{d}) & \text{if } a \mid b = c \\
a(\vec{d}) \mid b(\vec{e}) & = & \delta & \text{if } \vec{d} \neq \vec{e} \\
a(\vec{d}) \mid b(\vec{e}) & = & \delta & \text{if } a \mid b \text{ is undefined} \\
(a(\vec{d}) \cdot x) \mid b(\vec{e}) & = & (a(\vec{d}) \mid b(\vec{e})) \cdot x & \\
a(\vec{d}) \mid (b(\vec{e}) \cdot x) & = & (a(\vec{d}) \mid b(\vec{e})) \cdot x & \\
(a(\vec{d}) \cdot x) \mid (b(\vec{e}) \cdot y) & = & (a(\vec{d}) \mid b(\vec{e})) \cdot (x \parallel y) & \\
(x + y) \mid z & = & x \mid z + y \mid z & \\
x \mid (y + z) & = & x \mid y + x \mid z &
\end{array}
$$

Deadlock and Encapsulation

* The deadlock δ does not display any behavior.
* The encapsulation operators ∂_H, for sets of actions H, rename all actions of H in their argument into δ.

Encapsulation operators enable to enforce actions into communication.

Example: Let $s \,|\, r = c$.

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

Deadlock and Encapsulation - Axioms

$$
\begin{array}{rcll}
x + \delta & = & x & \\
\delta \cdot x & = & \delta & \\
\\
\partial_H(\delta) & = & \delta & \\
\partial_H(a(\vec{d})) & = & a(\vec{d}) & \text{if } a \notin H \\
\partial_H(a(\vec{d})) & = & \delta & \text{if } a \in H \\
\partial_H(x + y) & = & \partial_H(x) + \partial_H(y) & \\
\partial_H(x \cdot y) & = & \partial_H(x) \cdot \partial_H(y) & \\
\\
\delta \mathbin{⫫} x & = & \delta & \\
\delta \,|\, x & = & \delta & \\
x \,|\, \delta & = & \delta &
\end{array}
$$

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

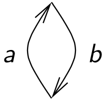
$$\partial_{\{s,r\}}((s(0) + s(1)) \parallel (r(0) + r(1)))$$

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

The axioms can be used to equate the process term above to

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

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

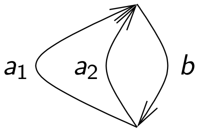
$$X(x_1{:}D_1, \ldots, x_n{:}D_n) \;=\; p$$

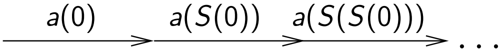
where the term p may contain expressions $Y(e_1, \ldots, e_m)$.

The initial declaration **init** consists of a single expression $X(d_1, \ldots, d_n)$, representing the initial state of the specification.

Question

How can one specify





Process Declaration - Example

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

act $\quad tick$

$\qquad display : Nat$

proc $\quad Clock(n{:}Nat) \;=\; tick \cdot Clock(S(n)) \;+\; display(n) \cdot Clock(n)$

init $\quad Clock(0)$

'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 is a data term of sort *Bool*, behaves as p if $b = \mathsf{T}$ and as q if $b = \mathsf{F}$.

$$\begin{aligned} x \triangleleft \mathsf{T} \triangleright y &= x \\ x \triangleleft \mathsf{F} \triangleright y &= y \end{aligned}$$

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

act $a, reset$

proc $Counter(n{:}Nat) =$

$$a \cdot Counter(S(n)) \triangleleft n < S(S(S(0))) \triangleright reset \cdot Counter(0)$$

init $Counter(0)$

Summation over a Data Type

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

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

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 μCRL, the distinction between **func** and **map** is used to build the set of constructor terms for summation over a data type.

Question

Let $send \,|\, recv = comm$.

What is the state space of

$$\partial_{\{send,recv\}}(send(S(0)) \| \sum_{n:Nat} recv(n)) \text{ ?}$$

Summation - Axioms

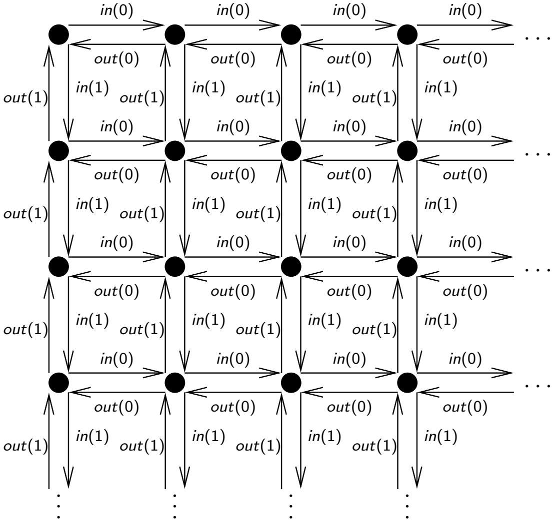
$$
\begin{array}{rcl}
\sum_{d:D} x & = & x \\
\sum_{d:D} P(d) & = & \sum_{d:D} P(d) + P(d_0) \qquad (d_0 \in D) \\
\sum_{d:D}(P(d) + Q(d)) & = & \sum_{d:D} P(d) + \sum_{d:D} Q(d) \\
(\sum_{d:D} P(d)) \cdot x & = & \sum_{d:D}(P(d) \cdot x)
\end{array}
$$

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

$$
\begin{array}{rcl}
(\sum_{d:D} P(d)) \mathbin{⌊\!⌊} x & = & \sum_{d:D}(P(d) \mathbin{⌊\!⌊} x) \\
(\sum_{d:D} P(d)) \mid x & = & \sum_{d:D}(P(d) \mid x) \\
x \mid (\sum_{d:D} P(d)) & = & \sum_{d:D}(x \mid P(d)) \\
\partial_H(\sum_{d:D} P(d)) & = & \sum_{d:D} \partial_H(P(d))
\end{array}
$$

Example - 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 $\{0, 1\}$:



Question

How could one specify the bag over $D = \{d_1, d_2\}$ in μCRL?

