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
A μCRL system specification consists of algebraic specifications of:

- data types
- parallel system components

An algebraic specification of data types consists of:

- A signature consisting of function symbols. They are the building blocks of terms.
- Axioms, i.e. equations between terms. They induce an equality relation on terms, which is closed under (1) equivalence, (2) substitution, and (3) context.
The signature of the natural numbers consists of constant 0, unary successor $S$, and binary addition $\text{plus}$ and multiplication $\text{mul}$.

The axioms are:
\[
\begin{align*}
\text{plus}(n, 0) &= n \\
\text{plus}(n, S(m)) &= S(\text{plus}(n, m)) \\
\text{mul}(n, 0) &= 0 \\
\text{mul}(n, S(m)) &= \text{plus}(\text{mul}(n, m), n)
\end{align*}
\]

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

There is a MSc course Term Rewriting Systems.
Constructors

µCRL 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.

\begin{align*}
\textbf{sort} & \quad \textit{Nat} \\
\textbf{func} & \quad 0 : \rightarrow \textit{Nat} \\
& \quad S : \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{map} & \quad plus, mul : \textit{Nat} \times \textit{Nat} \rightarrow \textit{Nat} \\
\textbf{var} & \quad n, m : \textit{Nat} \\
\textbf{rew} & \quad plus(n, 0) = n \\
& \quad plus(n, S(m)) = S(plus(n, m)) \\
& \quad mul(n, 0) = 0 \\
& \quad mul(n, S(m)) = plus(mul(n, m), n)
\end{align*}
Normal forms

A term is a normal form if it can’t be rewritten.

It should consist of only constructor symbols.

Rewriting of data terms is performed according to the innermost strategy:

A term $f(d_1, \ldots, d_n)$ is only rewritten if its arguments $d_1, \ldots, d_n$ are normal forms.

**Question:** Is it a good idea to add $\text{plus}(n, m) = \text{plus}(m, n)$ to the specification of the natural numbers?
Booleans

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

<table>
<thead>
<tr>
<th>sort</th>
<th>$\text{Bool}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>$T, F : \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>map</td>
<td>$\land, \lor : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td></td>
<td>$\neg : \text{Bool} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>var</td>
<td>$b : \text{Bool}$</td>
</tr>
<tr>
<td>rew</td>
<td>$b \land T = b$</td>
</tr>
<tr>
<td></td>
<td>$b \land F = F$</td>
</tr>
<tr>
<td></td>
<td>$b \lor T = T$</td>
</tr>
<tr>
<td></td>
<td>$b \lor F = b$</td>
</tr>
<tr>
<td></td>
<td>$\neg T = F$</td>
</tr>
<tr>
<td></td>
<td>$\neg F = T$</td>
</tr>
</tbody>
</table>
If-then-else function

A help function

\[ \text{if} : \text{Bool} \times D \times D \to D \]

with as equations

\[ \text{if}(T, d, e) = d \]
\[ \text{if}(F, d, e) = e \]

is often convenient in the specification of data types.
One needs to define an equality function \( \text{eq} : D \times D \to \text{Bool} \) for data types \( D \), where \( \text{eq}(d, e) = \text{T} \) if and only if \( d = e \).

**Example:**

\[
\begin{align*}
\text{map} & \quad \text{eq} : \text{Bool} \times \text{Bool} \to \text{Bool} \\
\text{rew} & \quad \text{eq}(\text{T}, \text{T}) = \text{T} \\
& \quad \text{eq}(\text{F}, \text{F}) = \text{T} \\
& \quad \text{eq}(\text{T}, \text{F}) = \text{F} \\
& \quad \text{eq}(\text{F}, \text{T}) = \text{F}
\end{align*}
\]

\[
\begin{align*}
\text{map} & \quad \text{eq} : \text{Nat} \times \text{Nat} \to \text{Bool} \\
\text{var} & \quad n, m : \text{Nat} \\
\text{rew} & \quad \text{eq}(0, 0) = \text{T} \\
& \quad \text{eq}(S(n), S(m)) = \text{eq}(n, m) \\
& \quad \text{eq}(0, S(n)) = \text{F} \\
& \quad \text{eq}(S(n), 0) = \text{F}
\end{align*}
\]
A shorter specification of the equality function on booleans is:

\[
\begin{align*}
eq(b, b) &= T \\
eq(T, F) &= F \\
eq(F, T) &= F
\end{align*}
\]

The next specification of an equality function doesn’t work in \( \mu \text{CRL} \):

\[
\begin{align*}
eq(x, x) &= T \\
eq(x, y) &= F
\end{align*}
\]

That is, rewrite rules aren’t necessarily ‘executed’ from top to bottom.
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 $T$: $\neg\neg T = \neg F = T$
- $b$ is $F$: $\neg\neg F = \neg T = F$

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

- **Base case, $n$ is 0**: $\text{plus}(0, 0) = 0$
- **Inductive case, $n$ is $S(m)$**: $\text{plus}(0, S(m)) = S(\text{plus}(0, m)) = S(m)$

**Question:** Prove $\text{mul}(S(0), n) = n$ by induction on $n$. 
Basic process terms are built from parametrized actions in a set Act, alternative composition and sequential composition.

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

- 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


text

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

\[
((a + b) \cdot c) \cdot d
\]

13 / 224
The link between a process term $p$ and its transitions $p \xrightarrow{a} p'$ and $p \xrightarrow{a} \sqrt{}$ can be formally defined by transition rules.

$$a(\vec{d}) \xrightarrow{a(\vec{d})} \sqrt{}$$

$$\begin{align*}
x_1 & \xrightarrow{a(\vec{d})} \sqrt{} \\
x_1 + x_2 & \xrightarrow{a(\vec{d})} \sqrt{}
\end{align*}$$

$$\begin{align*}
x_1 & \xrightarrow{a(\vec{d})} y \\
x_1 + x_2 & \xrightarrow{a(\vec{d})} y
\end{align*}$$

$$\begin{align*}
x_1 & \xrightarrow{a(\vec{d})} \sqrt{} \\
x_1 \cdot x_2 & \xrightarrow{a(\vec{d})} x_2
\end{align*}$$

$$\begin{align*}
x_1 & \xrightarrow{a(\vec{d})} y \\
x_1 \cdot x_2 & \xrightarrow{a(\vec{d})} y \cdot x_2
\end{align*}$$
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 \( B \) on states such that if \( s_1 B s_2 \), then:

1. \( s_1 \xrightarrow{a} s'_1 \) implies that there is a transition \( s_2 \xrightarrow{a} s'_2 \) with \( s'_1 B s'_2 \)
2. \( s_2 \xrightarrow{a} s'_2 \) implies that there is a transition \( s_1 \xrightarrow{a} s'_1 \) with \( s'_1 B s'_2 \)
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 \( B \) such that \( s_1 B s_2 \).

**Example:** \( a \cdot (b + c) \not\leftrightarrow (a \cdot b) + (a \cdot c) \)
Theorem: For basic process algebra terms $p$ and $q$:

$$p = q \iff p \leftrightarrow q$$
To specify that two action names can communicate (or synchronize):

\[
\text{comm} \quad a \| b = c
\]

Communication is commutative and associative.

\[
a \| b = b \| a \\
(a \| b) \| c = a \| (b \| c)
\]

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 \(\mu\text{CRL}\), the equality function only needs to be defined for data types that are used in parameters of actions that can communicate.
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 \( a \mid b = c \),

\[
a \parallel b = (a \cdot b + b \cdot a) + c
\]
Parallelism - Transition rules

\[
\begin{align*}
\frac{x_1 \overset{a(\vec{d})}{\rightarrow} \sqrt{\cdot}}{x_1 \parallel x_2 \overset{a(\vec{d})}{\rightarrow} x_2} & \quad & \frac{x_1 \overset{a(\vec{d})}{\rightarrow} y}{x_1 \parallel x_2 \overset{a(\vec{d})}{\rightarrow} y \parallel x_2} & \quad & \frac{x_2 \overset{a(\vec{d})}{\rightarrow} \sqrt{\cdot}}{x_1 \parallel x_2 \overset{a(\vec{d})}{\rightarrow} x_1} & \quad & \frac{x_2 \overset{a(\vec{d})}{\rightarrow} y}{x_1 \parallel x_2 \overset{a(\vec{d})}{\rightarrow} x_1 \parallel y} \\
x_1 \overset{a(\vec{d})}{\rightarrow} y & \quad & x_2 \overset{b(\vec{d})}{\rightarrow} \sqrt{\cdot} & \quad & x_1 \parallel x_2 \overset{c(\vec{d})}{\rightarrow} \sqrt{\cdot} & \quad & x_1 \overset{a(\vec{d})}{\rightarrow} y_2 & \quad & x_2 \overset{b(\vec{d})}{\rightarrow} y_2 \\
x_1 \overset{a(\vec{d})}{\rightarrow} y_1 & \quad & x_2 \overset{b(\vec{d})}{\rightarrow} \sqrt{\cdot} & \quad & x_1 \parallel x_2 \overset{c(\vec{d})}{\rightarrow} y_1 & \quad & x_1 \overset{a(\vec{d})}{\rightarrow} y_1 & \quad & x_2 \overset{b(\vec{d})}{\rightarrow} y_2 \\
\end{align*}
\]

In the last four rules, \(a\) and \(b\) carry the same data parameters, and \(a \parallel b = c\).
If all communications between action names result to $c$, then

\[(a \cdot b) \parallel (b \cdot a)\]
The **left merge** $\sqcup$ 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

\[
\begin{align*}
  a \sqcup b &= a \cdot b \\
  a|b &= c
\end{align*}
\]

These operators are needed to axiomatize the merge:

\[
p \parallel q = (p \sqcup q + q \sqcup p) + p|q
\]
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.

\[(x + y) \mid z = (x \mid z) + (y \mid z)\] holds.

