Coordination and Software Composition Using Reo

Farhad Arbab

Center for Mathematics and Computer Science (CWI), Amsterdam
Leiden Institute of Advanced Computer Science, Leiden University

Estonian Summer School on Computer and System Science (ESSCaSS)
Bernhard Hotel, Otepää, Estonia

21- 25 August 2011
Software Engineering

- Applying engineering discipline to construction of complex software intensive systems.
- A hallmark of all engineering disciplines is composition:
  - Construct more complex systems by composing simpler ones.
  - Derive properties of composed system as a composition of the properties of its constituents.
Evolution of Software Engineering

- Structured programming with subprograms and functions
- Object-oriented paradigm (OO)
- Component-based software engineering (CBSE)
- Service-oriented Computing (SoC)
  - Highly interactive, open, dynamic, heterogeneous, distributed
  - Service as the basic building block
  - Many existing examples
  - Strong academic and industrial interest
  - Active standardization efforts
- Software for multi-core systems
Common Composition Mechanisms

- Increasing flexibility of composition mechanisms:
  - Procedure call (local/remote)
    - Invokes a specific algorithm
    - Hard-codes the called algorithm into the caller
    - Intermixes computation and concurrency
  - Method invocation (local/remote)
    - Used in OO and OO-components
    - Invokes an operation on an object
    - Adds a level of indirection between caller and called algorithms
    - Looser control-flow coupling in asynchronous versions
  - Targeted messages (local/remote)
    - Used in concurrency platforms, non-OO-components, SoC
    - No control-flow coupling
    - Still hinges on referring to a foreign entity (the target)
Engineering of Complex Systems

- Engineering tackles complexity by:
  - **Coping with it**: Practice of Engineering
    - Methodologies
    - Standards, certification
    - Best practices
    - The art of engineering
  - **Simplifying it**: Science behind engineering
    - Deeper study of the foundational phenomena
    - Appropriate levels of abstraction
    - Formal, mathematical models
Complexity vs. Bewilderment

- **Complex Systems**

- **Complex task/algorithm/computation**
  - Examples:
    - Computations/equations in quantum mechanics, astronomy, engineering, etc.
    - Bit-map to jpeg conversion, sorting, etc.
  - This type of complexity is not bewildering!
    - Good, intricate mathematical models have tamed the complexity.

- **Systems** of simple components can exhibit very bewildering behavior
  - Example:
    - 4 components send messages to each other (12)
    - Each component can be in one of 4 states (256 system states)
    - Exchanges in the context of system state (3072 possibilities)
    - Asynchronous exchange: more to consider!
    - More than a single type of message: multiplicatively more to consider!
  - Bewildering complexity emerges out of interaction
  - Good formal models to tame this complexity?
Models of Concurrency

- Traditional models are action based
  - Petri nets
  - Work flow / Data flow
  - Process algebra / calculi
  - Actor models / Agents
  - ...

- A system composed of building blocks that represent actions/processes

- Interaction becomes an implicit side-effect
  - Makes coordination of interaction more difficult to
    - Specify
    - Verify
    - Manipulate
    - Reuse
Multi-core Processors

- Offer potential for massive concurrency
  - ILP, fine-grain parallelism, and SIMD
  - Coarse-grain
  - Low inter-process communication overhead
- Manual techniques cannot harness this potential
- Scaling-up (coarse-grain) concurrency
  - Coordination of interaction becomes dominant complexity
  - Requires compositional models
- Models of concurrency
  - Action based (shared memory, message passing, etc.)
  - Interaction based
Producers and Consumer

- Construct an application consisting of:
  - A Display consumer process
  - A Green producer process
  - A Red producer process

- The consumer must alternately display the contents made available by the Green and the Red consumers.
Java-like Implementation

- Shared entities
  ```java
  private final Semaphore greenSemaphore = new Semaphore(1);
  private final Semaphore redSemaphore = new Semaphore(0);
  private final Semaphore bufferSemaphore = new Semaphore(1);
  private String buffer = EMPTY;
  ```

- Consumer
  ```java
  while (true) {
      sleep (4000);
      bufferSemaphore.acquire();
      if (buffer != EMPTY) {
          println(buffer);
          buffer = EMPTY;
      }
      bufferSemaphore.release();
  }
  ```

- Producers
  ```java
  while (true) {
      sleep (5000);
      greenText = ...
      greenSemaphore.acquire();
      bufferSemaphore.acquire();
      buffer = greenText;
      bufferSemaphore.release();
      redSemaphore.release();
  }
  ```

- Where is green text computed?
- Where is red text computed?
- Where is text printed?
- Where is the protocol?
  - What determines who goes first?
  - What determines producers alternate?
  - What provides buffer protection?
  - Deadlocks?
  - Live-locks?
  - ...

- Protocol becomes
  - Implicit, nebulous, and intangible
  - Difficult to reuse
Skeletions

