Elements of Interaction

Farhad Arbab

Foundations of Software Engineering

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

28 October 2010
Complex Systems Design & Management
Paris, France

Complex Systems Design & Management

- 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 Design & Management Complex? How?
- 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?

Concurrency and Interaction

- Interaction is the subject of interest in concurrency
  - "Elements of Interaction"
    Robin Milner, Turing Award Lecture, 1991.
- How formal languages and models of concurrency represent interaction?

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

```
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

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

- Producers

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

```
while (true) {
  sleep (3000);
  redText = ...
  redSemaphore.acquire();
  bufferSemaphore.acquire();
  buffer = redText;
  bufferSemaphore.release();
  greenSemaphore.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

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 ai of process A collides with action bj 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 be
  - Manipulated?
  - Verified?
  - Debugged?
  - Reused ?
  - …

Possible only indirectly, through manipulating processes

Elements of Interaction

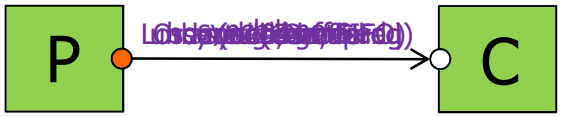
- **Coordination** is **constrained interaction**: it constrains interaction protocols among communicating software components.
  - "Coordination as Constrained Interaction" Peter Wegner
- What is an interaction protocol?
  - Synchrony / asynchrony
  - Atomicity
  - Ordering
  - Exclusion
  - Grouping
  - Selection
  - …
- How to formalize interaction explicitly?
  - **Constraints**

Interaction Based Concurrency

- Make interaction explicit!
- 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:
    - get(p, v) or get(p, v, t)
    - put(p, v) or 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.

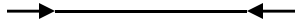
Channels

- 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.

Primitive Channels

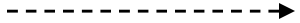
- Synchronous channel
  - write/take

- Synchronous drain channel
  - write/write

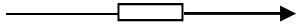


- Synchronous spout channel
  - take/take

- Lossy synchronous channel
  - write/take

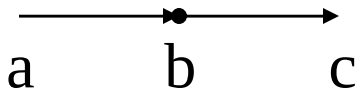


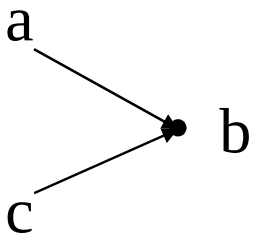
- Asynchronous FIFO1 channel
  - write/take

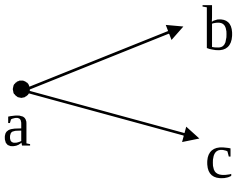


Join

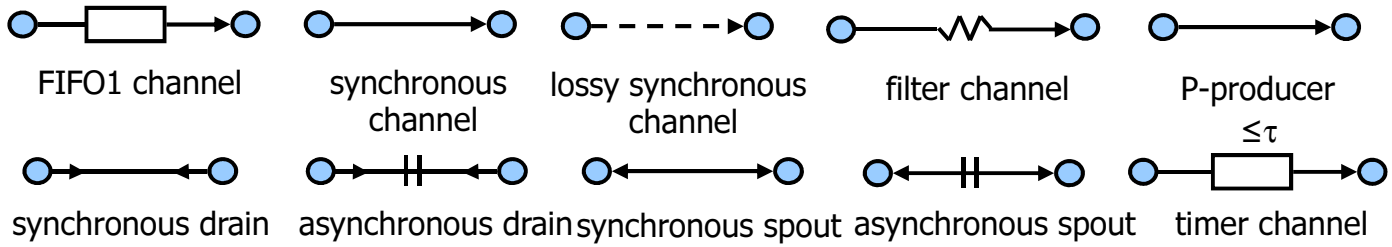
- Mixed node
  - Merge + replication combo
- Sink node
  - Non-deterministic merge
- Source node
  - Replication