\[x \mid (y + z) = (x \mid y) + (x \mid z)\] holds.
Parallelism - Axioms

\[ x \parallel y = (x \parallel y + y \parallel x) + x \mid y \]

\[ a(\vec{d}) \hat{\bot} y = a(\vec{d}) \cdot y \]

\[ (a(\vec{d}) \cdot x) \hat{\bot} y = a(\vec{d}) \cdot (x \parallel y) \]

\[ (x + y) \hat{\bot} z = x \hat{\bot} z + y \hat{\bot} z \]

\[ a(\vec{d}) \mid b(\vec{d}) = c(\vec{d}) \quad \text{if } a \mid b = c \]

\[ a(\vec{d}) \mid b(\vec{e}) = \delta \quad \text{if } \vec{d} \neq \vec{e} \text{ or } a \mid b \text{ is undefined} \]

\[ (a(\vec{d}) \cdot x) \mid b(\vec{e}) = (a(\vec{d}) \mid b(\vec{e})) \cdot x \]

\[ a(\vec{d}) \mid (b(\vec{e}) \cdot y) = (a(\vec{d}) \mid b(\vec{e})) \cdot y \]

\[ (a(\vec{d}) \cdot x) \mid (b(\vec{e}) \cdot y) = (a(\vec{d}) \mid b(\vec{e})) \cdot (x \parallel y) \]

\[ (x + y) \mid z = x \mid z + y \mid z \]

\[ x \mid (y + z) = x \mid y + x \mid z \]
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 $s | r = c$.

\[
\begin{align*}
s \parallel r & = (s \cdot r + r \cdot s) + c \\
\partial_{\{s, r\}}(s \parallel r) & = c
\end{align*}
\]
Deadlock and encapsulation - Axioms

\[ x + \delta = x \]
\[ \delta \cdot x = \delta \]
\[ \partial_H(\delta) = \delta \]
\[ \partial_H(a(\vec{d})) = a(\vec{d}) \quad \text{if } a \notin H \]
\[ \partial_H(a(\vec{d})) = \delta \quad \text{if } a \in H \]
\[ \partial_H(x + y) = \partial_H(x) + \partial_H(y) \]
\[ \partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y) \]
Soundness and completeness of the axioms

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

\[ p = q \iff p \leftrightarrow q \]
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 (synchronous) 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

\[(s(0) + s(1)) \| (r(0) + r(1))\]

\[= (s(0) + s(1)) \parallel (r(0) + r(1)) + (r(0) + r(1)) \parallel (s(0) + s(1)) + (s(0) + s(1)) \| (r(0) + r(1))\]

\[= s(0) \parallel (r(0) + r(1)) + s(1) \parallel (r(0) + r(1)) +\]
\[r(0) \parallel (s(0) + s(1)) + r(1) \parallel (s(0) + s(1)) +\]
\[s(0) | r(0) + s(0) | r(1) + s(1) | r(0) + s(1) | r(1)\]

\[= s(0) \cdot (r(0) + r(1)) + s(1) \cdot (r(0) + r(1)) +\]
\[r(0) \cdot (s(0) + s(1)) + r(1) \cdot (s(0) + s(1)) + c(0) + \delta + \delta + c(1)\]

\[= s(0) \cdot (r(0) + r(1)) + s(1) \cdot (r(0) + r(1)) +\]
\[r(0) \cdot (s(0) + s(1)) + r(1) \cdot (s(0) + s(1)) + c(0) + c(1)\]
Example - Bits through a channel

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

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

Recursion variables \( X \) and \( Y \) represent the states of the process.

A process declaration \textbf{proc} consists of recursive equations

\[ X(d_1:D_1, \ldots, d_n:D_n) = p \]

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

The initial declaration \textbf{init}, i.e. the initial state of the specification, is an expression \( X(t_1, \ldots, t_n) \) with \( t_1, \ldots, t_n \) closed data terms.
The process $Clock$ repeatedly performs action $tick$ or displays the current time.

\begin{verbatim}
act       display : Nat
tick

proc     Clock(n: Nat) = display(n) · Clock(n) + tick · Clock(S(n))
init     Clock(0)
\end{verbatim}

‘Unguarded’ process declarations such as $X = X$ and $Y = Y · a$ are illegal.
Proof principles for recursion

- A recursion variable can be replaced by its right-hand side in the process declaration.

- Each (guarded) process declaration has a unique solution.

Consider the process declarations $X = a \cdot X$ and $Y = a \cdot a \cdot Y$.

$X = a \cdot X = a \cdot a \cdot X$.

Hence $X = Y$. 
The process term \( p \triangleleft b \triangleright q \), where \( p \) and \( q \) are process terms, and \( b \) a data term of sort \( \text{Bool} \), behaves as \( p \) if \( b = \text{T} \) and as \( q \) if \( b = \text{F} \).

\[
\begin{align*}
  x \triangleleft \text{T} \triangleright y &= x \\
  x \triangleleft \text{F} \triangleright y &= y
\end{align*}
\]

**Example:** The process \( \text{Clock} \) is reset after three time steps:

\[
\begin{align*}
\text{act} & \quad \text{display} : \text{Nat} \\
& \quad \text{tick} \\
\text{proc} & \quad \text{Clock}(n: \text{Nat}) = \text{display}(n) \cdot \text{Clock}(n) + \\
& \quad \quad \quad \text{tick} \cdot \text{Clock}(S(n)) \triangleleft n < S(S(S(0))) \triangleright \text{tick} \cdot \text{Counter}(0) \\
\text{init} & \quad \text{Counter}(0)
\end{align*}
\]
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 $\mu$CRL, the distinction between *func* and *map* is used to build the set of constructor terms for summation over a data type.
\[
\sum_{d:D} x = x
\]

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

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

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

\[
(\forall d:D \ P(d) = Q(d)) \Rightarrow \sum_{d:D} P(d) = \sum_{d:D} Q(d)
\]
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 $\{d, e\}$:
If $D$ is $\{d, e\}$, then a $\mu$CRL specification of the bag is:

**act** \(\text{in, out} : D\)

**proc** \(Y(n: \text{Nat}, m: \text{Nat}) = \text{in}(d) \cdot Y(S(n), m) + \text{in}(e) \cdot Y(n, S(m)) + (\text{out}(d) \cdot Y(P(n), m) \triangleleft n > 0 \triangleright \delta) + (\text{out}(e) \cdot Y(n, P(m)) \triangleleft m > 0 \triangleright \delta)\)  

**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** \(\text{in, out} : D\)

**proc** \(X = \sum_{d: D} \text{in}(d) \cdot (X || \text{out}(d))\)

**init** \(X\)
The hidden action $\tau$ represents an internal computation step.

It allows to abstract away from actions, typically so that only the input-output relation remains.

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

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

\[ a \cdot (b + \tau \cdot (b + c)) \not\sim a \cdot (\tau \cdot (b + c) + c) \]

\[ a \cdot (b + c) \not\sim a \cdot (\tau \cdot b + c) \]
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.
$s_1$ 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

\[ x \cdot \tau = x \]

\[ x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z) \]

\[ \tau_1(\delta) = \delta \]

\[ \tau_1(\tau) = \tau \]

\[ \tau_1(a(\vec{d})) = a(\vec{d}) \quad \text{if} \ a \not\in I \]

\[ \tau_1(a(\vec{d})) = \tau \quad \text{if} \ a \in I \]

\[ \tau_1(x + y) = \tau_1(x) + \tau_1(y) \]

\[ \tau_1(x \cdot y) = \tau_1(x) \cdot \tau_1(y) \]
Soundness and completeness of the axioms

All transition rules and a **sound and complete axiomatization** are given in the textbook.

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

$$ p = q \iff p \leftrightarrow_{rb} q $$
**Fair abstraction**

τ-loops can be eliminated.

**Example:** The process $X$ with

\[
X = a \cdot Y \\
Y = \tau \cdot Y + b
\]

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 $a \mid b = c$)
* basic operators: (+ ·)
* data-dependent operators: ($\langle b \rangle \sum_{d:D}$)
* process declaration: (proc $X(d_1, \ldots, d_n)$)
* parallel operators: ($\parallel \perp |$)
* 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

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

where the recursive equations for $X_1, \ldots, X_k$ use only data, actions, basic operators, and data-dependent operators.
Queues $B_1$ and $B_2$ of capacity one in sequence behave as a queue of capacity two:

\[
\text{proc } X = \sum_{d:D} r_1(d) \cdot Y(d) \\
Y(d:D) = \sum_{d':D} r_1(d') \cdot Z(d, d') + s_2(d) \cdot X \\
Z(d:D, d':D) = s_2(d) \cdot Y(d')
\]

\text{init } X
In the $\mu$CRL specification of two one-bit queues in sequence, the following data types are needed:

* **$\text{Bool}$** with constructors $T$ and $F$, and with mappings $\text{and}$, $\text{or}$, $\text{not}$, $\text{eq}$.

* **$D$** with constructors $d_1$ and $d_2$, and with mapping $\text{eq}$.

Furthermore, the following action declarations are needed:

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

* $r_3 \parallel s_3 = c_3$
Symbolic proof example - One-bit queues in sequence

\[ B_1 \parallel B_2 \]  

(Summations \( \Sigma_{d:D} \) and data parameters \( d \) are omitted)

\[ = B_1 \| B_2 + B_2 \| B_1 + B_1 | B_2 \]

\[ = (r_1 \cdot s_3 \cdot B_1) \| B_2 + (r_3 \cdot s_2 \cdot B_2) \| B_1 + (r_1 \cdot s_3 \cdot B_1) | (r_3 \cdot s_2 \cdot B_2) \]

\[ = r_1 \cdot ((s_3 \cdot B_1) \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel B_1) + \delta \cdot ((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) \]

\[ = r_1 \cdot ((s_3 \cdot B_1) \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel B_1) \]

\[ \partial_{\{s_3,r_3\}}(B_1 \parallel B_2) \]

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

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

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

\[ = r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2) \]
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 queues in sequence

$$\tau_{c_3}(\partial_{s_3,r_3}(B_1 \parallel B_2)) = \tau_{c_3}(r_1 \cdot \partial_{s_3,r_3}((s_3 \cdot B_1) \parallel B_2))$$

$$ = r_1 \cdot \tau_{c_3}(\partial_{s_3,r_3}((s_3 \cdot B_1) \parallel B_2))$$

$$ = r_1 \cdot \tau_{c_3}(c_3 \cdot \partial_{s_3,r_3}(B_1 \parallel (s_2 \cdot B_2)))$$

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

$$ = r_1 \cdot \tau_{c_3}(\partial_{s_3,r_3}(B_1 \parallel (s_2 \cdot B_2)))$$

Likewise we can derive:

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

$$ + r_1 \cdot \tau_{c_3}(\partial_{s_3,r_3}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)))$$

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

$$ = s_2 \cdot \tau_{c_3}(\partial_{s_3,r_3}(B_1 \parallel (s_2 \cdot B_2)))$$
The **renaming** operator $\rho_f$, for a mapping $f : \text{Act} \rightarrow \text{Act}$, renames all action names $a$ in their argument into $f(a)$.