- **Concurrency control-flow abstractions**
  - Skeletal structures of primitives implementing a protocol
  - Must be fleshed out with computation code
  - Very useful in very narrow domains
  - Entanglement of computation can affect protocol

```java
... 
private final Semaphore greenSemaphore = new Semaphore(1);
private final Semaphore redSemaphore = new Semaphore(0);
private final Semaphore bufferSemaphore = new Semaphore(1);
private String buffer = EMPTY;
...

greenSemaphore.acquire();
bufferSemaphore.acquire();
...
bufferSemaphore.release();
redSemaphore.release();
...
...
redSemaphore.acquire();
bufferSemaphore.acquire();
...
bufferSemaphore.release();
greenSemaphore.release();
...
...```
Process Algebraic Model

- **Shared entities**
  - Synchronization points: g, r, b, d

- **Consumer**
  - B := ?b(t) . print(t) . !d("done") . B

- **Producers**
  - G := ?g(k) . genG(t) . !b(t) . ?d(j) . !r(k) . G
  - R := ?r(k) . genR(t) . !b(t) . ?d(j) . !g(k) . R

- **Model**
  - G | R | B | !g("token")

- What are the primitives and constructs in this model?
  - Shared names to synchronize communication:
    - g, r, b, d
  - Atomic actions:
    - Primitive actions defined by algebra:
      - ?(_,_), !(_,_)
    - User-defined actions:
      - genG(_), genR(_), print(_)
  - Composition operators:
    - ., |, +, :=, implied recursion

- A model is constructed by composing (atomic) actions into (more complex) actions.
- Primarily a model of actions/processes
  - Hence the name “process algebra”
- Where is interaction?
Implicit Interaction

- Interaction (protocol) is implicit in action-based models of concurrency
- Interaction is a by-product of processes executing their actions
  - Action $a_i$ of process A collides with action $b_j$ of process B
  - Interaction is the specific (timed) sequence of such collisions in a run
  - Interaction protocol is the (timed) sequence of the intended collisions in such a sequence.

- How can the intended and the coincidental be differentiated?
- How can the sequence of intended collisions (the interaction protocol) can:
  - Manipulated?
  - Verified?
  - Debugged?
  - Reused?
  - ...

Possible only indirectly, through manipulating processes
Elements of Interaction

- **Coordination** is *constrained interaction*: it constrains interaction protocols among communicating software components.
  
  - “Coordination as Constrained Interaction” Peter Wegner

- What is an interaction protocol?
  
  - Synchrony / asynchrony
  - Atomicity
  - Ordering
  - Exclusion
  - Grouping
  - Selection
  - ...

- How to formalize interaction explicitly?
  
  - *Constraints*
Explicit Interaction

- We can represent interaction as a constraint
- Offers a model of concurrency where interaction becomes an explicit, tangible concept.
- Protocols can be defined as constraint programs
- Protocols can be composed as constraint composition
- Properties of protocols can be analyzed as the properties of a set of constraints, independently of the properties of the actors they engage
Interaction Based Concurrency

- Start with a set of primitive interactions as binary constraints
- Define (constraint) composition operators to combine interactions into more complex interactions
- Properties of the resulting model of concurrency depend on
  - Set of primitive interactions
  - Composition operators
- As constraints, interaction protocols can be manifested independently of the processes that they engage
  - Connectors
- Imposing an interaction on actors exogenously coordinates their activities
Exogenous Coordination

- P and C are black-box component/services that:
  - Offer no inter-service methods nor make such calls
  - Do not send/receive targeted messages
  - Their only means of communication is through blocking I/O primitives that they can perform on their own ports:
    - \( \text{get}(p, v) \) or \( \text{get}(p, v, t) \)
    - \( \text{put}(p, v) \) or \( \text{put}(p, v, t) \)
  - Composing P and C with different connectors (that impose different protocols from outside) constructs different systems.
Reo

http://reo.project.cwi.nl

- Reo is an exogenous coordination language for compositional construction of interaction protocols.

- Interaction is the only first-class concept in Reo:
  - Explicit constructs representing interaction
  - Composition operators over interaction constructs

- A (coordination or interaction) protocol:
  - manifests as a connector
  - gets imposed on its engaged components/services from outside
  - remains mutually oblivious to its engaged components/services

- Reo offers:
  - Loose(st) coupling
  - Arbitrary mix of asynchrony, synchrony, and exclusion
  - Open-ended user-defined primitive channels
  - Distribution and mobility
  - Dynamically reconfigurable connectors
Concurrency in Reo

- Reo embodies a non-conventional model of concurrency:

- **Conventional**
  - action based
  - process as primitive
  - imperative
  - functional
  - imperative programming
  - protocol implicit in processes

- **Reo**
  - interaction based
  - Protocol as primitive
  - declarative
  - relational
  - constraint programming
  - Tangible explicit protocols

- Reo is more expressive than Petri nets, workflow, and dataflow models.
Channels

- Atomic connectors in Reo are called *channels*.
- Reo generalizes the common notion of channel.
- A channel is an abstract communication medium with:
  - exactly two ends; and
  - a constraint that relates (the flows of data at) its ends.
- Two types of channel ends
  - Source: data enters into the channel.
  - Sink: data leaves the channel.
- A channel can have two sources or two sinks.
- A channel represents a primitive interaction.
Binary Interaction Constraints

- Synchronous interaction
- Synchronous drain interaction
- Synchronous spout interaction
- Lossy synchronous interaction
- Asynchronous FIFO1 interaction
Primitive Channels

- Synchronous channel
  - write/take
- Synchronous drain channel
  - write/write
- Synchronous spout channel
  - take/take
- Lossy synchronous channel
  - write/take
- Asynchronous FIFO1 channel
  - write/take
Join

- Mixed node
  - Merge + replication combo
- Sink node
  - Non-deterministic merge
- Source node
  - Replication
Reo Connectors

- FIFO1 channel
- synchronous channel
- lossy synchronous channel
- filter channel
- P-producer ≤τ
- synchronous drain
- asynchronous drain
- synchronous spout
- asynchronous spout
- timer channel

Exclusive choice (deferred XOR)

Valve connector: controls flow from A to B

© F. Arbab 2011
A Simple Composed System

- Read-cue synchronous flow-regulator
Flow regulator

Write-cue synchronous flow-regulator
Flow Synchronization

- The write/take operations on the pairs of channel ends a/c and b/d are synchronized.

- Barrier synchronization.

```
   !x  ?  x
  a   c

   !y  ?  y
  b   d
```
Take a through d when b or c

- A take from d succeeds only if there is a value written to b or c.
- The values taken from d are elements of the stream a*
- We have d = ((b|c)/a)*
Alternator

- Subsequent takes from c retrieve the elements of the stream \((ab)^*\)
- Both a and b must be present before a pair can go through.
Alternator for $n>2$ Sources

As a step toward generalization, add a redundant Sync channel:

\[ \text{Diagram showing a circuit with nodes a, b, and c.} \]
N-Alternator

Subsequent takes from z retrieve the elements of the stream \((abcd)^*\)
We can use the alternator circuit to impose the protocol on the green and red producers of our example
- From outside
- Without their knowledge
**Sequencer**

- Writes to a, b, c, and d will succeed cyclically and in that order.
Sequenced Producers

A two-port sequencer and a few channels form the connector we need to compose and exogenously coordinate the green/red producers/consumer system.
Overflow Lossy FIFO1

A FIFO1 channel that accepts but loses new incoming values if its buffer is full.
Exclusive Router

- A value written to \( a \) flows through to either \( b \) or \( c \), but never to both.
Inclusive Router

- A value written to $a$ flows through to either $b$ or $c$, or to both.
Shift Lossy FIFO1

- A FIFO1 channel that loses its old buffer contents, if necessary, to make room for new incoming values.
Repeater

Every take on the output node of this connector obtains the same constant value, \( v \).
Replacer

- Pairs of write and take operations succeed on the input and output nodes of this connector synchronously.
- All written values are lost and every take obtains the same constant value, $v$. 
Variable

- Every input value is remembered and repeatedly reproduced as output, zero or more times, until it is replaced by the next input value.
Buffered Producers

- Adding variables to the sequencer solution, buffers the actions of the producers and the consumer.
Another \( c = (ab)^* \)

- An \( a \) can pass through even if its pairing \( b \) is not yet available.
$c = (aab)^*$
Inhibitor

- All values flow from d to c until a value is written to i.
- A write to i *inhibits* (i.e., blocks) further writes to both d and i.