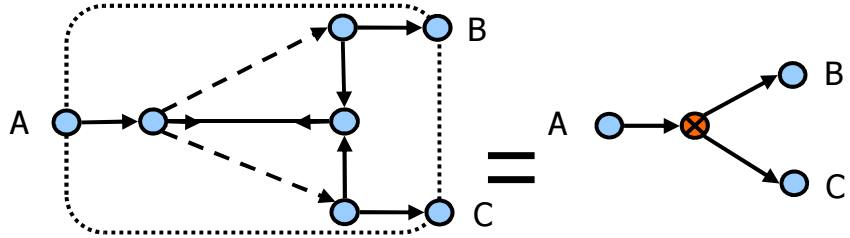




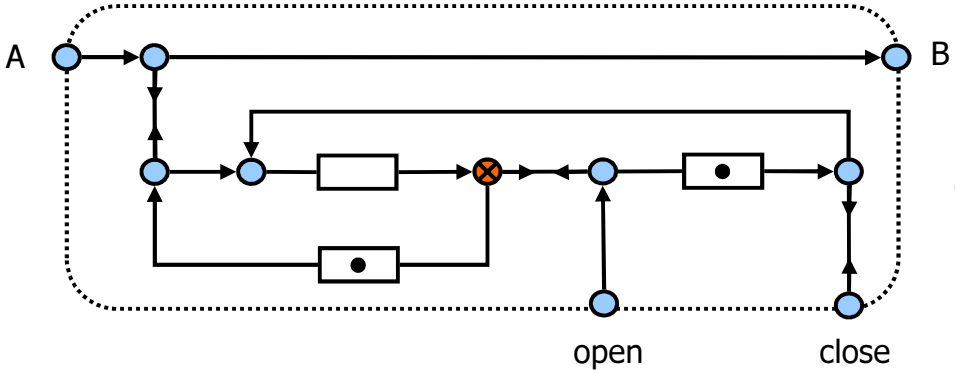


Reo Connectors





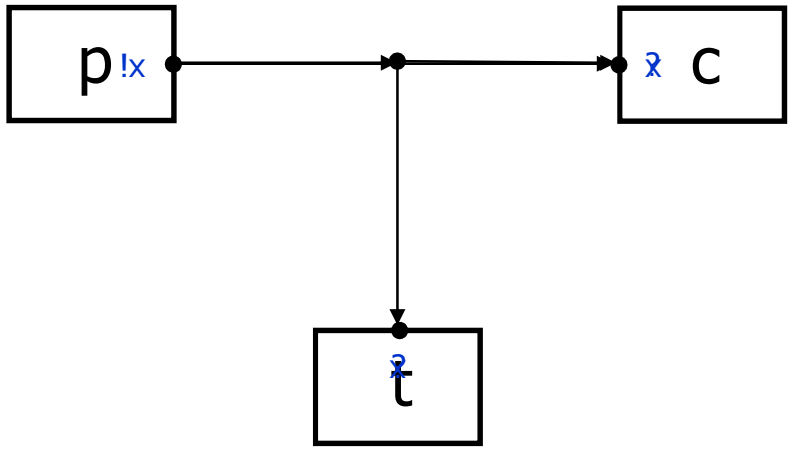
Exclusive choice (deffered XOR)



Valve connector:
controls flow from A to B

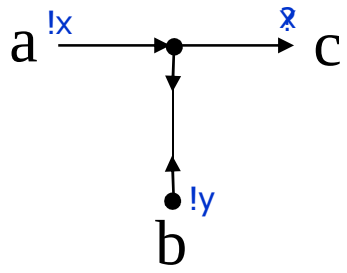
A Simple Composed System

- Read-cue synchronous flow-regulator



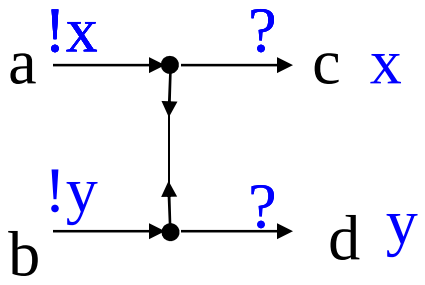
Flow regulator

- Write-cue synchronous flow-regulator



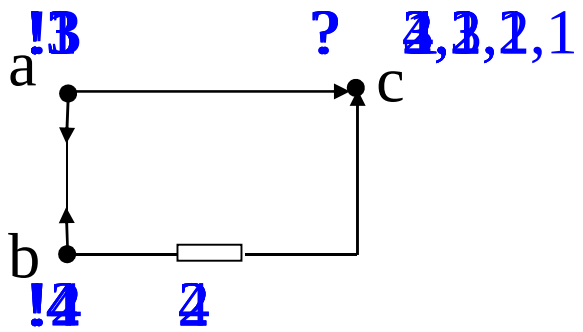
Flow Synchronization

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



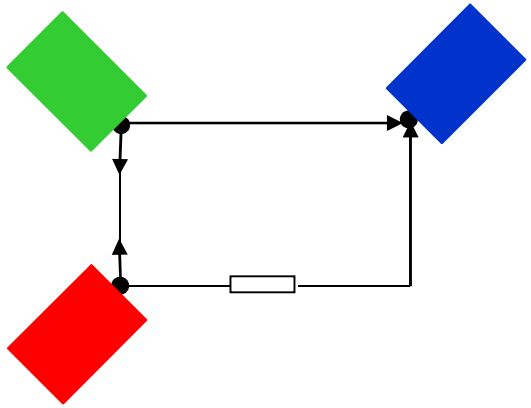
Alternator

- Subsequent takes from c retrieve the elements of the stream (ab)*
- Both a and b must be present before a pair can go through.