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

\[
\begin{align*}
\rho_f(\delta) &= \delta \\
\rho_f(\tau) &= \tau \\
\rho_f(a(\vec{d})) &= f(a)(\vec{d}) \\
\rho_f(x + y) &= \rho_f(x) + \rho_f(y) \\
\rho_f(x \cdot y) &= \rho_f(x) \cdot \rho_f(y)
\end{align*}
\]
Four one-bit queues in sequence

\textbf{act} \quad r_1, s_2, r_3, s_3, c_3 : D

\textbf{comm} \quad r_3 \mid s_3 = c_3

\textbf{proc} \quad B_1 = \sum_{d:D} r_1(d) \cdot s_3(d) \cdot B_1
\quad B_2 = \sum_{d:D} r_3(d) \cdot s_2(d) \cdot B_2
\quad C = \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel B_2))

\textbf{act} \quad r_4, s_4, c_4 : D

\textbf{comm} \quad s_4 \mid r_4 = c_4

\textbf{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)))
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 space 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 state space explosion problem
- provides general correctness proof
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:

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

\[T_{db} = r_F(b) \cdot S_{1-b} \]
\[+ (r_F(1-b) + r_F(\perp)) \cdot s_B(d, b) \cdot T_{db}\]

\(R_b\) expects to receive a datum with \(b\) attached:

\[
R_b = \sum_{d:\Delta} r_C(d, b) \cdot s_D(d) \cdot s_E(b) \cdot R_{1-b}
\]
\[+ \sum_{d:\Delta} r_C(d, 1-b) \cdot s_E(1-b) \cdot R_b \]
\[+ r_C(\perp) \cdot s_E(1-b) \cdot R_b\]
$K$ and $L$ represent the corrupted channels, to model \textit{asynchronous} communication. (The action $j$ represents an internal choice.)

\begin{align*}
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
\end{align*}

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

\begin{align*}
 s_B | r_B &= c_B \\
 s_C | r_C &= c_C \\
 s_E | r_E &= c_E \\
 s_F | r_F &= c_F
\end{align*}
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 $j$)
Motivation for internal choice $j$

The internal choice $j$ is included in $K$ and $L$ because else deadlocks could disappear in the abstract $\mu$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, then the protocol specification would work correctly.

Pitfalls for formal modeling and verification:

- A formal system specification may be too abstract, or not in sync with the real system.
- Moreover, a set of formal requirements may be incomplete, or a formal requirement may not express what you think.
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 1st and 2nd assumption will be dropped in the forthcoming bounded retransmission protocol.

The 2nd and 3rd issue are resolved in the sliding window protocol.
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 \( \bot \) and with mapping eq
- **Bit** with constructors 0, 1 and with mappings invert, eq

\( \star \) \( r_A, s_D : \Delta \)
\( s_B, r_B, c_B, s_C, r_C, c_C : \Delta \times \text{Bit} \)
\( s_E, r_E, c_E, s_F, r_F, c_F : \text{Bit} \)
\( s_C, r_C, c_C, s_F, r_F, c_F : \text{Error} \)

\( j \)

\( \star \) \( r_B \mid s_B = c_B \), \( r_C \mid s_C = c_C \), \( r_E \mid s_E = c_E \), \( r_F \mid s_F = c_F \)
Patient support system

System states:
- calibrated versus uncalibrated
- docked versus undocked
- emergency
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. Specify the behavior of the system components in $\mu$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.
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 eventually the bed must go up.
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

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

First a \(\mu\text{CRL}\) specification is turned into an LPE.

Next a state space can be generated.
We distinguish two types of recursive equations \( X(x_1:D_1, \ldots, x_n:D_n) = p \).

I. \( p \) contains only \( + \cdot \triangleleft b \triangleright \sum_{d:D} \)

II. \( p \) may also contain \( \parallel \partial_H \tau_I \rho_f \)

The \( \mu \text{CRL lineariser} \) 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 recursive equation.

**Example:** \( X = a \cdot (X \parallel X) \) can’t be linearized.
Example: Four one-bit queues in sequence.

\[
\begin{align*}
\text{proc } B_1 &= \sum_{d:D} r_1(d) \cdot s_3(d) \cdot B_1 \\
B_2 &= \sum_{d:D} r_3(d) \cdot s_2(d) \cdot B_2 \\
C &= \tau_{\{c_3\}}(\partial_{\{s_3, r_3\}}(B_1 \parallel B_2))
\end{align*}
\]

\[
\begin{align*}
\text{init } \tau_{\{c_4\}}(\partial_{\{s_4, r_4\}}(\rho_{s_2 \rightarrow s_4}(C) \parallel \rho_{r_1 \rightarrow r_4}(C)))
\end{align*}
\]

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, by replacing them by the right-hand side of $C$’s recursive equation.
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.
First we explain, by an example, the linearization method for type I equations invoked by \texttt{mcrl}.

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

\( Y \) either performs \( k \) \( a \)'s, then a \( c \), and then \( k \) \( b \)'s, for any \( k \geq 0 \), or an infinite number of \( a \)'s.

**Step 1:** Make a \textit{Greibach normal form}.

\[
Y = a \cdot Y \cdot Z + c \\
Z = b
\]
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 $\text{in} : D \times \text{List} \rightarrow \text{List}$ are the constructors of List.

$\text{empty} : \text{List} \rightarrow \text{Bool}$ and $\text{head} : \text{List} \rightarrow D$ and $\text{tail} : \text{List} \rightarrow \text{List}$ are standard operations on lists.
Linearization of type I equations - Example

\[ Y = a \cdot Y \cdot Z + c \]
\[ Z = b \]

is transformed into

\[ X(\lambda:List) = a \cdot X(\text{in}(Y, \text{in}(Z, \text{tail}(\lambda)))) \triangleleft \text{eq}(\text{head}(\lambda), Y) \triangleright \delta \]
\[ + (c \triangleleft \text{empty}(\text{tail}(\lambda)) \triangleright c \cdot X(\text{tail}(\lambda))) \triangleleft \text{eq}(\text{head}(\lambda), Y) \triangleright \delta \]
\[ + (b \triangleleft \text{empty}(\text{tail}(\lambda)) \triangleright b \cdot X(\text{tail}(\lambda))) \triangleleft \text{eq}(\text{head}(\lambda), Z) \triangleright \delta \]

**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.
The stack employed in mcrl gives a lot of overhead.

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

For finite state spaces, mcrl -regular usually works much more efficiently than mcrl.

But mcrl -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 $X$.

\[
Y = a \cdot X \\
Z = b \cdot Z + b \\
X = Z \cdot Y
\]

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

\[
Y = a \cdot X \\
Z = b \cdot Z + b \\
X = b \cdot Z \cdot Y + b \cdot Y
\]

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

\[
Y = a \cdot X \\
Z = b \cdot Z + b \\
X = b \cdot X + b \cdot Y
\]
Linearization with pattern matching - Example 2

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

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

\[ X(n : \text{Nat}) = a(n) \cdot Y(n) \]
\[ Y(n : \text{Nat}) = b(S(n)) \cdot Z(n) \]
\[ Z(n : \text{Nat}) = c(S(S(n))) \cdot X(S(S(S(n)))) \]
mcrl -regular doesn’t always terminate.

Example:

\[
\begin{align*}
Y &= a \cdot Y \cdot b + c \\
Y &= a \cdot X_1 + c \\
X_1 &= Y \cdot b \\
Y &= a \cdot X_1 + c \\
X_1 &= a \cdot X_1 \cdot b + c \cdot b \\
X_2 &= X_1 \cdot b \\
Z_1 &= b \\
Y &= a \cdot X_1 + c \\
X_1 &= a \cdot X_2 + c \cdot Z_1 \\
X_2 &= a \cdot X_2 \cdot b + c \cdot Z_1 \cdot b \\
Z_1 &= b \\
\vdots
\end{align*}
\]
Linearization of type II equations - Example

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

**Example:** Let $a \parallel b = c$, 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:

\[
Y(m: \text{Nat}, n: \text{Nat}) = a(m) \cdot Y(S(m), n) \triangleleft m < 10 \triangleright \delta + b(m) \cdot Y(S(S(m)), n) \triangleleft m > 5 \triangleright \delta
\]

\[
+ a(n) \cdot Y(m, S(n)) \triangleleft n < 10 \triangleright \delta
\]

\[
+ b(n) \cdot Y(m, S(S(n))) \triangleleft n > 5 \triangleright \delta
\]

\[
+ c(m) \cdot Y(S(m), S(S(n))) \triangleleft m < 10 \land n > 5 \land \text{eq}(m, n) \triangleright \delta
\]

\[
+ c(n) \cdot Y(S(S(m)), S(n)) \triangleleft m > 5 \land n < 10 \land \text{eq}(m, n) \triangleright \delta
\]
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 $Closed = \emptyset$.

\[
\textbf{while } Open \neq \emptyset \textbf{ do} \\
\quad \text{select } d \in Open; \text{ } Open := \text{Open} \setminus \{d\}; \text{ } Closed := \text{Closed} \cup \{d\}
\]

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

\[
\textbf{if } d' \notin Open \cup Closed \textbf{ then } Open := Open \cup \{d'\}
\]

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

Check efficiently whether $d' \notin Open \cup Closed$. 
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”).
If a generated state $d'$ isn’t in the hash table, then the check $d' \notin \text{Open} \cup \text{Closed}$ requires an expensive disk lookup.