```
d ───> c
     """""""""""""""""
     ^
     |   |
     | ▼
     i ───> O
```

© F. Arbab 2011
\[ C = a^* \mid b^* \]

- The drain is asynchronous; dashed arrows show synchronous lossy channels; all other channels are synchronous.
Asynchronous Drain

- An AsyncDrain can be composed out of a SyncDrain and 3 (or 2) Sync channels.
Sequencer With Reset

Same as a simple sequencer, except that a write to reset forces the sequencer to start over with \textit{a}.
Valve (open)

- A write to \( c \) closes the flow of data from \( a \) to \( b \).
Valve (closed)

A write to $c$ opens the flow of data from $a$ to $b$. 
$n$-Counter

- Makes a token available on its output upon the availability of every $4^{th}$ input data item.
$n^{th}$-Filter

- Passes every $4^{th}$ input item.
Sparing Delayed 1-out-of-\(n\)

- Outputs one of the \(n\) input values in each cycle.
- Output is delayed until the end of a cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are spared for the next cycle.
Lossy Delayed 1-out-of-$n$

- Outputs one of the $n$ input values in each cycle.
- Output is delayed until the end of a cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are lost as they arrive.
Sparing Delayed 1\textsuperscript{st} Out of \( n \)

- Outputs only the \textit{first} of the \( n \) arriving inputs in each cycle.
- Output is \textit{delayed} until the end of each cycle.
- \textit{Cycle ends after}:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are \textit{spared} for the next cycle.
Lossy Delayed 1st Out of n

- Outputs only the first of the $n$ arriving inputs in each cycle.
- Output is delayed until the end of each cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are lost as they arrive.
Sparing Prompt 1st Out of $n$

• Outputs only the first of the $n$ arriving inputs in each cycle.
• Output is possible prompt after the first input arrives.
• Cycle ends after:
  • A value arrives on each input node, and
  • A value is taken from the output node.
• Extra input values of a node are spared for the next cycle.
Lossy Prompt 1st Out of n

- Outputs only the first of the n arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are lost as they arrive.
Sparing Forced 1st Out of n

- Outputs only the first of the n arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is forced to end after a value is taken from the output.
- Extra input values of a node are spared for the next cycle.
Lossy Forced 1st Out of n

- Outputs only the first of the n arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is forced to end after a value is taken from the output.
- Extra input values of a node are lost as they arrive.
Sparing Prompt $m$ Out of $n$

- Outputs only the first of the $n$ arriving inputs in each cycle.
- Output is possible prompt after the first $m$ input values arrive.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are spared for the next cycle.
Lossy Prompt $m$ Out of $n$

- Outputs only the first of the $n$ arriving inputs in each cycle.
- Output is possible prompt after the first $m$ input values arrive.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are lost as they arrive.
Sparing Forced $m$ Out of $n$

- Outputs only the **first** of the $n$ arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is **forced** to end after a value is taken from the output.
- Extra input values of a node are **spared** for the next cycle.
Lossy Forced $m$ Out of $n$

- Outputs only the first of the $n$ arriving inputs in each cycle.
- Output is possible prompt after the first input arrives.
- Cycle is forced to end after a value is taken from the output.
- Extra input values of a node are lost as they arrive.
Synchronous FIFO1

- Combines the behavior of Sync and FIFO1.
- Behaves as a FIFO1, except that if the buffer is empty, and
  - a take is pending on B,
  - the value written to A synchronously goes to B and leaves the buffer empty.

A  B
Write $z$ to SyncFIFO1
Take $z$ from SyncFIFO1
SyncFIFO1 Resets Itself
Write After a Pending Take
After Synchronous Write/Take
Linda

A model for coordination of parallel processes operating on *tuples* stored in and retrieved from a logical shared *tuple-space*.

**Operations:**

- **out(t)**
  - Adds tuple \( t \) to the tuple space.
  - Always succeeds “immediately”.

- **in(p)**
  - Returns a tuple \( t \) from the tuple space that matches pattern \( p \).
  - If no such \( t \) exists, the operation blocks.
  - Removes \( t \) from the tuple space.

- **rd(p)**
  - Same as \( \text{in}(p) \), but does not remove the matched tuple.

- **eval(e)**
  - Creates a new process to run the expression \( e \).

Some extensions include primitives to test non-existence of tuples.
Conceptual view of Linda

Tuple Space

- Synchronous I/O by processes
- Blocking
- Only asynchronous inter-process communication
- Pattern matching
- Centralized
Linda vs Reo

- Synchronous I/O by processes
- Blocking
- Only asynchronous inter-process communication
- Pattern matching

Reo Circuits
- Synchronous I/O by processes
- Both sync & async inter-process communication
- Arbitrarily complex protocols composed of
  - User defined primitives
  - Arbitrary mix of synchrony and asynchrony
- Distributed
Linda vs Reo

- Synchronous I/O by processes
- Blocking
- Only asynchronous inter-process communication

- Pattern matching
- Centralized

Linda
- Tuple Space
- Reo Circuits

Reo
- Synchronous I/O by processes
- Both sync & async inter-process communication
- Arbitrarily complex protocols composed of
  - User defined primitives
  - Arbitrary mix of synchrony and asynchrony
  - Distributed
Reo vs Linda

Blue must alternately read the outputs of Green and Red

Tuple Space

- Synchronous I/O by processes
- Blocking