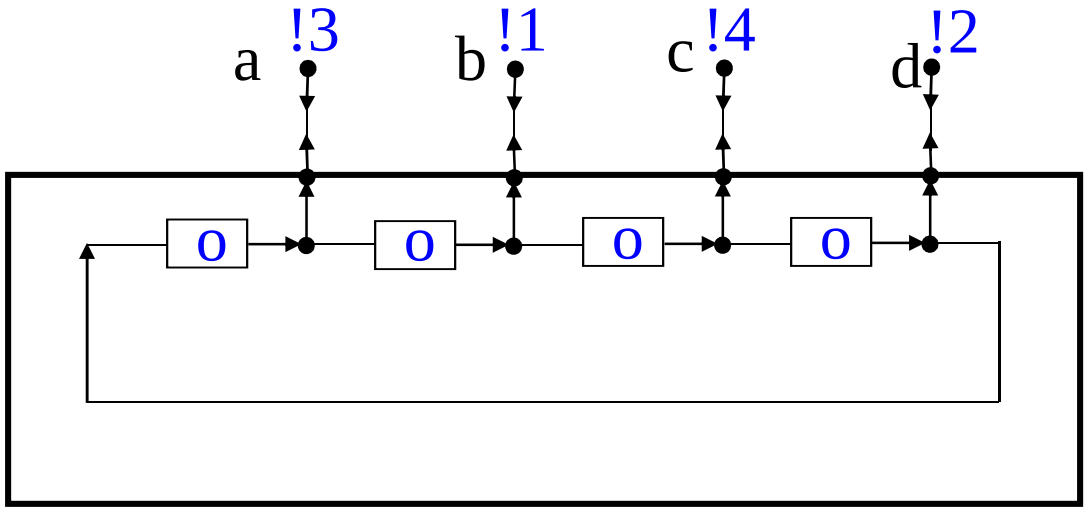
Alternating Producers

- We can use the alternator circuit to impose the protocol on the green and red producers of our example
  - From outside
  - Without their knowledge



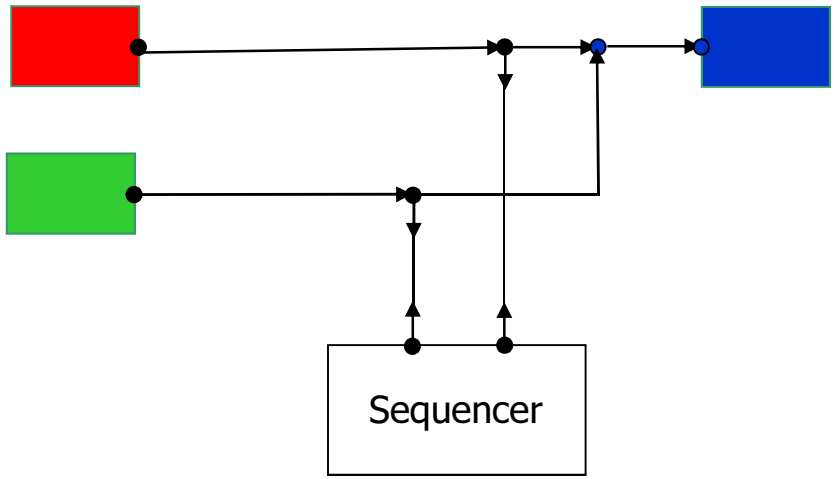
Sequencer

- 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.