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

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

A Bloom filter uses 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 $h_1(d), \ldots, h_k(d)$ are set to 1.
Suppose a state \( d' \) is generated, and doesn’t occur at entry \( h(d') \) in the hash table.

Then we check if positions \( h_i(d') \) for \( i = 1, \ldots, k \) in the Bloom filter all contain 1.

If not, then \( d' \notin \text{Open} \cup \text{Closed} \).

Else, still an expensive disk lookup is required.
When \( n \) elements have been inserted in \( \textit{Open} \cup \textit{Closed} \), the possibility that a certain position in a Bloom filter of length \( m \) with \( k \) hash functions contains 0 is

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

So the probability that \( k \) different 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 \( k = 4 \).)
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 space explosion

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

Empty list \( [] \) and \( \text{in} : D \times \text{List} \rightarrow \text{List} \) are the constructors of the data type \( \text{List} \).

Impose a total order \( \prec \) on the data type \( D \).

\( \text{add} : D \times \text{List} \rightarrow \text{List} \) produces sorted lists:

\[
\text{add}(d, []) = \text{in}(d, []),
\]

\[
\text{add}(d, \text{in}(e, \ell)) = \text{if}(d \prec e, \text{in}(d, \text{in}(e, \ell)), \text{in}(e, \text{add}(d, \ell)))
\]

where \( \text{if} : \text{Bool} \times \text{List} \times \text{List} \rightarrow \text{List} \) acts as “if-then-else”:

\[
\text{if}(T, \ell_1, \ell_2) = \ell_1 \quad \text{if}(F, \ell_1, \ell_2) = \ell_2
\]
Assume a finite state space.
(For simplicity we disregard termination states.)

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 $a \in \text{Act} \cup \{\tau\}$, $s_0 \in \text{split}_a(P_i, P_j)$ 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 $s \in \text{split}_a(P_i, P_j)$ 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 $(\ast)$ is trivially satisfied.)

Suppose that at some point $S$ is partitioned into $P_1 \cup \cdots \cup P_k$. If $\emptyset \subset split_a(P_i, P_j) \subset P_i$ with $a \neq \tau$ or $i \neq j$, then in the partition, $P_i$ can be replaced by

$$split_a(P_i, P_j) \text{ and } P_i \setminus split_a(P_i, P_j).$$

Splitting continues until no further split is possible.

$(\ast)$ 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 \( P_1 \cup \cdots \cup P_k \) 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.

\( P_i \xrightarrow{\alpha} P_j \) if \( s \xrightarrow{\alpha} s' \) for some \( s \in P_i \) and \( s' \in P_j \), with \( \alpha \neq \tau \) 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.

\[ P \]

\[ s_0 \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ P_1 \]

\[ s_0 \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ P_2 \]

\[ s_0 \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ P_{21} \]

\[ s_0 \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ P_{22} \]

\(split_a(P, P)\) 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

\[ \{s_0\} \]

\[ \{s_1, s_3\} \]

\[ \{s_2\} \]
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.
States, i.e. constructor data terms, are represented compactly as ATerms, consisting of a head symbol and pointers to 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.
LPEs are stored using ATerms.

**Example:** The process term $X(S(S(0)), S(0))$ is stored as follows.
Distributed state space generation - Hash function

The state space can be stored on a cluster of computers (e.g. DAS-5).

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.
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.
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 $(d_1, \ldots, d_k)$ becomes a list of indices $(i_1, \ldots, i_k)$.

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

The database usually remains small, compared to the state space, because relatively few data terms are used in states.
To compute the successors of a state \((i_1, \ldots, i_k)\), it must be expanded into \((d_1, \ldots, d_k)\).

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

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

The hash value of \((j_1, \ldots, j_k)\) is computed, and this list is sent to the responsible processor.

The state space generator must continuously consult the database, to move between the two representations of states.
(Part of) the central database is replicated at all processors.

Each local database at a processor \( P \) always contains all indices from 0 up to some \( \text{max}_P \).

If some generated data terms are absent from \( P \)’s database, it sends these terms together with \( \text{max}_P \) 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 \( \text{max}_P + 1 \) up to \( \text{max} \) 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.
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**.
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.
A processor \( P \) first stores the state 01100100.
$P$ next stores the state 01000101.
$P$ next stores the state 01001100.
$P$ 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.
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.
We define some basic modal logic operators to express properties of states.

\[ \phi ::= T \mid F \mid \phi \land \phi' \mid \phi \lor \phi' \mid \langle a \rangle \phi \mid [a] \phi \]

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

- \( T \) holds in all states, and \( F \) in no state
- \( \land \) denotes conjunction, and \( \lor \) 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 \( s' \)
The states \( s \) that satisfy a formula \( \phi \), denoted \( s \models \phi \), are defined inductively by:

\[
\begin{align*}
    s &\models T \\
    s &\not\models F \\
    s &\models \phi \land \phi' \quad \text{if } s \models \phi \text{ and } s \models \phi' \\
    s &\models \phi \lor \phi' \quad \text{if } s \models \phi \text{ or } s \models \phi' \\
    s &\models \langle a \rangle \phi \quad \text{if for some state } s', s \xrightarrow{a} s' \text{ with } s' \models \phi \\
    s &\models [a] \phi \quad \text{if for all states } s', s \xrightarrow{a} s' \text{ implies } s' \models \phi
\end{align*}
\]

**Example:** If \( s \xrightarrow{a} \), then \( s \models [a] F \) and \( s \not\models \langle a \rangle T \).
Fixpoints

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

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

d $\in D$ is a fixpoint of $S : D \to D$ 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.

$$\phi ::= T \mid F \mid \phi \land \phi' \mid \phi \lor \phi' \mid \langle a \rangle \phi \mid [a] \phi \mid X \mid \mu X. \phi \mid \nu X. \phi$$

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

Example:

$$\mu X. \neg \langle a \rangle X$$ has no fixpoint.
\( \mu X. (\langle a \rangle X \lor \langle b \rangle T) \) represents those states that can execute \( a^k b \) for some \( k \geq 0 \).

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

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

**Question:** How about \( \mu X. (\langle a \rangle X \lor \langle b \rangle X) \)?
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 $\nu X.\langle b \rangle (\mu Y.(\langle a \rangle X \lor \langle a \rangle Y))$.

$$
\begin{array}{c|c|c}
 & Y & X \\
\hline
\emptyset & \emptyset & \{s\} \\
\{s\} & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\end{array}
$$

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.
For two nested minimal (or maximal) fixpoints, recomputing a fixpoint isn’t 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 \(\nu X. \nu Y. (\langle a \rangle X \lor \langle b \rangle Y)\).

\[
\begin{array}{c|c}
Y & X \\
\{s_0, \ldots, s_{2n}\} & \{s_0, \ldots, s_{2n}\} \\
\{s_0, \ldots, s_{2n-1}\} & \{s_0, \ldots, s_{2n-2}\} \\
\{s_0, \ldots, s_{2n-3}\} & \{s_0, \ldots, s_{2n-4}\} \\
\vdots & \vdots \\
\emptyset & \emptyset \\
\end{array}
\]

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 $\text{NP} \cap \text{co-NP}$.

It is an open question whether it is in $\text{P}$. 
Regular $\mu$-calculus

$$\alpha ::= T \mid a \mid \neg \alpha \mid \alpha \land \alpha' \quad (a \in \text{Act} \cup \{\tau\})$$

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

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

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

$$\phi ::= T \mid F \mid \phi \land \phi' \mid \phi \lor \phi' \mid \langle \beta \rangle \phi \mid [\beta] \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$
Regular $\mu$-calculus formulas are transformed into $\mu$-calculus formulas.

\[
\langle \alpha \rangle \phi = \bigvee_{a \in \alpha} \langle a \rangle \phi \quad \quad [\alpha] \phi = \bigwedge_{a \in \alpha} [a] \phi
\]

\[
\langle \beta \cdot \beta' \rangle \phi = \langle \beta \rangle \langle \beta' \rangle \phi \quad \quad [\beta \cdot \beta'] \phi = [\beta] [\beta'] \phi
\]

\[
\langle \beta | \beta' \rangle \phi = \langle \beta \rangle \phi \lor \langle \beta' \rangle \phi \quad \quad [\beta | \beta'] \phi = [\beta] \phi \land [\beta'] \phi
\]

\[
\langle \beta^* \rangle \phi = \mu X.(\phi \lor \langle \beta \rangle X) \quad \quad [\beta^*] \phi = \nu X.(\phi \land [\beta] X)
\]

Formulas $\nu Y.(\langle \beta^* \rangle \phi(Y))$ and $\mu Y.([\beta^*] \phi(Y))$ aren’t alternation-free.
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:

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

Each send is eventually followed by a read:

$$[T^\ast \cdot send] \mu X. (\langle T \rangle T \land [\neg read] X)$$
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 analyze the error trace, omit the hiding operator from the initial state before state space generation.
CADP syntax for formulas

**Examples:**

\[(\text{not } "\text{send}(d)"*)\text{."recv}(d)"] \text{ false}

\[(\text{not } '\text{send}(\text{.*})')*.'\text{recv}(\text{.*})'] \text{ false}

\[(\text{true})*."\text{send}(d)"] \mu X. (\langle\text{true}\rangle \text{ true and [not } "\text{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.
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

\[ + \ test(x_1, x_3) \cdot X(x_1, x_2, x_3) \]

where \( test \) is a “fresh” action name.

Such a self-loop doesn’t increase the number of reachable states. But it can influence the validity of properties that ignore fairness.
Data packets are sent from RC to TV.

The last datum of a packet is labeled.

An alternating bit is attached to each datum.

A datum may get lost.

Only one kind of ack.

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

If RC doesn’t receive an ack 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.)