Example - Bag

If D is $\{d_1, d_2\}$, then a μCRL specification of the bag is:

$$\begin{array}{ll}
\textbf{act} & in, out : D \\
\textbf{proc} & Y(n{:}Nat, m{:}Nat) = in(d_1){\cdot}Y(S(n), m) \\
& \qquad + in(d_2){\cdot}Y(n, S(m)) \\
& \qquad + (out(d_1){\cdot}Y(P(n), m) \lhd n > 0 \rhd \delta) \\
& \qquad + (out(d_2){\cdot}Y(n, P(m)) \lhd m > 0 \rhd \delta) \\
\textbf{init} & Y(0, 0)
\end{array}$$

where $P(S(n)) = n$.

An alternative μCRL specification that works for general D is:

$$\begin{array}{ll}
\textbf{act} & in, out : D \\
\textbf{proc} & X \; = \; \sum_{d:D} in(d){\cdot}(X \parallel out(d)) \\
\textbf{init} & X
\end{array}$$

Hidden Action and Hiding

* The hidden action τ represents an internal computation step.
* The hiding operators τ_I, for $I \subset \text{Act}$, rename all actions of I in their argument into τ.

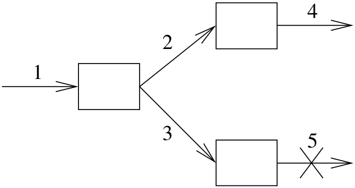
τ does not communicate with any action names.

Hiding operators can make actions inert.

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

$$\begin{aligned} a{\cdot}(b+\tau{\cdot}\delta) &\neq a{\cdot}b \\ \partial_{\{c\}}(a{\cdot}(b+\tau{\cdot}c)) &\neq \partial_{\{c\}}(a{\cdot}(b+c)) \\ a{\cdot}(b+\tau{\cdot}c) &\neq a{\cdot}(b+c) \end{aligned}$$

Solution: A hidden action is only inert if it does not lose possible behaviors.

Example: $a{\cdot}(b+\tau{\cdot}(b+c)) \;=\; a{\cdot}(b+c)$

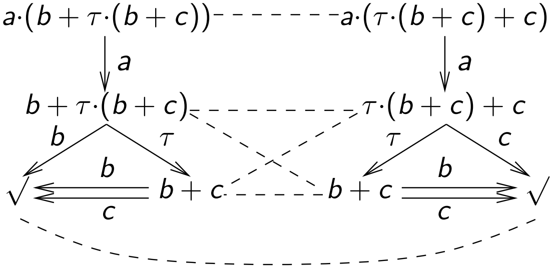
Branching Bisimulation Equivalence

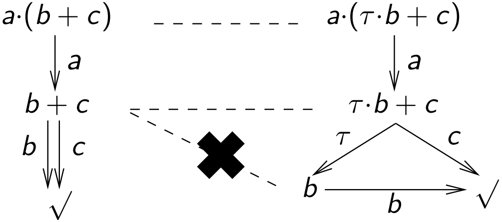
s_1 and s_2 are branching bisimilar states, denoted by $s_1 \underline{\leftrightarrow}_b s_2$, if:

- if $s_1 \xrightarrow{\tau} s_1'$ is inert, then $s_1' \underline{\leftrightarrow}_b s_2$
- a non-inert transition $s_1 \xrightarrow{a} s_1'$ (or $s_1 \downarrow$) is simulated by s_2 after zero or more inert τ-transitions: $s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} \hat{s}_2$, where $s_1 \underline{\leftrightarrow}_b \hat{s}_2$ and $\hat{s}_2 \xrightarrow{a} s_2'$ with $s_1' \underline{\leftrightarrow}_b s_2'$ (or $\hat{s}_2 \downarrow$)

and vice versa.

Branching Bisimulation - Examples





Which Hidden Actions Are Inert? - Part II

Initial τ's are not 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 does not lose possible behaviors and is not initial.

Rooted Branching Bisimulation Equivalence

s_1 and s_2 are rooted branching bisimilar, denoted $s_1 \underline{\leftrightarrow}_{rb}\ s_2$, if:

1. if $s_1 \xrightarrow{a} s_1'$, then $s_2 \xrightarrow{a} s_2'$ with $s_1' \underline{\leftrightarrow}_b\ s_2'$
2. if $s_1 \downarrow$, then $s_2 \downarrow$

and vice versa.