- Only asynchronous inter-process communication
- Pattern matching
- Centralized
Dining Philosophers (problem)
Dining Philosophers (solution)
Fork

- The fork component used in the dining philosophers problem is a pure coordinator and can be constructed as a Reo connector circuit.
Semantics

- Reo allows:
  - Arbitrary user-defined channels as primitives.
  - Arbitrary mix of synchrony and asynchrony.
  - Relational constraints between input and output.
- Reo is more expressive than, e.g., dataflow models, Kahn networks, workflow models, stream processing models, Petri nets, and synchronous languages.
- Formal semantics:
  - Coalgebraic semantics based on timed-data streams.
  - Constraint automata.
  - SOS semantics (in Maude).
  - Constraint propagation (connector coloring scheme).
  - Intuitionistic linear logic
Abstract Behavior Type

- An ABT defines an abstract behavior without specifying any detail about:
  - the operations that may be used to implement such behavior.
  - the data types they may manipulate for its realization.

- We use a (maximal) relation among a set of timed-data-streams to represent an ABT.

- For ABT $R(I_1, I_2, ... I_m; O_1, O_2, ... O_n)$ the $I_m$ are its input and $O_n$ are its output portals.
Streams

- For any set $A$, an infinite sequence over $A$ is called a **stream** (over $A$).
- Elements in a stream $\alpha$ are
  - indexed by non-negative integers, and
  - denoted as $\alpha(0)$, $\alpha(1)$, $\alpha(2)$, ...
- Head and tail of $\alpha$ are
  - $\alpha(0)$ the initial value of $\alpha$
  - $\alpha'$ the (first) derivative of $\alpha$
- The $k$-th tail of $\alpha$ is
  - $\alpha^{(k)}$ the $k$-th derivative of $\alpha$
- Relational operators on streams:
  - $\alpha < \beta$ is shorthand for $\alpha(0) < \beta(0)$, $\alpha(1) < \beta(1)$, $\alpha(2) < \beta(2)$, ...
A timed-data-stream is a twin pair of infinite streams:
\[ \langle \alpha, a \rangle \]

- **Data stream** $\alpha$
  - Elements of $\alpha$ are uninterpreted data items
- **Time stream** $a$
  - Elements of $a$ are non-negative real numbers
  - Time elapses incrementally: $\forall i \geq 0, a(i) < a(i + 1)$
  - Finite steps in any interval: $\forall N, \exists i : a(i) > N$
- **Data item** $\alpha(i)$ is observed at time $a(i)$. 
Unbounded Queue

The behavior of an unbounded queue is defined as the relation FIFO between a pair of input/output timed-data-streams:

\[
\langle \alpha, a \rangle \text{FIFO} \langle \beta, b \rangle \equiv \alpha = \beta \land a < b
\]

Whatever goes in ...
- Comes out,
- After it goes in.
1-Bounded Queue

The behavior of a queue with bounded capacity of 1 is defined as the relation FIFO1 between a pair of input/output timed-data-streams:

\[ \langle \alpha, a \rangle \text{FIFO1} \langle \beta, b \rangle \equiv \alpha = \beta \land a < b < a \]

Whatever goes in ...
- Comes out,
- After it goes in,
- But before the next item goes in.
The behavior of a queue with bounded capacity of \( k \) is defined as the relation \( \text{FIFO}_{k} \) between a pair of input/output timed-data-streams:

\[
\langle \alpha, a \rangle \text{FIFO}_k \langle \beta, b \rangle \equiv \alpha = \beta \land a < b < a^{(k)}
\]

Whatever goes in ...

- Comes out,
- After it goes in,
- But before the \( k \)-th next item goes in.
A Sample of Channels

- Synchronous channel
  - write/take \( \langle \alpha, a \rangle \) sync \( \langle \beta, b \rangle \) \( \equiv \alpha = \beta \land a = b \)

- Synchronous drain: two sources
  - write/write \( \langle \alpha, a \rangle \) synr \( \langle \beta, b \rangle \) \( \equiv a = b \)

- Synchronous spout: two sinks
  - take/take \( \langle \alpha, a \rangle \) synsp \( \langle \beta, b \rangle \) \( \equiv a = b \)

- Lossy synchronous channel
  \( \langle \alpha, a \rangle \) synloss \( \langle \beta, b \rangle \) \( \equiv \beta = L(\alpha, a, b) \)
  \( L(\alpha, a, b) = \begin{cases} \alpha(0) \lor L(\alpha', a', b') & \text{if } b(0) \leq a(0) < b(1) \\ L(\alpha', a', b) & \text{otherwise} \end{cases} \)

- Asynchronous FIFO1 channel
  - write/take \( \langle \alpha, a \rangle \) fifo1 \( \langle \beta, b \rangle \) \( \equiv \alpha = \beta \land a < b < a' \)
Merger ABT

- An ABT that merges two sorted incoming timed-data-streams into an outgoing one is a ternary relation:

\[
M(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) = \begin{cases} 
\alpha(0) = \gamma(0) \land a(0) = c(0) \land M(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) \leq b(0) \\
\beta(0) = \gamma(0) \land b(0) = c(0) \land M(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma, c' \rangle) & \text{otherwise}
\end{cases}
\]

- This merger is biased toward its first input.

- Alternative mergers:
  - Biased toward its second input.
  - Non-deterministic choice, fair or otherwise.
Replicator ABT

- The behavior of a replicator is defined as a ternary relation among one input and two output timed-data-streams:

\[ R(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \beta = \alpha \land \gamma = \alpha \land b = a \land c = a \]

- Whatever goes in ...
  - Comes out in duplicates,
  - Simultaneously as it goes in.
ABT Composition

- Two ABTs can be composed over $k \geq 0$ pairs of their portals, yielding another ABT through a special form of relational join composition:
  - The portals in each pair must have opposite senses.
  - The timed data streams in each pair must be equal.
  - The paired portals are deleted from the result.

$$R(I_1, I_2, \ldots I_m; O_1, O_2, \ldots O_n) \equiv R_1(I_{11}, I_{12}^3, \ldots I_{1p_1}^2; O_{11}^1, O_{12}, \ldots O_{1q_1}) \circ R_2(I_{21}, I_{22}^1, \ldots I_{2p_2}; O_{21}^2, O_{22}^3, \ldots O_{2q_2})$$
Expressiveness

What ABTs can be produced by composition of a given set of channels?