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

\[
Y(\lambda : \Lambda, b : \text{Bit}, n : \text{Nat}) = (s_B(\text{head}(\lambda), b) \triangleleft \text{length}(\lambda) > S(0) \triangleright s_B(\text{head}(\lambda), b, \text{last})) \cdot Z(\lambda, b, n)
\]

\[
Z(\lambda : \Lambda, b : \text{Bit}, n : \text{Nat}) = r_F(\text{ack}) \cdot (Y(\text{tail}(\lambda), 1 - b, S(0)) \triangleleft \text{length}(\lambda) > S(0) \triangleright s_A(l_{OK}) \cdot X) + r_G(\text{to}) \cdot (Y(\lambda, b, S(n)) \triangleleft n < \text{max} \triangleright (s_A(l_{NOK}) \triangleleft \text{length}(\lambda) > S(0) \triangleright s_A(l_{DK})) \cdot s_H(\text{to}) \cdot X)
\]
Bounded retransmission protocol - Television

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

\[ W(b: \text{Bit}) = \sum_{d: \Delta} r_C(d, b) \cdot s_D(d) \cdot s_E(\text{ack}) \cdot W(1-b) \]
\[ + \sum_{d: \Delta} r_C(d, b, \text{last}) \cdot s_D(d, \text{last}) \cdot s_E(\text{ack}) \cdot V \]
\[ + \sum_{d: \Delta} r_C(d, 1-b) \cdot s_E(\text{ack}) \cdot W(b) \]
\[ + r_H(\text{to}) \cdot V \]
\[ K = \sum_{d: \Delta} \sum_{b: \{0, 1\}} \left( r_B(d, b) \cdot (j \cdot s_C(d, b) + j \cdot s_G(to)) \right) \cdot K \\
+ \ r_B(d, b, last) \cdot (j \cdot s_C(d, b, last) + j \cdot s_G(to)) \cdot K \] \\

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

The state space is minimized modulo \( \leftrightarrow_b \).
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

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

for all \( d \in D \), where the LPE

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

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, $\tau \cdot a \not\leftrightarrow_r b \tau \cdot b$. 
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 \( m \in \text{Nat} \).
A boolean property $\mathcal{I}$ on the states in a state space is an invariant if

$$(\mathcal{I}(s) \land s \xrightarrow{a} s') \Rightarrow \mathcal{I}(s')$$

If $\mathcal{I}(s)$, then from $s$ one can only reach states $s'$ with $\mathcal{I}(s')$.

**Question**: Give a state space and a boolean property $P$ on its states such that $P$ holds for all states reachable from the initial state, but it isn't an invariant.
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 \( e \in E \)

\[ \mathcal{I}(d) \land h_i(d, e) \Rightarrow \mathcal{I}(g_i(d, e)) \]

An invariant is satisfied by all the reachable states of the LPE in question, given an initial state \( d_0 \) with \( \mathcal{I}(d_0) \).
Invariants

Note that the invariant property must be checked on all states (including the unreachable ones).

Questions:

▶ Is $I(d) = T$ for all $d \in D$ always an invariant?

▶ Is $I(d) = F$ for all $d \in D$ always an invariant?
Let $X(n:\text{Nat}) = a(n) \cdot X(S(S(n)))$.

Invariants for this LPE are

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

and

$I(n) = \begin{cases} 
F & \text{if } n \text{ is even} \\
T & \text{if } n \text{ is odd} 
\end{cases}
$
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 $I(d)$, with $I$ 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 $I(d)$. 
even : \textit{Nat} \to \textit{Bool} is defined by even(0) = T, even(S(n)) = \neg even(n).

\[ X(n: \textit{Nat}) = a(even(n)) \cdot X(S(S(n))) \]

\[ Y = a(T) \cdot Y \]

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

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

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

\[ X(n) = Y \quad \text{for even numbers } n. \]
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.

--- External actions

--- Progressing internal actions
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 all states $s$ and $s'$ in $G_1$ and $a \neq \tau$:

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

If $\phi$ satisfies the matching criteria, then it establishes a branching bisimulation relation.
Cones and foci - Example

\[ G_1 \quad \phi \quad G_2 \]

\[ a \quad c \]

\[ \tau \]

\[ a \quad a \quad c \]
It is essential that each state belongs to the cone of a focus point.

Example:

\[
\xymatrix{
  & \tau \\
\ldots & \ar[r] & \ar[d] & a \\
  \ldots & & & \ldots
}\]
Assume a convergent LPE (of the original $\mu$CRL specification)

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

and an LPE $Z$ (of its external behavior) without $\tau$’s:

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

$\phi : D \rightarrow D'$ satisfies the matching criteria at some $d \in D$ if for all $b \in \text{Act}$ and $e \in E$:

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

where the focus condition $FC_X(d)$ denotes $\forall e:E \neg h_\tau(d, e)$.
Let $X(d:D)$ be a convergent LPE, and $Z(d':D')$ an LPE without $\tau$’s.

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

If $\phi: D \rightarrow D'$ satisfies the matching criteria at all $d \in D$ with $I(d)$
then for all $d \in D$ with $I(d)$

$$X(d) \leftrightarrow_b Z(\phi(d))$$

If moreover $FC_X(d)$, then

$$X(d) \leftrightarrow_{rb} Z(\phi(d))$$
Cones and foci - Example

\[ \text{State} = \{1, 2, 3\} \]

\[
X(k:\text{State}) = \tau \cdot X(2) \triangleleft k = 1 \triangleright \delta \\
+ a \cdot X(3) \triangleleft k = 1 \lor k = 2 \triangleright \delta \\
+ c \cdot X(3) \triangleleft k = 2 \triangleright \delta
\]

\[
Z(b:\text{Bool}) = a \cdot Z(F) \triangleleft b \triangleright \delta \\
+ c \cdot Z(F) \triangleleft b \triangleright \delta
\]

Questions: What is the focus condition?  
How is the state mapping defined?  
What are the matching criteria?  
Which equivalences between states have been proved?
Focus condition $FC_X(k)$:  $k \neq 1$

State mapping:

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

Matching criteria:

- $k = 1 \Rightarrow \phi(k)$
- $k = 1 \lor k = 2 \Rightarrow \phi(k)$
- $k = 2 \Rightarrow \phi(k)$
- $k = 1 \lor k = 2 \Rightarrow \neg \phi(3)$
- $k = 2 \Rightarrow \neg \phi(3)$
- $k \neq 1 \land \phi(k) \Rightarrow k = 1 \lor k = 2$
- $k \neq 1 \land \phi(k) \Rightarrow k = 2$
IEEE 1394

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

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

The network size is unknown to the nodes.
IEEE 1394 leader election algorithm

We determine a root in the network, and parent-child relations.

A node can send a parent request to a neighbor, as soon as this is the only possible parent.

A node without possible parents becomes the leader (i.e., root).

We assume that communication is synchronous.

\[ X(i: \text{Node}, \ p: \text{Nodelist}, \ s: \text{State}) \] represents node \( i \) in state \( s \) with possible parents \( p \).

- In state 0 a node is looking for a parent.
- In state 1 a node has a parent, or is the root.
IEEE 1394 leader election algorithm

\[ X(i: \text{Node}, p: \text{Nodelist}, s: \text{State}) = \sum_{j: \text{Node}} r(j, i) \cdot X(i, p\setminus\{j\}, s) \bowtie T \triangleright \delta \]
\[ + \sum_{j: \text{Node}} s(i, j) \cdot X(i, p, 1) \bowtie p = \{j\} \land s = 0 \triangleright \delta \]
\[ + \text{leader}(i) \cdot X(i, p, 1) \bowtie \text{empty}(p) \land s = 0 \triangleright \delta \]

Example:

This network is captured by

\[ \tau_I(\partial_H(X(i_0, \{i_1, i_2, i_3\}, 0) \parallel X(i_1, \{i_0\}, 0) \parallel X(i_2, \{i_0\}, 0) \parallel X(i_3, \{i_0\}, 0))) \]
The external behavior of the network on the previous slide:

This state space is minimal modulo $\leftrightarrow_b$. 
A non-connected network will produce multiple roots.

A network with a cycle won’t produce a root.

Example:

\[
\begin{array}{c}
\text{i}_2 \\
\text{i}_1 \quad \text{i}_0
\end{array}
\]

In IEEE 1394, cycles are detected using a *time-out*.

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

**Goal:** Prove that one root is selected, i.e., the external behavior (after minimization modulo $\leftrightarrow_b$) is $\text{leader} \cdot \delta$.

We specify the node processes without parametrization of $\text{leader}$:

$$X(i: \text{Node}, p: \text{Nodelist}, s: \text{State})$$

$$= \sum_{j: \text{Node}} r(j, i) \cdot X(i, p\{j\}, s) \triangleleft T \triangleright \delta$$

$$+ \sum_{j: \text{Node}} s(i, j) \cdot X(i, p, 1) \triangleleft p = \{j\} \land s = 0 \triangleright \delta$$

$$+ \text{leader} \cdot X(i, p, 1) \triangleleft \text{empty}(p) \land s = 0 \triangleright \delta$$

The initial state is specified by

$$\tau_I(\partial_H(X(i_0, p_0[i_0], 0) \parallel \cdots \parallel X(i_k, p_0[i_k], 0)))$$

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


Linearization

\[
X(i: \text{Node}, p: \text{Nodelist}, s: \text{State}) = \sum_{j: \text{Node}} r(j, i) \cdot X(i, p \setminus \{j\}, s) \triangle T \triangleright \delta \\
+ \sum_{j: \text{Node}} s(i, j) \cdot X(i, p, 1) \triangle p = \{j\} \land s = 0 \triangleright \delta \\
+ \text{leader} \cdot X(i, p, 1) \triangle \text{empty}(p) \land s = 0 \triangleright \delta
\]

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 an LPE \( Y(p, s) \), defined by

\[
Y(p: \text{Nodelistlist}, s: \text{Statelist})
= \sum_{i, j: \text{Node}} \tau \cdot Y(p[i] := p[i] \setminus \{j\}, s[j] := 1) \triangle p[j] = \{i\} \land s[j] = 0 \triangleright \delta \\
+ \sum_{i: \text{Node}} \text{leader} \cdot Y(p, s[i] := 1) \triangle \text{empty}(p[i]) \land s[i] = 0 \triangleright \delta
\]

\( Y(p, s) \) is convergent, because each execution of \( \tau \) reduces the number of nodes \( j \) with \( s[j] = 0 \).
Focus points of $Y$ are states $(p, s)$ that satisfy:

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

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

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

\[ Y(p:\text{Nodelist}, s:\text{Statelist}) \]
\[ = \sum_{i,j: \text{Node}} \tau \cdot Y(p[i] := p[i] \setminus \{j\}, s[j] := 1) \land p[j] = \{i\} \land s[j] = 0 \triangleq \delta \]
\[ + \sum_{i: \text{Node}} \text{leader} \cdot Y(p, s[i] := 1) \land \text{empty}(p[i]) \land s[i] = 0 \triangleq \delta \]

Matching criteria

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

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

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

\( (\forall i, j: \text{Node} (p[j] \neq \{i\} \lor s[j] = 1) \land \phi(p, s)) \Rightarrow \exists i': \text{Node} (\text{empty}(p[i']) \land s[i'] = 0) \)
Invariants

We need some *invariants* for the LPE $\mathcal{Y}$.