Hidden Action and Hiding - Axioms

$$
\begin{array}{rcll}
x{\cdot}\tau & = & x & \\
x{\cdot}(\tau{\cdot}(y+z)+y) & = & x{\cdot}(y+z) & \\
\\
\tau_I(\delta) & = & \delta & \\
\tau_I(\tau) & = & \tau & \\
\tau_I(a(\vec{d})) & = & a(\vec{d}) & \text{if } a \notin I \\
\tau_I(a(\vec{d})) & = & \tau & \text{if } a \in I \\
\tau_I(x+y) & = & \tau_I(x)+\tau_I(y) & \\
\tau_I(x{\cdot}y) & = & \tau_I(x){\cdot}\tau_I(y) & \\
\tau_I(\sum_{d:D} P(d)) & = & \sum_{d:D} \tau_I(P(d)) &
\end{array}
$$

Soundness and Completeness of the Axioms

Theorem: For process algebra terms p and q:

$$p = q \iff p \mathrel{\underline{\leftrightarrow}}_{rb} q$$

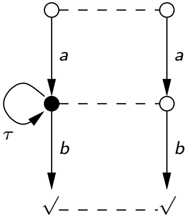
Fair Abstraction

τ-loops can be eliminated.

Example: The process X with

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

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** $a : D_1 \times \cdots \times D_n$, **comm** $a \,|\, b = c$)
* basic operators: $(+ \quad \cdot)$
* data-dependent operators: $(\triangleleft\, b \,\triangleright \quad \sum_{d:D})$
* process declaration: (**proc** $X(d_1, \ldots, d_n)$)
* parallel operators: $(\parallel \quad \llfloor \quad |\,)$
* deadlock and encapsulation: $(\delta \quad \partial_H)$
* hidden action and hiding: $(\tau \quad \tau_I)$

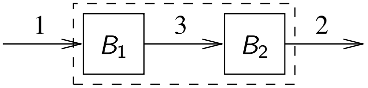
In general, the **init** declaration of a μCRL specification is of the form

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

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

Example - One-bit Buffers in Sequence

$$
\begin{array}{ll}
\textbf{act} & r_1, s_2, r_3, s_3, c_3 : D \\
\textbf{comm} & r_3 \,|\, s_3 = c_3 \\
\textbf{proc} & B_1 \;=\; \sum_{d:D} r_1(d)\cdot s_3(d)\cdot B_1 \\
 & B_2 \;=\; \sum_{d:D} r_3(d)\cdot s_2(d)\cdot B_2 \\
\textbf{init} & \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2))
\end{array}
$$



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

$$
\begin{array}{ll}
\textbf{proc} & X \;=\; \sum_{d:D} r_1(d)\cdot Y(d) \\
 & Y(d{:}D) \;=\; \sum_{d':D} r_1(d')\cdot Z(d, d') + s_2(d)\cdot X \\
 & Z(d{:}D, d'{:}D) \;=\; s_2(d)\cdot Y(d') \\
\textbf{init} & X
\end{array}
$$

Symbolic Proof Example: One-bit Buffers in Sequence

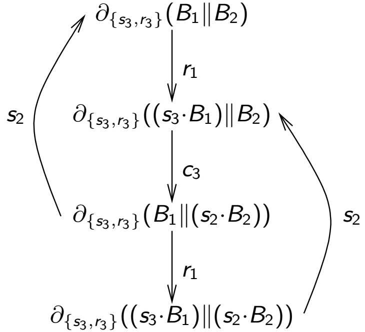
$$
\begin{array}{ll}
 & B_1 \parallel B_2 \qquad \text{(Summations } \Sigma_{d:D} \text{ and data parameters } d \text{ are omitted)} \\
= & B_1 \mathbin{⫫} B_2 \;+\; B_2 \mathbin{⫫} B_1 \;+\; B_1 \,|\, B_2 \\
= & (r_1 \cdot s_3 \cdot B_1) \mathbin{⫫} B_2 \;+\; (r_3 \cdot s_2 \cdot B_2) \mathbin{⫫} B_1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

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

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

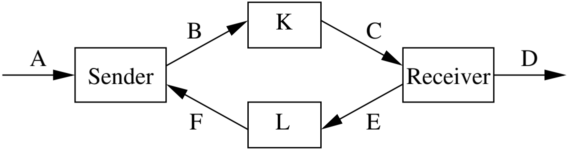
$$
\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel B_2)) &= \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
&= r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
&= r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(B_1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

Likewise we can derive:

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

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

Alternating Bit Protocol



Data elements are sent from *Sender* to *Receiver* via a corrupted channel. *Sender* alternatingly 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:

$$\begin{array}{rcl} S_b & = & \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{B}}(d,b) \cdot T_{db} \\ T_{db} & = & r_{\mathrm{F}}(b) \cdot S_{1-b} \\ & + & (r_{\mathrm{F}}(1-b) + r_{\mathrm{F}}(\bot)) \cdot s_{\mathrm{B}}(d,b) \cdot T_{db} \end{array}$$

R_b expects to receive a datum with b attached:

$$\begin{array}{rcl} R_b & = & \sum_{d:\Delta} r_{\mathrm{C}}(d,b) \cdot s_{\mathrm{D}}(d) \cdot s_{\mathrm{E}}(b) \cdot R_{1-b} \\ & + & \sum_{d:\Delta} r_{\mathrm{C}}(d,1-b) \cdot s_{\mathrm{E}}(1-b) \cdot R_b \\ & + & r_{\mathrm{C}}(\bot) \cdot s_{\mathrm{E}}(1-b) \cdot R_b \end{array}$$