- With a small set of 5 channel types, the equivalent of regular expressions can be constructed.
- Turing equivalence is possible with the above set of channel types, plus unbounded FIFO.
Fibonacci Series

- This circuit produces the Fibonacci series using an adder component.
Automata Models for Reo

- Coalgebraic semantics of Reo allows proofs using coinduction as the only reasoning principle.
- Proofs that involve fixed-point solution are non-obvious in this model:
  - Containment
  - Equivalence
  - Correctness
  - Synthesis
  - Etc.
- Need for a formalism that allows model-checking.
  - Existing automata models are unsuitable
  - Constraint automata express the formal semantics of Reo.
  - They also provide a “centralized” implementation of Reo.
Constraint Automata

- States represent configurations.
- Transitions encode maximally-parallel stepwise evolution.
- Transition labels show maximal sets of active nodes and sets of data constraints.

© F. Arbab 2011
Constraint Automata of Basic Connectors

- Aifo channel (1-bounded)
- Sync channel
- Losy sync channel
- Filter channel
- $P$-producer
- Sync spout
- Sync drain
- Async spout
- Async drain

- Synchronous channel
  - $\{A, B\}$
  - $d_A = d_B$

- $P$-producer
  - $\{A, B\}$
  - $d_B \in P$

- Synchronous drain or spout
  - $\{A, B\}$

- Asynchronous drain or spout
  - $\{A\}$

- Losy synchronous channel
  - $\{A, B\}$
  - $d_A = d_B$

- Synchronous channel with filter
  - $\{A\}$
  - $d_A \notin P$

- Executer
  - $\{A, B\}$
  - $d_A = d_B$

- Shift-losy aifo channel (data abstract)
  - $\{A\}$

© F. Arbab 2011
Definition 4.1 [Product-automaton] The product-automaton of the two constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \rightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \rightarrow_2, Q_{0,2})$, is:

$$\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \rightarrow, Q_{0,1} \times Q_{0,2})$$

where $\rightarrow$ is defined by the following rules:

$$q_1 \xrightarrow{N_1,g_1} p_1, \quad q_2 \xrightarrow{N_2,g_2} p_2, \quad N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1$$

$$\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle$$

and

$$q_1 \xrightarrow{N,g} p_1, \quad N \cap \mathcal{N}ames_2 = \emptyset$$

$$\langle q_1, q_2 \rangle \xrightarrow{N,g} \langle p_1, q_2 \rangle$$

and latter’s symmetric rule. □
Product of 2 FIFO Automata
Hiding of Node C
Context Sensitive Behavior

- Certain channels may have context-sensitive behavior.
- Nodes must respect and propagate such context information.

write \[\rightarrow\] take

Write and take, data must be transferred
Effect on Node Behavior

- Node B must make sure that the first write to A is never lost.

- Even in this case
Other Automata Models

- The pure CA cannot capture context sensitivity
- Extensions to CA are necessary:
  - Intentional Constraint Automata
  - Context sensitive CA
  - Reo automata
Distributed Semantics

- Automata models capture the global behavior of a Reo circuit
- Reo primitives (must) act locally
  - Need a model to allow global behavior of a circuit emerge as a consensus of the possible local behavior alternatives of its primitives.
  - Primitives that coincide on a node must agree on a common behavior
    - Primitives constrain each other’s behavior alternatives
    - Viable global behavior can be found through constraint solving.
Node Expansion

- Explicitly represent the merge and replicate behavior of nodes as (builtin) primitives.

Merger

Replicator
Coloring Semantics

- A model for the semantics of Reo
  - Preserves circuit topology.
  - Allows an open set of primitives.
  - Composes behavior alternatives of primitives.
  - Suitable for distributed implementation.

- We use (initially two) different colors to represent alternative forms of (dataflow) behavior of primitives.

- Data flows
- Data does not flow
Merger (2-color)

- Alternative forms of dataflow behavior of merger in the 2-color scheme.
Replicator (2-color)

- Alternative forms of dataflow behavior of replicator in the 2-color scheme.
2-color Scheme

- Alternative forms of dataflow behavior of a typical set of channels.

- Representing I/O operations at boundary nodes:
Circuit Coloring

- Nodes must match the colors of their coincident channel ends.

- Total no-flow alternative always exists.
  - Annoyance: unbridled non-determinism can always choose it.
Lack of Context Awareness

- The 2-color scheme does not support context-sensitivity.
3-color Scheme

- Two different reasons for no-flow:
  - Unavailability:
    - A (place-holder for a) data item does not exist.
  - Exclusion:
    - The state of the channel refuses to use it.

- Adorn no-flow with one of two markers to show its cause.
3-color Scheme

- Distinguish between the two possible causes of no-flow:
  - Non-availability: inbound chevron
  - Exclusion: outbound chevron
  - The chevron points to the reason for no-flow

- Representing I/O operations at boundary nodes:
  - ■—■
  - ■→■
  - ■←■
  - □←□
  - □→□
Replicator (3-color)

- Alternative forms of dataflow behavior of replicator in the 3-color scheme.
Merger (3-color)

Alternative forms of dataflow behavior of merger in the 3-color scheme.
General Rules for 3-color Primitives (1)

In sensible primitives:

- A no-flow behavior alternative with exclusion on all of its ends is not allowed.
In sensible primitives:

- The existence of a behavior alternative with an exclusion no-flow on one of its ends implies that the primitive tolerates non-availability no-flow on that same end.
  - If this is present
  - Then this must be implied as well
  - If this is present
  - Then these must be implied as well
Context Awareness

- The 3-color scheme supports context-sensitivity.