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

If a transition removes $j$ from $p[i]$, then the action is $\tau$, and in the resulting state $p[j] = \{i\}$, so in particular $i \in p[j]$.

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

$$\mathcal{I}_2 \quad \forall i, j: \text{Node} \ (j \not\in p[i] \land i \in p[j]) \Rightarrow s[j] = 1$$

If $i \not\in p[j]$ or $s[j] = 1$, then after a transition this still holds.

Suppose $j \in p[i]$ and $s[j] = 0$, while after a transition $j \not\in p[i]$. Then this action was $\tau$, and in the resulting state $s[j] = 1$.

$\mathcal{I}_2$ holds in the initial state of $\mathcal{Y}$. 
Invariants

\[ I_3 \quad \forall j: \text{Node} \ (s[j] = 1 \Rightarrow (\text{empty}(p[j]) \lor \text{singleton}(p[j]))) \]

\(|p[j]| \leq 1\) is preserved by transitions.

If \(s[j] = 0\) and \(|p[j]| > 1\), then after any transition still \(s[j] = 0\).

\(I_3\) holds in the initial state of \(Y\).

\[ I_4 \quad \forall i, j: \text{Node} \ ((j \in p[i] \land s[i] = 0) \Rightarrow (i \in p[j] \land s[j] = 0)) \]

The proof that \(I_4\) is an invariant is left as an exercise.

\(I_4\) holds in the initial state of \(Y\).
Validity of the matching criteria

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

By \( I_4 \), \( s[i] = 0 \).

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

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

\( \phi(p, s) \), because \( s[i] = 0 \).
One more invariant: Uniqueness of the root

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

**Proof:** By *connectedness*, there are distinct nodes $i = i_0, i_1, \ldots, i_m = j$ with $i_{k+1} \in p_0[i_k]$ for $k = 0, \ldots, m - 1$.

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

$k = 0:$ \[ I_1 \Rightarrow (i_1 \in p[i_0] \lor i_0 \in p[i_1]) \]
\[ \text{empty}(p[i_0]) \Rightarrow (i_1 \not\in p[i_0] \land i_0 \in p[i_1]) \]
\[ I_2 \Rightarrow s[i_1] = 1 \]
\[ I_3 \land i_0 \in p[i_1] \Rightarrow p[i_1] = \{i_0\} \]

$k \geq 1:$ \[ I_1 \Rightarrow (i_{k+1} \in p[i_k] \lor i_k \in p[i_{k+1}]) \]
\[ p[i_k] = \{i_{k-1}\} \Rightarrow (i_{k+1} \not\in p[i_k] \land i_k \in p[i_{k+1}]) \]
\[ I_2 \Rightarrow s[i_{k+1}] = 1 \]
\[ I_3 \land i_k \in p[i_{k+1}] \Rightarrow p[i_{k+1}] = \{i_k\} \]

Hence, $s[i_m] = 1$ and $p[i_m] = \{i_{m-1}\}$. 
∀i: Node ((empty(p[i]) ∧ s[i] = 0) ⇒ ¬φ(p, s[i] := 1))

By uniqueness of the root, empty(p[i]) implies s[j] = 1 for each j ≠ i.

Hence, ¬φ(p, s[i] := 1).
(∀i, j: Node (p[j] ≠ {i} ∨ s[j] = 1) ∧ φ(p, s)) ⇒ ∃i′: Node (empty(p[i′]) ∧ s[i′] = 0)

Since φ(p, s), there is a node i′ with s[i′] = 0.

Suppose p[i′] isn’t empty; we derive a contradiction.

Let j ∈ p[i′] for some j. By I₄, i′ ∈ p[j] and s[j] = 0.
So by the condition of the matching criterion, p[j] ≠ {i′}.
Then there is a k ≠ i′ with k ∈ p[j].

By I₄, j ∈ p[k] and s[k] = 0.
So by the condition of the matching criterion, p[k] ≠ {j}.
Then there is an ℓ ≠ j with ℓ ∈ p[k].

Et cetera. This contradicts the fact that there is no cycle.
If $p_0$ establishes a connected network without cycles, and $s_0[i] = 0$ for all nodes $i$, then

$$Y(p_0, s_0) \leftrightarrow_b Z(\phi(p_0, s_0))$$

Hence,

$$Y(p_0, s_0) \leftrightarrow_b leader \cdot \delta$$

This implies

$$\tau_I(\partial_H(X(i_0, p_0[i_0], 0) \parallel \cdots \parallel X(i_k, p_0[i_k], 0))) \leftrightarrow_b leader \cdot \delta$$

So in a finite network that is connected and free of cycles, one leader will be elected.
Now suppose communication is \textit{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 \textit{root contention}.

An \textit{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

- Node 0:
  - Send req (only one neighbour)
  - Receive req

- Node 1:
  - Send ack (one remaining req)
  - Receive req

- Node 1’:
  - Send req (no remaining ack)
  - Receive req
  - Send ack

- Node 2:
  - Receive ack

- Node 3:
  - Receive req

- Node 4:
  - Become leader (no remaining ack)

- Node 0 becomes the leader after receiving the final req.
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 $G$,

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

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

It makes sense to apply confluence reduction repeatedly.
Theorem: If $T$ is confluent and $s \xrightarrow{\tau} T s'$, then $s \leftrightarrow_b s'$.

Let the state space $G$ be free of $\tau$-loops.

If $s \xrightarrow{\tau} T s'$, then

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

**Example:**

\[
\tau \quad \quad \quad \quad a
\]

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 (SCCs).
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 SCCs 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 $G$, to obtain a directed graph $G^R$. Apply depth-first search to $G^R$; each new exploration starts in the unvisited node with the highest $F$-value.

Each exploration in $G^R$ determines an SCC in $G$. 
Kosaraju’s algorithm - Example

or

$F$-values of the 1st and $D$-values of the 2nd depth-first search are given.
Kosaraju’s algorithm - Correctness & complexity

**Correctness:** Clearly, nodes in the same SCC are discovered during the same subcall of depth-first search on $G^R$.

Let node $v$ be discovered during the subcall of depth-first search on $G^R$ started at node $u$.

Then $F_u \geq F_v$, and there is a path from $v$ to $u$ in $G$.

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

- If depth-first search on $G$ visited $v$ before $u$, then $F_v > F_u$ (because there is a path from $v$ to $u$).
- If depth-first search on $G$ visited $u$ before $v$, then $F_v > F_u$ (because there is no path from $u$ to $v$).

This contradicts the fact that $F_u \geq F_v$.

Hence $v$ is in the same SCC as $u$.

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

**Step 1:** Compute the SCCs consisting of $\tau$-transitions, and collapse each of them to a single state.

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

**Step 3:** While $S \neq \emptyset$, take an $s \xrightarrow{a} s'$ from $S$.
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 $S$.
(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 $S = \emptyset$, then output $T$.

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

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

Assume an LPE

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

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

\[ d \xrightarrow{\tau} g_i(d, e_1) \]

\[ a_j(f_j(d, e_2)) \]

\[ g_j(d, e_2) \xrightarrow{\tau} g_j(d, e_2) \]

or \( a_j = \tau \) and \( g_i(d, e_1) = g_j(d, e_2) \).
Let \( a_i = \tau \). If \( h_i(d, e_1) \land h_j(d, e_2) \) always implies

\[
\begin{align*}
h_j(g_i(d, e_1), e_2) & \land h_i(g_j(d, e_2), e_1) \\
\land f_j(d, e_2) & = f_j(g_i(d, e_1), e_2) \\
\land g_j(g_i(d, e_1), e_2) & = g_i(g_j(d, e_2), e_1) \\
\lor a_j & = \tau \land g_i(d, e_1) = g_j(d, e_2)
\end{align*}
\]

then summand \( i \) of the LPE is confluent.
Two unbounded queues

\[
B_1(\lambda_1: \text{List}) = \sum_{d: \Delta} r_1(d) \cdot B_1(\text{in}(d, \lambda_1)) \\
+ s_3(\text{toe}(\lambda_1)) \cdot B_1(\text{untoe}(\lambda_1)) \triangleleft \neg \text{empty}(\lambda_1) \triangleright \delta
\]

\[
B_2(\lambda_2: \text{List}) = \sum_{d: \Delta} r_3(d) \cdot B_2(\text{in}(d, \lambda_2)) \\
+ s_2(\text{toe}(\lambda_2)) \cdot B_2(\text{untoe}(\lambda_2)) \triangleleft \neg \text{empty}(\lambda_2) \triangleright \delta
\]

The initial state is \( \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1(\text{[]}) \parallel B_2(\text{[]}))) \).