Alternating Bit Protocol - Channels

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

$$K \;=\; \sum_{d:\Delta} \sum_{b:\{0,1\}} r_{\mathrm{B}}(d,b) \cdot (j \cdot s_{\mathrm{C}}(d,b) + j \cdot s_{\mathrm{C}}(\bot)) \cdot K$$

$$L \;=\; \sum_{b:\{0,1\}} r_{\mathrm{E}}(b) \cdot (j \cdot s_{\mathrm{F}}(b) + j \cdot s_{\mathrm{F}}(\bot)) \cdot L$$

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

$$s_{\mathrm{B}} \,|\, r_{\mathrm{B}} = c_{\mathrm{B}} \qquad s_{\mathrm{C}} \,|\, r_{\mathrm{C}} = c_{\mathrm{C}} \qquad s_{\mathrm{E}} \,|\, r_{\mathrm{E}} = c_{\mathrm{E}} \qquad s_{\mathrm{F}} \,|\, r_{\mathrm{F}} = c_{\mathrm{F}}$$

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.

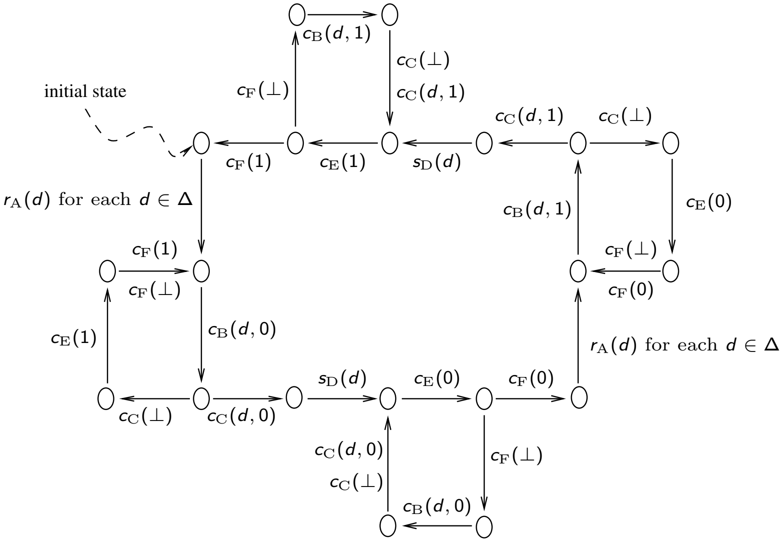
$\tau_I(\partial_H(R_0 \parallel S_0 \parallel K \parallel L))$ exhibits the desired external behavior

$$X \;=\; \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{D}}(d) \cdot X$$

Question: What is the behavior of $\tau_I(\partial_H(R_1 \parallel S_0 \parallel K \parallel L))$?

Alternating Bit Protocol - State Space

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



Question

What is the state space of

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

for $\Delta = \{d_1, d_2\}$, after minimization modulo $\underline{\leftrightarrow}_b$?

Alternating Bit Protocol - Symbolic Proof

A symbolic correctness proof of the alternating bit protocol, for any data set Δ, was checked with the theorem prover Coq.

(Bezem & Groote, 1993)

Question

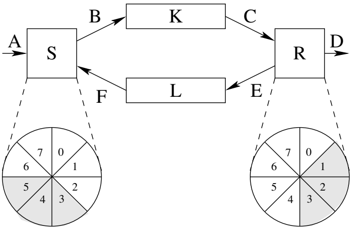
Which assumptions underlying the alternating bit protocol are unrealistic or impractical?

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

Sliding Window Protocol



The sliding window protocol has better use of available bandwidth.

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

B. Badban, W. Fokkink, J.F. Groote, J. Pang and J. van de Pol
Verification of a sliding window protocol in μCRL and PVS
Formal Aspects of Computing, 17(3):342-388, 2005

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*.

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

Model Checking Example: A Distributed Lift System

J. Pang, B. Karstens and W. Fokkink
Analyzing the redesign of a distributed lift system in UPPAAL
Proc. ICFEM'03, Singapore, Lecture Notes in Computer Science 2885, Springer, 2003

Linear Process Equation

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

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

with

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

Linearization - Type I and II Equations

Two types of recursive equations $X(d_1{:}D_1, \ldots, d_n{:}D_n) = p$ are distinguished:

- **I** p contains only $\cdot$, $+$, $\lhd\, b \,\rhd$, $\sum_{d:D}$
- **II** p also contains $\|$, ∂_H, τ_I

The μCRL lineariser requires that all recursion variables with a recursive equation of type II can be substituted away from right-hand sides of recursive equations and from the initial declaration.

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

Linearization of Type I Equations - Example

First we explain, by an example, the standard linearization method for type I equations (`mcrl`).

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.

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

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

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

is transformed into

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

If recursion variables carry data parameters, then function symbols, brackets and comma's are also pushed on the stack.
Here Y, Z do not carry data parameters, so D contains only Y, Z.

Disadvantage: The stack gives a lot of overhead.