- It works even when Sync channels are inserted at B!
A set of Eclipse plug-ins provide the ECT visual programming environment.

Protocols can be designed by composing Reo circuits in a graphical editor.

The Reo circuit can be animated in ECT.

ECT can automatically generate the CA for a Reo circuit.

Model-checkers integrated in ECT can be used to verify the correctness properties of a protocol using its CA.

ECT can generate executable (Java/C) code from a CA as a single sequential thread.

http://reo.project.cwi.nl
## Tool support

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reo graphical editor</td>
<td>Drag and drop editing of Reo circuits</td>
</tr>
<tr>
<td>Reo animation plug-in</td>
<td>Flash animation of data-flow in Reo circuits</td>
</tr>
<tr>
<td>Extensible Automata editor and tools</td>
<td>Graphical editor and other automata tools</td>
</tr>
<tr>
<td>Reo to constraint automata converter</td>
<td>Conversion of Reo to Constraint Automata</td>
</tr>
<tr>
<td>Verification tools</td>
<td>• Vereofy model checker (<a href="http://www.vereofy.de">www.vereofy.de</a>)</td>
</tr>
<tr>
<td></td>
<td>• mCRL model checking</td>
</tr>
<tr>
<td></td>
<td>• Bounded model checking of Timed Constraint Automata</td>
</tr>
<tr>
<td>Java code generation plug-in</td>
<td>State machine based coordinator code (Java, C, and CA interpreter for Tomcat servlets)</td>
</tr>
<tr>
<td>Distributed Reo middleware</td>
<td>Distributed Reo code generated in Scala (Actor-based Java)</td>
</tr>
<tr>
<td>(UML / BPMN / BPEL) GMT to Reo converter</td>
<td>Automatic translation of UML SD / BPMN / BPEL to Reo</td>
</tr>
<tr>
<td>Reo Services platform</td>
<td>Web service wrappers and Mash-ups</td>
</tr>
<tr>
<td>Markov chain generator</td>
<td>Compositional QoS model based on Reo Analysis using, e.g., probabilistic symbolic model checker Prism (<a href="http://www.prismmodelchecker.org">http://www.prismmodelchecker.org</a>)</td>
</tr>
<tr>
<td>Algebraic Graph Transformation</td>
<td>Dynamic reconfiguration of Reo circuits</td>
</tr>
</tbody>
</table>
Tool snapshots

Reo graphical editor

ReceiveOrder

Reo simulation plug-in

Reo to constraint automata converter

http://reo.project.cwi.nl
Snapshot of Reo Editor
Reo Animation Tool

Credit broker (Network)

List of animations

Animation 1
(2 steps)

Animation 2
(4 steps)

Animation 3
(4 steps)

Animation 4
(5 steps)
Constraint Automata Tools

- ECT includes a graphical editor for CA and related automata models
  - Create and edit automata graphically
  - Perform product and hiding on automata
- ECT includes tools to automatically derive the CA of a Reo circuit
- ECT includes simulator engines to show automata runs
Constraint Automata Editor
Model Checking

- Constraint automata are used for model checking of Reo circuits
- Model checker for Reo built in Dresden:
  - Symbolic model, LTL, and CTL-like logic for specification
  - Can also verify properties such as deadlock-freeness and behavioral equivalence
- SAT-based bounded model checking of Timed Constraint Automata
- Translation of Reo to mCRL for model checking
Vereofy Model Checker

- Symbolic model checker for Reo:
  - Based on constraint automata
  - Developed at the University of Dresden
  - LTL and CTL-like logic for property specification

- Modal formulae
  - Branching time temporal logic:
    - $AG[EX[true]]$
    - check for deadlocks
  - Linear temporal logics:
    - $G(request \rightarrow F(reject \cup sendFormOut))$
    - check that admissible states reject or sendFormOut are reached

- [http://www.vereofy.de](http://www.vereofy.de)
Verification with Vereofy

- **Modal formulae**
  - Branching time temporal logic: $\text{AG}[\text{EX}[\text{true}]]$ - check for deadlocks
  - Linear temporal logics: $\text{G}(\text{request} \rightarrow \text{F}(\text{reject} \cup \text{sendFormOut}))$ - check that admissible states reject or sendFormOut are reached
Data-Dependent Control-Flow

- **Input parameters:**
  - **Activation condition**
    - **Data:** b: Boolean
    - **Filter condition:** b==true, b==false
  - **Check condition**
    - **Data:** x, y: Real; (e.g., credit amount, maximal amount)
    - **Filter condition:** x < y

- **Problems:**
  - Data constraint specification language is needed
  - Properties that include conditions:
    - \( G [(b \& \neg(x < y)) \rightarrow F \text{ violation}] \)
Verification with mCRL2

- mCRL2 behavioral specification language and associated toolset developed at TU Eindhoven
  - [http://www.mcrl2.org](http://www.mcrl2.org)
  - Based on the Algebra of Communicating Processes (ACP)
  - Extended with data and time
  - Expressive property specification language ($\mu$ calculus)
  - Abstract data types, functional language ($\lambda$ calculus)

- Automated mapping from Reo to mCRL2
  - N. Kokash, E. d. V., C. Krause, Data-aware Design and Verification of Service Compositions with Reo and mCRL2, in: ACM Symposium on Applied Computing, 2010
mCRL2 specification language

- **Actions** are atomic events (e.g. a firing of a port or a request arrival in a Reo connector)
- **Processes** are the active entities defined as expressions over actions and other processes
  - Multiaction: \(a/b\) (synchronized actions)
  - Alternative composition: \(a + b\) (nondeterministic choice)
  - Sequence composition: \(a.b\) (\(b\) started after \(a\))
  - Conditional: \(\exp \rightarrow a \quad b\) (if-then-else)
  - At operator: \(a^t\) (action \(a\) happens at time \(t\))
  - Parallel composition: \(a//b\) (interleavings \(a.b + b.a + a|b\))
  - Renaming: \(\rho_{R}(a)\) where \(R\) is a set of renamings of the form \(b \rightarrow c\), meaning that every occurrence of \(b\) in \(a\) is replaced by \(c\)
  - Hiding: \(\tau_{H}(a)\) renames all actions of \(H\) in \(a\) to \(\tau\)