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  [] :→ List
       in : Δ × List → List

map  empty : List → Bool
      if : Bool × List × List → List
      toe : List → Δ
      untoe : List → List

var  d : Δ
     λ, λ₁, λ₂ : List

rew  empty([]) = T
     empty(in(d, λ)) = F
     if (T, λ₁, λ₂) = λ₁
     if (F, λ₁, λ₂) = λ₂
     toe(in(d, λ)) = if (¬empty(λ), toe(λ), d)
     untoe(in(d, λ)) = if (¬empty(λ), in(d, untoe(λ)), [])
Two unbounded queues - Confluence formulas

\[\tau \{c_3\} (\partial \{s_3, r_3\} (B_1(\lambda_1) \parallel B_2(\lambda_2)))\] linearizes to

\[X(\lambda_1:\text{List}, \lambda_2:\text{List}) = \sum_{d:\Delta} r_1(d) \cdot X(in(d, \lambda_1), \lambda_2) + \tau \cdot X(\text{untoe}(\lambda_1), in(\text{toe}(\lambda_1), \lambda_2)) \triangleleft \neg \text{empty}(\lambda_1) \triangleright \delta + s_2(\text{toe}(\lambda_2)) \cdot X(\lambda_1, \text{untoe}(\lambda_2)) \triangleleft \neg \text{empty}(\lambda_2) \triangleright \delta\]

We compute the confluence formulas.

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

\[-\text{empty}(\lambda_1) \implies \neg \text{empty}(in(d, \lambda_1))\]
\[\land in(d, \text{untoe}(\lambda_1)) = \text{untoe}(in(d, \lambda_1))\]
\[\land in(\text{toe}(\lambda_1), \lambda_2) = in(\text{toe}(in(d, \lambda_1)), \lambda_2)\]

(2) Commutation of \(\tau\) and \(s_2(\text{toe}(\lambda_2))\):

\[-\text{empty}(\lambda_1) \land \neg \text{empty}(\lambda_2) \implies \neg \text{empty}(in(\text{toe}(\lambda_1), \lambda_2))\]
\[\land \text{toe}(in(\text{toe}(\lambda_1), \lambda_2)) = \text{toe}(\lambda_2)\]
\[\land \text{untoe}(in(\text{toe}(\lambda_1), \lambda_2)) = in(\text{toe}(\lambda_1), \text{untoe}(\lambda_2))\]
If a $\tau$-summand in a convergent LPE is confluent, then it can be given priority over the other (non-confluent) 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

\[
X(\lambda_1:List, \lambda_2:List)
= \sum_{d:\Delta} r_1(d) \cdot X(in(d, \lambda_1), \lambda_2) \triangleleft empty(\lambda_1) \triangleright \delta
+ \tau \cdot X(untoe(\lambda_1), in(toe(\lambda_1), \lambda_2)) \triangleleft \neg empty(\lambda_1) \triangleright \delta
+ s_2(toe(\lambda_2)) \cdot X(\lambda_1, untoe(\lambda_2)) \triangleleft empty(\lambda_1) \land \neg empty(\lambda_2) \triangleright \delta
\]

So if $\lambda_1$ isn’t empty, then only the $\tau$-summand can be executed.
Consider a \textit{finite} state space, possibly containing $\tau$-loops, with a confluent set of $\tau$-transitions.

For each reachable state $d$ we compute a \textit{representative} state $\text{repr}(d)$, such that:

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

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 \textit{terminal} SCC of confluent $\tau$-transitions.

In the first case the \textit{representative} is returned, and in the second case the \textit{least state} in the terminal SCC (assuming a total order on states).
State space generation modulo confluence

Consider a (possibly non-convergent) LPE that gives rise to a finite state space and has confluent \( \tau \)-summands.

During state space generation, only representatives of states need to be generated, which are computed as before.

State space generation starts from \( \text{repr}(d_0) \), with \( d_0 \) the initial state.

If the representative of a state \( d \) is added to the state space, then for each 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 \( \text{repr}(d) = \text{repr}(d') \)).
<table>
<thead>
<tr>
<th>System</th>
<th>Standard State Space</th>
<th>Reduced State Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>states</td>
<td>transitions</td>
</tr>
<tr>
<td>abp(2)</td>
<td>97</td>
<td>122</td>
</tr>
<tr>
<td>brp(2)</td>
<td>1,952</td>
<td>2,387</td>
</tr>
<tr>
<td>tip(10)</td>
<td>72,020</td>
<td>389,460</td>
</tr>
<tr>
<td>tip(12)</td>
<td>446,648</td>
<td>2,853,960</td>
</tr>
<tr>
<td>tip(14)</td>
<td>2,416,632</td>
<td>17,605,592</td>
</tr>
</tbody>
</table>
\textbf{confcheck -mark file.tbf}

renames confluent tau-summands in the input file file.tbf into ctau. The flag \texttt{-invariant} allows to attach an invariant.

\textbf{confelm file.tbf}

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

\textbf{instantiator -confluent ctau file.tbf}

generates the state space from file.tbf, while prioritizing ctau-transitions.
A theorem prover within the μCRL 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.
**Simplification of LPEs**

*rewr* rewrites the data terms in the LPE using their equations.

If a condition in the LPE rewrites to F, the corresponding summand is removed.

*constelm*, *parelm* and *sumelm* simplify an LPE by eliminating data parameters that do not influence its behavior.

These static analysis tools are remarkably useful, because linearisation may introduce redundant data parameters.
Consider an LPE, with initial state \((d_1, \ldots, d_m)\), of the form

\[
X(x_1 : D_1, \ldots, x_m : D_m) = \sum_{i=1}^{k} \sum_{y_1^i : E_1^i} \cdots \sum_{y_n^i : E_n^i} a_i(f_i(\bar{x}, \bar{y}^i)) \cdot \nabla X(g_1^i(\bar{x}, \bar{y}^i), \ldots, g_m^i(\bar{x}, \bar{y}^i)) \triangleleft h_i(\bar{x}, \bar{y}^i) \triangleright \delta
\]

Initially \(j\) is unmarked for all \(j = 1, \ldots, m\).

Check for each unmarked \(j\) and summand \(i\) whether

\[
(\bigwedge_{j \text{ unmarked}} eq(x_j, d_j) \land h_i(\bar{x}, \bar{y}^i)) \Rightarrow eq(g_j^i(\bar{x}, \bar{y}^i), d_j)
\]

rewrites to \(T\) (using \(\text{rewr}\)).

If this check fails (for some \(i\)), then \(j\) is marked.

The check must then be repeated for all unmarked indices \(j\).

When the check succeeds for all unmarked \(j\) and all summands \(i\), these \(x_j\) are replaced by \(d_j\) and eliminated as argument from \(X\).
Initially, all \( j \) are unmarked.

An unmarked \( j \) is marked if \( x_j \) occurs in an

\[ h_i(\vec{x}, \vec{y}^i), \]
\[ f_i(\vec{x}, \vec{y}^i), \text{ or} \]
\[ g^i_k(\vec{x}, \vec{y}^i) \text{ with } k \text{ marked.} \]

When no further \( j \) can be marked, the unmarked \( x_j \) are eliminated as argument from \( X \).

This removes all occurrences of unmarked \( x_j \) from the LPE.

\texttt{parelm} may potentially reduce an infinite to a finite state space.
If a sum variable occurs only under its summation sign, then this summation sign is eliminated.

If the range of a sum variable $y^i_k$ is restricted to a single value $d$, then its summation sign is eliminated and $y^i_k$ is replaced by $d$.

sumelm determines whether $h_i(\vec{x}, \vec{y}^i)$ implies $eq(y^i_k, d)$ for some $d$.

If $h_i(\vec{x}, \vec{y}^i)$ is a

- conjunction, one of the conjuncts must imply this equality;
- disjunction, both disjuncts must imply this equality.
Consider the LPE

\[
X(d : D, b : Bit) = \sum_{d' : D} a \cdot X(d', b) \triangleleft \text{eq}(d, d_2) \lor \text{eq}(b, 0) \triangleright \delta \\
+ \sum_{b' : Bit} c \cdot X(d, b') \triangleleft \text{eq}(b', 0) \triangleright \delta
\]

The initial state is \(X(d_1, 0)\).

\text{sumelm} \text{ replaces } b' \text{ by 0 and removes its summation sign.}

\text{rewr} \text{ rewrites } \text{eq}(0, 0) \text{ to } T.
Simplification of LPEs - Example

\[ X(d : D, b : Bit) = \sum_{d' : D} a \cdot X(d', b) \triangleleft eq(d, d_2) \lor eq(b, 0) \triangleright \delta \]
\[ + c \cdot X(d, 0) \triangleleft T \triangleright \delta \]

constelm marks \( d \) (due to the occurrence of \( d' \) in the first argument of \( X \)), but leaves \( b \) unmarked.

So \( b \) is replaced by its initial value 0, and this parameter is removed from \( X \).

rewr rewrites \( eq(d, d_2) \lor eq(0, 0) \) to \( T \).
Simplification of LPEs - Example

\[ X(d : D) = \sum_{d' : D} a \cdot X(d') \triangleleft T \triangleright \delta + c \cdot X(d) \triangleleft T \triangleright \delta \]

Since \( d \) occurs neither in conditions nor in arguments of actions, \texttt{parelm} removes this parameter from \( X \).

Finally, \texttt{sumelm} removes the sum variable \( d' \).

\[ X = a \cdot X \triangleleft T \triangleright \delta + c \cdot X \triangleleft T \triangleright \delta \]
We take $\text{Bool} = \{0, 1\}$.

Let each state be encoded as a *bit string* of length $n$.

Transitions are expressed by functions

$$\varphi_a : \text{Bool}^{2n} \rightarrow \text{Bool}$$

for each $a \in \text{Act} \cup \{\tau\}$, where

$$\varphi_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1$$

if and only if $(b_1, \ldots, b_n) \xrightarrow{a} (b'_1, \ldots, b'_n)$. 
Any $\varphi : \mathbb{Bool}^m \to \mathbb{Bool}$ can be represented as a binary decision tree of depth $m$.

Each node at depth $i < m$ has two outgoing edges to nodes at depth $i + 1$, labelled with 0 and 1.