Linearization with Pattern Matching - Example

`mcrl -regular` invokes another linearization algorithm for type I equations, which is based on pattern matching.

When the state space is finite, `mcrl -regular` usually works better than `mcrl`. But `mcrl -regular` does not always terminate.

Example:

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

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

Linearization with Pattern Matching - Example

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

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

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

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

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

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

Linearization with Pattern Matching - Example

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

$$\begin{aligned} X(n{:}Nat) &= a(n) \cdot Y(n) \\ Y(n{:}Nat) &= b(S(n)) \cdot c(S(S(n))) \cdot X(S(S(S(n)))) \end{aligned}$$

$$\begin{aligned} X(n{:}Nat) &= a(n) \cdot Y(n) \\ Y(n{:}Nat) &= b(S(n)) \cdot Z(n) \\ Z(n{:}Nat) &= c(S(S(n))) \cdot X(S(S(S(n)))) \end{aligned}$$

Linearization with Pattern Matching - Non-termination

`mcrl -regular` does not always terminate.

Example:

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

Linearization of Type II Equations - Example

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

Example: Let $a \mid b = c$, and

$$X(n{:}Nat) \;=\; a(n){\cdot}X(S(n)) \lhd n < 10 \rhd \delta \;+\; b(n){\cdot}X(S(S(n))) \lhd n > 5 \rhd \delta$$

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

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

Question

How can an application of τ_I to an LPE be reduced to an LPE?

How can an application of ∂_H to an LPE be reduced to an LPE?

State Space Generation

From an LPE $X(d{:}D)$ and initial state d_0, the state space is generated (`instantiator`). The algorithm below focuses on finding reachable *states* (i.e., transitions are ignored).

M contains all "explored" states, and L the generated states that still need to be explored.

Initially, $L = \{d_0\}$ and $M = \emptyset$.

while $L \neq \emptyset$ **do**

select $d \in L$; $L := L \backslash \{d\}$; $M := M \cup \{d\}$

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

if $d' \notin L \cup M$ **then** $L := L \cup \{d'\}$

Challenges: Store large state spaces in memory.

Check efficiently whether $d' \notin L \cup M$.

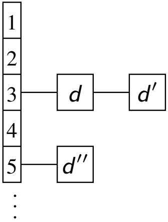
Hash Tables

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

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

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' is *not* in the hash table, the check $d' \notin L \cup M$ requires an expensive disk lookup.

A Bloom filter allows an inexpensive check whether $d' \notin L \cup M$, allowing for false positives.

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

A Bloom filter is a bit array of length m.

Initially, all bits are set to 0.

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

Bloom Filter

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

If not, then $d' \notin L \cup M$.

Else, still an expensive disk lookup is required.

Bloom Filter - Analysis

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

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

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

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

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

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.

Distributed Verification

Store the state space on a cluster of computers (e.g. DAS-3).

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.

Distributed versions exist of:

- state space generation
- minimization modulo $\underline{\leftrightarrow}_b$
- model checking

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

Distributed Verification - Example

We model checked a cache coherence protocol for a *distributed shared memory implementation* of Java.

For 2 processors, each with 1 thread, the protocol is correct.

For 2 processors, one with 1 and one with 2 threads, with distributed model checking, we detected a deadlock.

Namely, while a thread is waiting for the write lock of a region, the home node of the region may migrate to the thread's processor, so that the thread actually accesses the region at home.

J. Pang, W. Fokkink, R. Hofman and R. Veldema
Model checking a cache coherence protocol for a Java DSM implementation
Journal of Logic and Algebraic Programming, 71(1):1-43, 2007

Sorted Lists to Fight State Explosion

Storing the message buffers of processes, and messages in channels, in a sorted list, reduces the number of states considerably.

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

Model Checking

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

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

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

- T holds in all states, and 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 ϕ holds in state s'

  $[a]\, \phi$ holds in state s if for each transition $s \xrightarrow{a} s'$, ϕ holds in state s'

Question

Does $\langle a \rangle \, \phi$ imply $[a] \, \phi$?

Does $[a] \, \phi$ imply $\langle a \rangle \, \phi$?

Model Checking

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

$$
\begin{array}{ll}
s \models \mathsf{T} & \\
s \not\models \mathsf{F} & \\
s \models \phi \wedge \phi' & \text{if } s \models \phi \textit{ and } s \models \phi' \\
s \models \phi \vee \phi' & \text{if } s \models \phi \textit{ or } s \models \phi' \\
s \models \langle a \rangle\, \phi & \text{if for } \textit{some} \text{ state } s',\ s \xrightarrow{a} s' \text{ with } s' \models \phi \\
s \models [a]\, \phi & \text{if for } \textit{all} \text{ states } s',\ s \xrightarrow{a} s' \text{ implies } s' \models \phi
\end{array}
$$

Example: If $s \overset{a}{\nrightarrow}$, then $s \models [a]\,\mathsf{F}$ and $s \not\models \langle a \rangle\,\mathsf{T}$.

Fixpoints

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

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

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

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

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

Question

Give an example to show that if D is *infinite*, monotonic mappings $\mathcal{S} : D \to D$ need not have a fixpoint.

μ-calculus

The μ-calculus is a temporal logic.

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

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 μX or νX.