  - Blocking: \(\delta_{B}(a)\) where \(B\) is a set of actions that is not allowed to occur in \(a\)
  - Communication: \(\Gamma_{C}(p)\), where \(C\) is a set of allowed communications of the form \(a_0/.../a_n \rightarrow c\), \(n \geq 1\) which means that every group of actions \(a_0/.../a_n\) within a multiaction is replaced by an action \(c\)
- **Actions and processes** can be parametrized with **data**
  - Summation: \(\Sigma_{d \in D} a(d)\) (\(a(d_1) + a(d_2) + a(d_3)\)...)

© F. Arbab 2011
\( \mu \)-calculus

- **Modal logic with fix points operators**
  - Extends Hennessy-Milner Logic
  - Regular formulas (allow the use of sequences of actions in modalities)
  - Fixed point modalities
  - Data and time

\[ \alpha ::= \tau \mid a(t_1, \ldots, t_n) \mid \alpha \mid \alpha \]

\[ \alpha^f ::= t \mid a \mid \text{true} \mid \text{false} \mid \alpha^f \land \alpha^f \mid \alpha^f \lor \alpha^f \mid \forall d : D. \alpha^f \mid \exists d : D. \alpha^f \mid \alpha^f \uparrow u \]

\[ R ::= \varepsilon \mid \alpha \mid f \mid R \cdot R \mid R + R \mid R^* \mid R^+ \]

\[ \phi ::= \text{true} \mid \text{false} \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi \mid \forall d : D. \phi \mid \exists d : D. \phi \mid (R) \phi \mid [R] \phi \mid \Delta^c u \mid \nabla^c u \]

\[ \mu X(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).\phi \mid vX(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).\phi \mid X(t_1, \ldots, t_n) \]
Mapping Reo to mCRL2

- Data flow observed at a channel end = action
- Synchronous channel, synchronous drain
  - $\text{Sync} = A/B . \text{Sync}$;
- Non-deterministic synchronous lossy channel
  - $\text{LossySync} = (A/B + A).\text{LossySync}$;
- Asynchronous drain
  - $\text{AsyncDrain} = (A + B).\text{AsyncDrain}$;
- FIFO1
  - $\text{FIFO1} = A.B . \text{FIFO1}$;
  - $\text{FullFIFO1} = B . \text{FIFO1}$;
  - Alternative encoding: $\text{FIFO1}(b: \text{Bool}) = (\neg b \rightarrow A\ B).\text{FIFO1}(\neg b)$;
- Replication node
  - $\text{Replicator} = X/Y/Z . \text{Replicator}$;
- Merge node
  - $\text{Merger} = (X/Z + Y/Z) . \text{Merger}$;

Synchronize and hide actions corresponding to the connected channel
Adding data to Reo

- **Components/services**
  - Data domain: \( S = \text{struct } c_1(p_{11}:S_{11}, ..., p_{k1}:S_{k1})?r_1/.../c_n(p_{1n}:S_{1n}, ..., p_{kn}:S_{kn})?r_n \)
  - Constraints: salary < 10M

- **Channels in a circuit must accept any data items**
  - \( Data = \text{struct } D_1(e_1: DT_1)?isDT_1/.../D_n(e_2: DT_n)?isDT_n \)
  - Join operation:
    - \( \text{Join} = \sum_{d_1,d_2 \in Data} (X(d_1)|Y(d_2)|Z(\text{tuple}(d_1, d_2)).\text{Join}; \)
  - \( Data = \text{struct } D_1(e_1: DT_1)?isDT_1/.../D_n(e_n: DT_n)?isDT_n|\text{tuple}(e_1: Data, e_2: Data)?isTuple \)

- **Filter conditions**
  - \( (isDT_1(d) \to (e_1(d)==0)) \to ... \)

- **Functions for data transformation**
  - \( f: Data \to Data \)
  - Wrappers: \( c: InputDT_i \to Data \)
  - Projections: \( p: Data \to OutputDT_j \)
  - \( \text{square_sum}(x,y) = z*z \text{ whr } z = x+y \text{ end}; \)

© F. Arbab 2011
Reo with data to mCRL2

**act** $A, B$: Data

- **Synchronous channel**
  - $\text{Sync} = \sum_{d \in \text{Data}} A(d) | B(d) . \text{Sync};$

- **Synchronous drain**
  - $\text{SyncDrain} = \sum_{d_1,d_2 \in \text{Data}} A(d_1) | B(d_2) . \text{SyncDrain};$

- **Synchronous lossy channel**
  - $\text{LossySync} = \sum_{d \in \text{Data}} (A(d) | B(d) + A(d)) . \text{LossySync};$

- **Asynchronous drain**
  - $\text{AsyncDrain} = \sum_{d \in \text{Data}} (A(d) + B(d)) . \text{AsyncDrain};$

- **Filter**
  - $\text{Filter} = \text{sum} \sum_{d \in \text{Data}} (\exp(d) \rightarrow A(d) | B(d) A(d)). \text{Filter}$, where $\exp(d)$ is a boolean expression

- **Transformer**
  - $\text{Transformer} = \sum_{d \in \text{Data}} A(d) | B(f(d)) . \text{Transformer};$

- **Replication node**
  - $\text{Replicator} = \sum_{d \in \text{Data}} X(d) | Y(d) | Z(d) . \text{Replicator};$

- **Merge node**
  - $\text{Merger} = \sum_{d \in \text{Data}} (X(d) | Z(d) + Y(d) | Z(d)) . \text{Merger};$

- **FIFO1**
  - $\text{FIFO1DataFIFO1} = \text{struct empty?isEmpty} | \text{full(e:Data)?isFull}$
  - $(f: \text{DataFIFO1}) = \sum_{d \in \text{Data}} \text{isEmpty}(f) \rightarrow A(d).\text{Fifo1}(\text{full}(d))$
Data flow analysis with mCRL2

- Generated process algebra specification
- Show labeled transition system
- Specify and check formal property
Data Dependent Control Flow

struct el(activated: Bool, amount: Nat)

(amount(d)<1)

(amount(d)==2)

No data

(amount(d)==2)

© F. Arbab 2011
Process verification tools: summary

- **Vereofy:**
  - **Advantages:**
    - Developed for Reo and Constraint Automata
    - Visualization of counterexamples
  - **Disadvantages:**
    - No support for abstract data types
    - Global domain for all components
    - Primitive data constraint specification language (for filter channels)

- **mCRL2**
  - **Advantages:**
    - Support abstract data types including lists and sets
    - Allows the definition of functions
    - Very rich property specification format (mu-calculus)
  - **Disadvantages:**
    - Hard to extract counterexamples
    - For infinite domains model checker often does not terminate (problems with algorithms for formulae rewriting)
ECT Converter Toolset

Architecture of ECT Converters
BPMN to Reo Converter
BPMN to Reo Converter

- A set of pattern matching and transformation rules
- Implemented in the Atlas Transformation Language (ATL)
  - 16 ATL rules
  - 33 helpers
  - Purely declarative, top-down parsing
- Accepts Eclipse BPMN models as input
Sample ATL Rule

- A control flow arrow is modeled in BPMN as a *SequenceEdge* element