The leaves (at depth $m$) carry a label from $\mathbb{Bool}$, such that any path from the root of the binary decision tree over edges labelled with $b_1, \ldots, b_m$ leads to a leaf with the label $\varphi(b_1, \ldots, b_m)$.
\[(b_1 \land b_2) \lor (\neg b_1 \land \neg b_3)\], with \(b_i\) the \(i\)th argument of this formula, results in

---

**Question**: Which binary decision tree do we get if the arguments are reversed (i.e., \(b_3, b_2, b_1\)?)
An ordered binary decision diagram (OBDD) is obtained by maximal sharing of nodes in a binary decision tree.

The leaves labelled 0 are collapsed, and also the leaves labelled 1.

Next, two minimisation steps are performed:

- If two non-leaves associated with the same boolean variable have 0-transitions to the same node as well as 1-transitions to the same node, then they are collapsed.

- If the 0- and 1-transition of a non-leaf $\nu$ lead to the same node $\nu'$, then $\nu$ is eliminated, and its incoming edges are redirected to $\nu'$.

These steps are repeated until neither of them can be applied anymore.

Each order in which the steps are performed leads to the same result.
We transform the binary decision tree of \((b_1 \land b_2) \lor (\neg b_1 \land \neg b_3)\) into an OBDD.

First leaves with the same label are collapsed.
The two rightmost nodes associated with $b_3$ can both be eliminated, because their 0- and 1-transition lead to the same node.
The two remaining nodes associated with $b_3$ can be collapsed, because their 0- and their 1-transitions lead to the same node.
Finally, the leftmost node associated with $b_2$ can be eliminated, because its 0- and 1-transition lead to the same node.
Checking equivalence of two mappings from $\text{Bool}^m$ to $\text{Bool}$ boils down to checking isomorphism of the corresponding OBDDs.

A different ordering on variables usually produces a different OBDD.

**Question:** What is the OBDD of $(b_1 \land b_2) \lor (\neg b_1 \land \neg b_3)$ with reversed arguments $b_3, b_2, b_1$?

The chosen ordering can have a huge impact on the number of nodes in the resulting OBDD.

Finding an ordering on variables to obtain an OBDD of minimal size is an NP-complete problem.
Operations on OBDDs

Let $\mathcal{O}$ and $\hat{\mathcal{O}}$ both be OBDDs over boolean variables $b_1, \ldots, b_m$.

The OBDDs for $\neg \mathcal{O}$, $\mathcal{O} \land \hat{\mathcal{O}}$, $\mathcal{O} \lor \hat{\mathcal{O}}$, $\exists b \mathcal{O}$ and $\forall b \mathcal{O}$ can be computed efficiently.

The resulting graph is in general not yet an OBDD.
As final step, the minimization procedure produces an OBDD.

**Question:** How is the OBDD for $\neg \mathcal{O}$ obtained?
Suppose the roots of $O$ and $\hat{O}$ are associated with the same variable.

Let their 0-transitions lead to OBDDs $O'$ and $\hat{O}'$, respectively.

Let their 1-transitions lead to the OBDDs $O''$ and $\hat{O}''$, respectively.

The 0- and 1-transition from the root of $O \lor \hat{O}$ lead to the OBDDs for $O' \lor \hat{O}'$ resp. $O'' \lor \hat{O}''$, which are computed recursively.

**Question**: What to do if the root of $O$ and $\hat{O}$ are associated to *different* boolean variables?

**Question**: What to do if $O$ and $\hat{O}$ are *leaves*?

**Question**: How is the OBDD for $O \land \hat{O}$ computed?
The OBDD for $\exists b \mathcal{O}$ is obtained as follows.

- OBDDs $\mathcal{O}'$ and $\mathcal{O}''$ are obtained by eliminating each node $\nu$ in $\mathcal{O}$ associated with $b$.

  For $\mathcal{O}'$, all incoming edges of $\nu$ are redirected to the node that results after taking the 0-transition from $\nu$.

  For $\mathcal{O}''$, all incoming edges of $\nu$ are redirected to the node that results after taking the 1-transition from $\nu$.

- Compute the OBDD for $\mathcal{O}' \lor \mathcal{O}''$.

**Question:** How is the OBDD for $\forall b \mathcal{O}$ computed?
States as bit strings

Recall that each state is encoded as a *bit string* of length \( n \).

Transitions are expressed by OBDDs

\[
O_a : \text{Bool}^{2n} \to \text{Bool}
\]

for each \( a \in \text{Act} \cup \{\tau\} \), where

\[
O_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1
\]

if and only if \((b_1,\ldots,b_n) \xrightarrow{a} (b'_1,\ldots,b'_n)\).

For any \( \mu \)-calculus formula \( \phi \), an OBDD \( B(\phi) \) over \( b_1, \ldots, b_n \) is computed, representing the states that satisfy \( \phi \)
Symbolic model checking

TRUE is the leaf 1, and FALSE the leaf 0.

\[ B(T) = \text{TRUE} \]

\[ B(F) = \text{FALSE} \]

\[ B(\phi \land \phi') = B(\phi) \land B(\phi') \]

\[ B(\phi \lor \phi') = B(\phi) \lor B(\phi') \]

\[ B(\langle a \rangle \phi) = \exists b'_1 \cdots \exists b'_n (O_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \land B'(\phi)) \]

\[ B([a] \phi) = \forall b'_1 \cdots \forall b'_n (\neg O_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \lor B'(\phi)) \]

\[ B'(\phi) \text{ is an OBDD over } b'_1, \ldots, b'_n. \]
Symbolic model checking

\[ \mathcal{B}(\mu X. \phi) = \text{FIX}(\phi, X = \text{FALSE}) \]

\[ \mathcal{B}(\nu X. \phi) = \text{FIX}(\phi, X = \text{TRUE}) \]

\text{FIX}(\phi, X = O) \] iteratively computes OBDD \( \mathcal{B}(\phi) \) over \( b_1, \ldots, b_n \), by induction over the structure of \( \phi \), with a different interpretation of \( \mathcal{B}(X) \) in each iteration step.

In the first iteration step, \( \mathcal{B}(X) = \mathcal{O} \).

In the second iteration step, \( \mathcal{B}(X) = \text{FIX}(\phi, X = \mathcal{O}) \), etc.

This is repeated until a fixpoint is reached.

**Question:** What is the value of \( \mathcal{B}(X) \) in the third iteration?
The computation of $\mathcal{B}(\phi)$ can be expressed as solving a boolean equation system: an ordered list of equations $\mu x = b$ and $\nu x = b$, with $b$ a boolean formula.

For $\mu$CRL, we need to solve a parametrized boolean equation system of equations

\[
\mu x (d_1 : D_1, \ldots, d_n : D_n) = b \\
\nu x (d_1 : D_1, \ldots, d_n : D_n) = b
\]

Overview of the µCRL toolset

- constelm
- sumelm
- parelm
- rew
- instantiator with confluence reduction
- lineariser
- pretty printer
- minimisation
- confluence reduction
- instantiator

µCRL

LPE

state space

BCG

minimisation
confluence reduction

instantiator

instantiator with confluence reduction

pretty printer

lineariser

text

simulation

model checking

symbolic model checking

theorem proving
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.

This reduction of the state space comes at the price of loss of information.
Abstraction - Bag of size $N$

\[
\begin{array}{c}
\text{in}(0) \\ \text{out}(0) \\
\vdots \\
\text{in}(0) \\ \text{out}(0)
\end{array}
\]

\[
\begin{array}{c}
\text{in}(1) \\ \text{out}(1) \\
\vdots \\
\text{in}(1) \\ \text{out}(1)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(0) \\
\vdots \\
\text{out}(1) \\ \text{in}(0)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(1) \\
\vdots \\
\text{out}(1) \\ \text{in}(1)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(0) \\
\vdots \\
\text{out}(1) \\ \text{in}(0)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(1) \\
\vdots \\
\text{out}(1) \\ \text{in}(1)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(0) \\
\vdots \\
\text{out}(1) \\ \text{in}(0)
\end{array}
\]

\[
\begin{array}{c}
\text{out}(1) \\ \text{in}(1) \\
\vdots \\
\text{out}(1) \\ \text{in}(1)
\end{array}
\]
Abstraction - Bag of size $N$

Let $N \geq 3$.

Let $\mathcal{S} = \{\text{empty}, \text{middle}, \text{full}\}$ and $\mathcal{A} = \{\hat{1}, \hat{0}\}$.

\[
\begin{align*}
\pi(s_{0,0}) & = \text{empty} \\
\pi(s_{i,j}) & = \text{middle} & 0 < i + j < N \\
\pi(s_{i,j}) & = \text{full} & i + j = N \\
\theta(\text{in}(b)) & = \hat{1} & b \in \{0, 1\} \\
\theta(\text{out}(b)) & = \hat{0} & b \in \{0, 1\}
\end{align*}
\]
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 exists 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.
The abstracted state space for the bag of size $N \geq 3$ is
Model checking with abstraction

\[
\begin{align*}
C(T) &= \hat{S} \\
C(F) &= \emptyset \\
C(\phi \land \phi') &= C(\phi) \cap C(\phi') \\
C(\phi \lor \phi') &= C(\phi) \cup C(\phi') \\
C(\langle \hat{a} \rangle \phi) &= \{ \hat{s} \in \hat{S} \mid \exists \hat{s} \xrightarrow{\hat{a}} \Box \hat{s}' \ (\hat{s}' \in C(\phi)) \} \\
C([\hat{a}] \phi) &= \{ \hat{s} \in \hat{S} \mid \forall \hat{s} \xrightarrow{\hat{a}} \Diamond \hat{s}' \ (\hat{s}' \in C(\phi)) \}
\end{align*}
\]

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 \( \theta^{-1}(\hat{a}) \).
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.
The initial state of the *abstracted* state space of the bag of size $N$ satisfies

$$[(¬(î))^* \cdot \hat{\circ}] F$$

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

$$[(¬(in(0) \mid in(1)))^* \cdot (out(0) \mid out(1))] F$$