We need to explain how ϕ (with X as only free variable) is interpreted as a (monotonic) mapping from sets of states to sets of states.

μ-calculus

Consider a *finite* state space.
(For simplicity we ignore successful termination.)

Let X be the *only free variable* in μ-calculus formula ϕ.
We define the meaning of $\mu X.\phi$ and $\nu X.\phi$.

ϕ maps each set P of states to the set of states that satisfy ϕ, 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 $\{s_1\}$.

μ-calculus

As partial order we take set inclusion.

Theorem: For each ϕ 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 only by the states in
the minimal and maximal fixpoint of ϕ, respectively.

μ-calculus - Examples

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

$\nu X.(\langle a \rangle X \vee \langle b \rangle \mathsf{T})$ represents those states that can execute a^∞ 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 $\mu X.(\langle a \rangle X \vee \langle b \rangle X)$?

μ-calculus - Negation Violates Monotonicity

Absence of negation in the μ-calculus is needed for monotonicity.

Example:

$$s_0 \underset{a}{\overset{a}{\rightleftarrows}} s_1$$

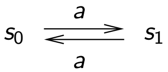
$\mu X.\neg \langle a \rangle X$ has no fixpoint.

μ-calculus - Complexity

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

where $N(\phi)$ is the longest chain of nested fixpoints in ϕ.

Example:



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

Y	X
$\emptyset$	$\{s_0, s_1\}$
$\{s_0, s_1\}$	$\emptyset$
$\emptyset$	$\emptyset$

In the second iteration, recomputation of Y *must* start at $\emptyset$ (instead of $\{s_0, s_1\}$).

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

Alternation-free μ-calculus

For two nested minimal (or maximal) fixpoints, recomputing a fixpoint is not so expensive.

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

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

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

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 μ-calculus

Worst-case time complexity: $O(|\phi|\cdot m\cdot n^{N(\phi)})$
for model checking the μ-calculus, where $N(\phi)$ is the longest chain of nested *alternating* fixpoints in ϕ (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 μ-calculus.

Model checking the *full* μ-calculus is in $\mathrm{NP} \cap \text{co-NP}$.

It is an open question whether it is in P.

Regular μ-calculus

$$\begin{array}{rcl}\alpha & ::= & \mathsf{T} \mid a \mid \neg\alpha \mid \alpha \wedge \alpha' \quad (a \in \mathsf{Act} \cup \{\tau\}) \\ \beta & ::= & \alpha \mid \beta\cdot\beta' \mid \beta|\beta' \mid \beta^* \\ \phi & ::= & \mathsf{T} \mid \mathsf{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle\beta\rangle\,\phi \mid [\beta]\,\phi \mid X \mid \mu X.\phi \mid \nu X.\phi\end{array}$$

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

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

Regular μ-calculus - Examples

Deadlock freeness: $[\mathsf{T}^*]\,\langle \mathsf{T}\rangle\,\mathsf{T}$

Absence of *error*: $[\mathsf{T}^*\cdot \mathit{error}]\,\mathsf{F}$

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

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

Question: Specify the properties:

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

Regular μ-calculus - Examples

There is an infinite execution sequence:

$$\nu X.(\langle \mathsf{T} \rangle\, X)$$

No reachable state exhibits an infinite τ-sequence:

$$[\mathsf{T}^*]\, \mu X.[\tau]\, X$$

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

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

CADP Model Checker

CADP supports model checking of alternation-free, regular μ-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, one can include artificial self-loops that carry these variables as action variables.

Example: To the process declaration of a recursion variable $X(d_1{:}D_1, d_2{:}D_2, d_3{:}D_3)$ one can add a summand

$$+\ \mathit{test}(d_1, d_3) \cdot X(d_1, d_2, d_3)$$

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

Such a self-loop does not increase the number of reachable states.

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 is not in the uppermost position, and the Up button is pressed, then the bed must go up.)

Pitfalls for Requirements

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?)

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

Example: *"the controllers must communicate asynchronously"*

(Implementation detail, cannot be verified using model checking.)

Minimization Algorithm

Assume a finite state space.

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

For $a \in \mathrm{Act} \cup \{\tau\}$, $s_0 \in \mathit{split}_a(P_i, P_j)$ if
$s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$ with $s_0, \ldots, s_{n-1} \in P_i$ and $s_n \in P_j$.

If $s \in \mathit{split}_a(P_i, P_j)$ and $s' \in P_i \backslash \mathit{split}_a(P_i, P_j)$ 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 in S. (So $(*)$ is trivially satisfied.)

Suppose that at some point S is partitioned in $P_1 \cup \cdots \cup P_k$.

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

$$\emptyset \subset split_a(P_i, P_j) \subset P_i$$

then in the partition, P_i can be replaced by $split_a(P_i, P_j)$ and $P_i \backslash split_a(P_i, P_j)$.

Splitting continues until no further split is possible.

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

Minimization Algorithm - Correctness and Complexity

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.

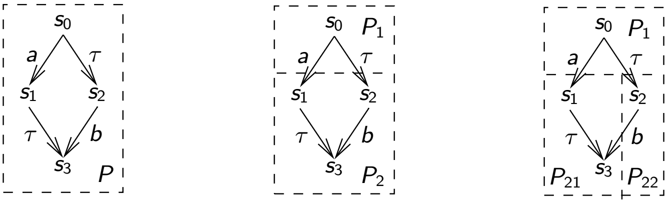
Theorem: Let $P_1 \cup \cdots \cup P_k$ denote the final partition of S. Two states s and s' are in the same set P_i if and only if $s \,\underline{\leftrightarrow}_b\, s'$ in the original state space.