  ``` ATL
  rule sequenceEdge2sync
  {
    from
    q : bpmn!SequenceEdge(not(q.source.Fifoable or q.source.IsRuleEvent or q.source.IsXorDb
    or q.source.IsInclusiveOr_Open or q.source.IsTimerEvent or q.source.IsMessageEvent
    or q.IsSpecial))
  
    to
    y : reo!Sync(sourceEnds <- c, sinkEnds <- k),
    c : reo!SourceEnd(node <- q.source),
    k : reo!SinkEnd(node <- q.target)
  }
  ```

- Converts a BPMN as a *SequenceEdge* element into a Reo *Sync* channel
BPMN to Reo Converter - Input
Output in Reo Editor
UML AD to Reo Converter

UML SD Converter

- Accepts UML 2.x SD models as input
- Generates Reo circuits representing the communication protocol
- Can combine SDs for different scenarios and use-cases
- Enables verification and reasoning about the combined protocol
- Originally, a stand-alone tool
- Modified and improved to accept Bouml XMI input
- Support for Eclipse UML2 tool coming
UML SD Editor
UML SD to Reo Converter
BPEL to Reo Converter

- Old converter not integrated in ECT
  - Stand-alone file converter
- New converter is now integrated in the ECT
  - Based on the previous approach
  - Resolves various problems in the previous tool
    - Disjoint nodes resulting from disconnected BPEL structures
    - Large size due to lack of support for components
- Can consider converters from Graphical BPEL and BPEL4People
Compositional Models

(a) Variable

(b) ShiftLossy

(c) Router

(d) Automata
Converting into a Component
BPEL to Reo Converter - Output
Code Generation

- A re-targetable generator produces executable code from constraint automata, yielding centralized controllers that implement coordination protocols of Reo circuits:
  - Java
  - C/C++

- A runtime library implements constraint automata ports as blocking, zero-length buffers.

- A graphical wiring editor lets users connect Java components to the coordinator by pairing corresponding ports together.

- Loading and interpreting constraint automata at runtime are also possible
  - useful for deploying Reo coordinators in Java application servers
Performance Analysis

- Quantitative Intentional Automata (QIA) extend CA with quantitative properties:
  - arrival rates at ports
  - average delays of data-flows between ports
- Quantified Reo circuits are converted to QIA
- Markov Chain models are derived from QIA
  - Resulting Markov Chains are very compact: efficient model checking
- PRISM is used for analysis of MC models
  - average response times
  - meetings of deadlines
  - worst/best case scenarios
Reo Primitives with Delays
QIA for FIFO1
QIA for Sync
QIA for LossySync
QIA for SyncDrain

\[
\begin{align*}
&l_0,\{\}\quad \Rightarrow \quad \{A\}\|\{\}, ((\{A\},\{\},dA)) \\
&B\|\{\}, ((\{B\},\{\},dB)) \\
&\quad \Downarrow \quad dA = dB, \quad \{(\{A\},\{\},dAB)\} \\
&l_0,\{B\} \quad \Rightarrow \quad \{A\}\|\{\}, ((\{A\},\{\},dA)) \\
&C\|\{\}, ((\{C\},\{\},dB)) \\
&\quad \Downarrow \quad dB = dB, \quad \{(\{C\},\{\},dB)\} \\
&l_0,\{A, B\} \quad \Rightarrow \quad \{B\}\|\{\}, ((\{B\},\{\},dB)) \\
&D\|\{\}, ((\{D\},\{\},dB)) \\
&\quad \Downarrow \quad dB = dB, \quad \{(\{D\},\{\},dB)\} \\
&l_0,\{A, B\} \quad \Rightarrow \quad \{B\}\|\{\}, ((\{B\},\{\},dB)) \\
&\quad \Downarrow \quad dB = dB, \quad \{(\{B\},\{\},dB)\} \\
&l_0,\{A, B\} \quad \Rightarrow \quad \{B\}\|\{\}, ((\{B\},\{\},dB)) \\
&\quad \Downarrow \quad dB = dB, \quad \{(\{B\},\{\},dB)\} \\
&l_0,\{A, B\} \\
\end{align*}
\]
QIA of Alternator Reo Circuit
Markov Chain for Alternator
Connector Reconfiguration

- Ordinary data-flow can change the state of a connector, but not its topology.
- (Dynamic) reconfiguration means to change the topology of a connector at (run-time).
- Reconfiguration by graph transformation
  - Perform complex (global) reconfigurations in an atomic step.
  - Definition and validation of reconfigurations as graph grammars using visual tools.
- Reconfiguration model and engine for Reo accessible in ECT through a basic reconfiguration view.
  - Reconfiguration rules used inside the editor for dynamic creation of connectors from templates.
  - Self-reconfiguring connectors triggered by data flow.
Create a Barrier Synchronizer

Figure: reconfiguration rule for a barrier synchronization.
Growing an Exclusive Router

Figure: reconfiguration rule for a \( n \)-ary exclusive router.
Growing a Discriminator

Figure: reconfiguration rule for a n-ary discriminator.
Negative Rules and Regions

Reconfiguration rules can be assigned to disjoint reconfiguration regions (sub-graphs) of a connector.
Analysis and Verification

- Formal analysis and verification of the properties of rules and consequences of their applications

Figure: an example network
Add/Delete Worker

DelWorker is defined as the inverse rule.

Figure: example reconfiguration rule: AddWorker
Add/Delete Resource

Figure: example reconfiguration rule: \textit{AddResource}

\textit{DelResource} is defined as the inverse rule
Export to AGG

- http://tfs.cs.tu-berlin.de/agg

Figure: exported rule AddWorker
Analysis in AGG

• Critical-Pair analysis:

Figure: Results of a critical pair analysis in AGG
Model Checking in GROOVE

- \(\text{AG(StartWorker} \rightarrow \text{EX(StopWorker)})\): After starting a worker it can always be stopped again in the next step.

- \(\text{AG(AddResource)}\): It is always possible to add a resource.

- \(\text{AG(StopWorker} \rightarrow \text{EX(AddResource)})\): After stopping a worker it is always possible to add a resource in the next step.
Conclusion

- Making interaction explicit in concurrency allows its direct
  - Specification
  - composition
  - Analysis
  - Verification
  - reuse

- Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of pure (coordination or concurrency) protocols.
  - Looser interdependencies and strict separation of concerns.
  - Unique emphasis on interaction, as (the only) first-class concept.
  - Free combination of synchrony, exclusion, and asynchrony, as relational constraints simplifies definition of interaction protocols and atomic transactions.
  - Exogenous interaction/coordination

http://reo.project.cwi.nl