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

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

Minimization Algorithm - Example

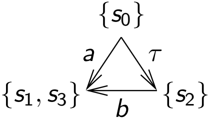
Consider $(a{\cdot}\tau + \tau{\cdot}b){\cdot}\delta$. P contains all four states in the state space.



$split_a(P, P)$ 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} cannot be split any further, so the minimized state space is



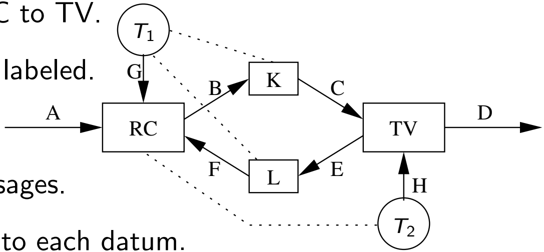
Questions

How can we, in the previous example, split on b followed by a split on a?

How is $(a{\cdot}\tau + \tau{\cdot}a){\cdot}\delta$ minimized?

How is $a{\cdot}a{\cdot}\delta$ minimized?

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.

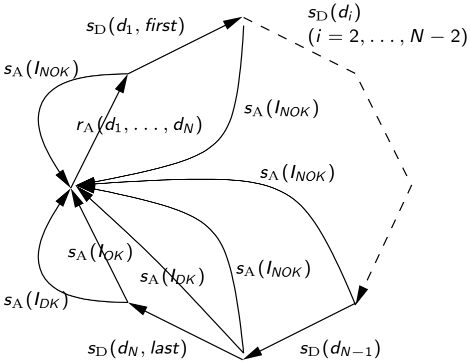
Only one kind of acknowledgement.

If RC does not receive an acknowledgement within a certain time, it resends the datum.

A datum is resent a limited number of times.

If TV does not 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 Λ consist of lists over Δ. (Only lists of length ≥ 2 can be transmitted.)

$$X \;=\; \sum_{\lambda:\Lambda} r_{\mathrm{A}}(\lambda)\cdot Y(\lambda, 0, S(0)) \lhd \mathit{length}(\lambda) > S(0) \rhd \delta$$

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

$$(s_{\mathrm{B}}(\mathit{head}(\lambda), b) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{B}}(\mathit{head}(\lambda), b, \mathit{last}))\cdot Z(\lambda, b, n)$$

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

$$r_{\mathrm{F}}(\mathit{ack})\cdot(Y(\mathit{tail}(\lambda), 1{-}b, S(0)) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{A}}(I_{OK})\cdot X)$$

$$+\; r_{\mathrm{G}}(\mathit{to})\cdot(Y(\lambda, b, S(n)) \lhd n < \mathit{max} \rhd$$

$$(s_{\mathrm{A}}(I_{NOK}) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{A}}(I_{DK}))\cdot s_{\mathrm{H}}(\mathit{to})\cdot X)$$

Bounded Retransmission Protocol - Television

$$
\begin{array}{rcl}
V & = & \sum_{d:\Delta} r_{\mathrm{C}}(d,0)\cdot s_{\mathrm{D}}(d,\mathit{first})\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(1) \\
& + & \sum_{d:\Delta}(r_{\mathrm{C}}(d,0,\mathit{last}) + r_{\mathrm{C}}(d,1,\mathit{last}))\cdot s_{\mathrm{E}}(\mathit{ack})\cdot V \\
& + & r_{\mathrm{H}}(\mathit{to})\cdot V
\end{array}
$$

$$
\begin{array}{rcl}
W(b{:}\mathit{Bit}) & = & \sum_{d:\Delta} r_{\mathrm{C}}(d,b)\cdot s_{\mathrm{D}}(d)\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(1-b) \\
& + & \sum_{d:\Delta} r_{\mathrm{C}}(d,b,\mathit{last})\cdot s_{\mathrm{D}}(d,\mathit{last})\cdot s_{\mathrm{E}}(\mathit{ack})\cdot V \\
& + & \sum_{d:\Delta} r_{\mathrm{C}}(d,1-b)\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(b) \\
& + & r_{\mathrm{H}}(\mathit{to})\cdot V
\end{array}
$$

Bounded Retransmission Protocol - Medium

$$
\begin{aligned}
K &= \textstyle\sum_{d:\Delta} \sum_{b:\{0,1\}} (r_{\mathrm{B}}(d,b)\cdot(j\cdot s_{\mathrm{C}}(d,b) + j\cdot s_{\mathrm{G}}(to))\cdot K \\
&+ \; r_{\mathrm{B}}(d,b,last)\cdot(j\cdot s_{\mathrm{C}}(d,b,last) + j\cdot s_{\mathrm{G}}(to))\cdot K)
\end{aligned}
$$

$$
L = r_{\mathrm{E}}(ack)\cdot(j\cdot s_{\mathrm{F}}(ack) + j\cdot s_{\mathrm{G}}(to))\cdot L
$$

Bounded Retransmission Protocol - Initial State

The initial state is specified by

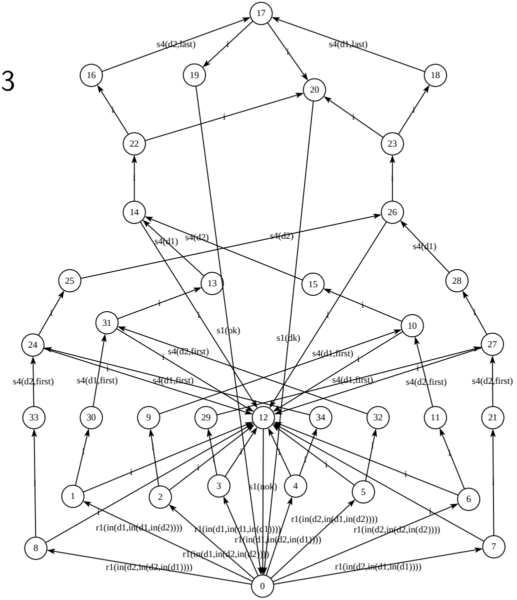
$$\tau_I(\partial_H(V \parallel X \parallel K \parallel L))$$

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

Bounded Retransmission Protocol - External Behavior

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

Λ consists of lists of length 3



Bounded Retransmission Protocol - Exercises

- An incomplete specification of the BRP is available at

  `http://www.cs.vu.nl/~tcs/pv/brp-scheme`

  You need to supply rewrite rules for the functions *smaller*, *head*, *tail* and *length*, and specify the TV.

- Generate the minimized state space for $\Delta = \{d_1, d_2\}$.

- Formulate the next properties in regular μ-calculus:
  - Each action $r_{\mathrm{A}}(\ell)$ from the remote control is eventually followed by an action $s_{\mathrm{A}}(I_{OK})$ or $s_{\mathrm{A}}(I_{NOK})$ or $s_{\mathrm{A}}(I_{DK})$ from the remote control.
  - Each action $s_{\mathrm{A}}(I_{OK})$ from the remote control is preceded by an action $s_{\mathrm{D}}(d, \mathit{last})$ from the television.

  Verify these two properties using the CADP model checker.

Bounded Retransmission Protocol - Exercises

- Suppose that the timer T_1 is absent.

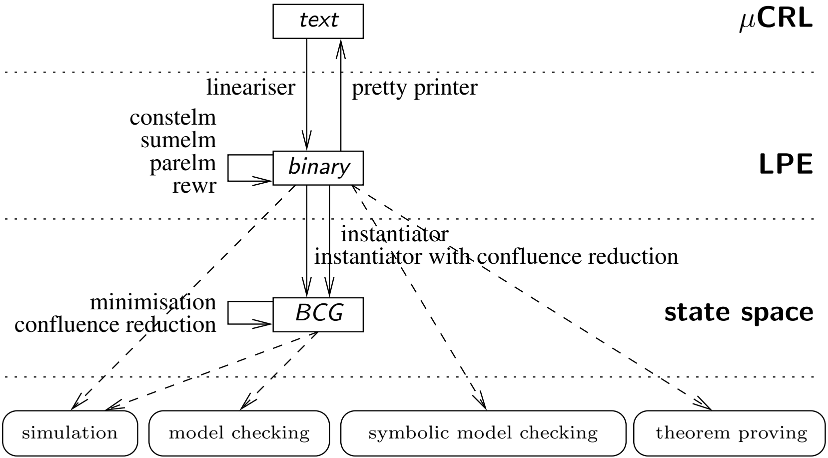
  Verify using the CADP toolset that this version of the BRP contains a deadlock.

- What would go wrong if the timer T_2 were absent from the BRP?

  Would the resulting system contain a deadlock?

  Analyse the state space of this system using the CADP toolset.

Overview of the μCRL Toolset



Automated Theorem Prover

A theorem prover within the μCRL toolset provides (semi-)automated support for proving (large) formulas.

This theorem prover is *not* complete!
(Equalities over an abstract data type are in general undecidable.)

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

In some cases, the formula is not valid in all states of the system, but does hold in all reachable states.
The user may supply invariants, and formulas can be proved under the assumption of invariants.
Such invariants must be proved separately, which can again be done using the theorem prover.

Download μCRL, CADP and mCRL2

μCRL can be downloaded at

`http://homepages.cwi.nl/~mcrl`

CADP can be downloaded at

`http://www.inrialpes.fr/vasy/cadp/`

mCRL2, a successor of μCRL with functional data types that is being developed at Eindhoven University of Technology, can be downloaded at

`http://www.mcrl2.org`

Other Useful Links

At the web site of my course *Protocol Validation*

`http://www.cs.vu.nl/~tcs/pv/`

you can find exercises, lab assignments and solutions.

My book *Modelling Distributed Systems* can be downloaded at

`http://www.springer.com`

Summary

Basic framework

- abstract data types
- process algebra
- branching bisimilarity

State space

- linearisation / state space generation
- minimisation of the state space
- regular μ-calculus / model checking

What's More

Fighting state space explosion

- store state space on a cluster of computers
- partial order reduction
- abstraction
- symbolic model checking
- theorem prover

Symbolic proof techniques

- axioms
- linear process equations
- invariants
- cones and foci