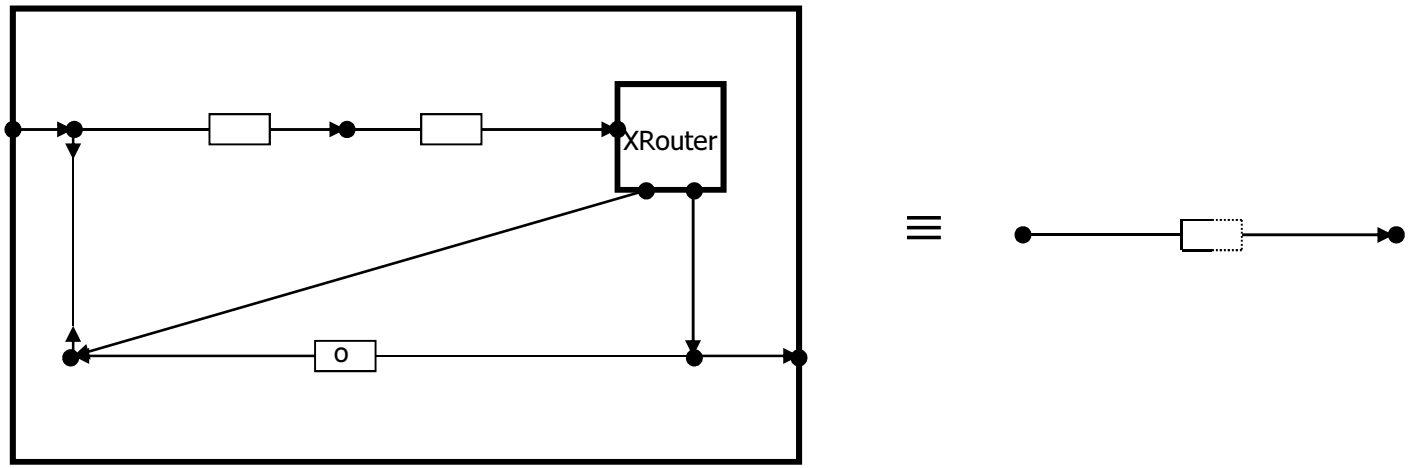


Overflow Lossy FIFO1

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

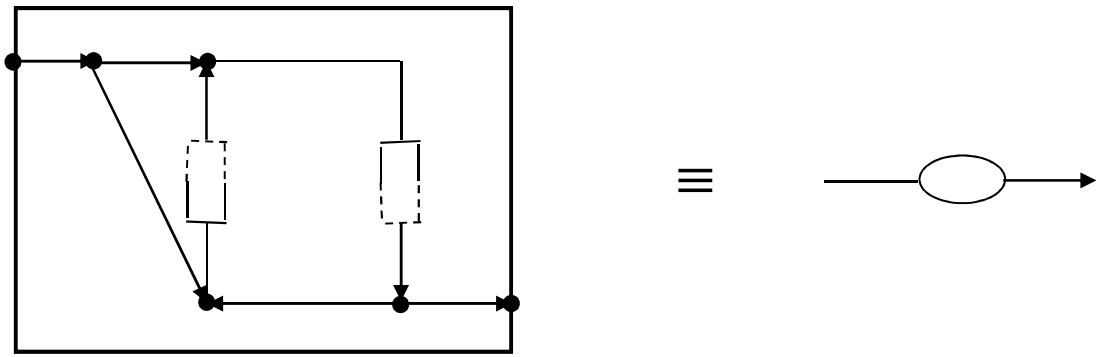
Shift Lossy FIFO1

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



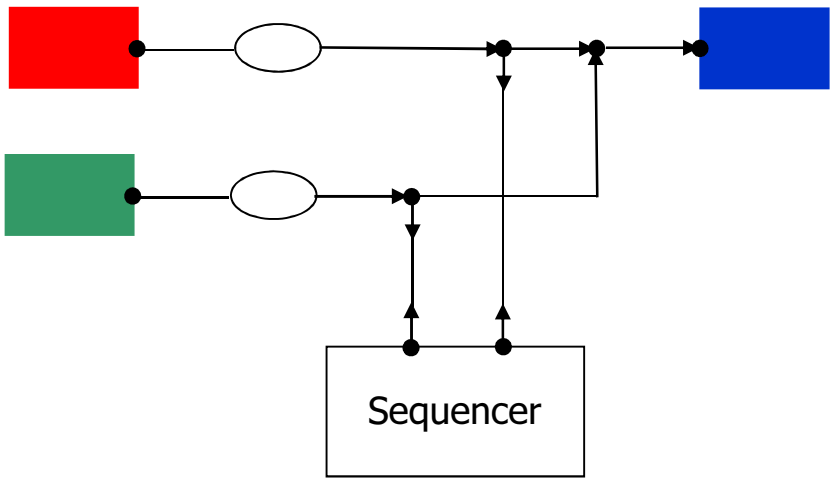
Variable

- 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.



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

Eclipse Coordination Tools

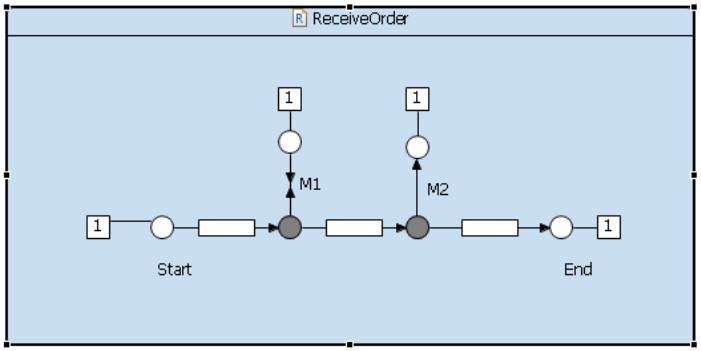
- A set of Eclipse plug-ins provide the ECT visual programming environment.
- Protocols can be designed by composing Reo circuits in a graphical editor.
- The Reo circuit can be animated in ECT.
- ECT can automatically generate the CA for a Reo circuit.
- Model-checkers integrated in ECT can be used 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

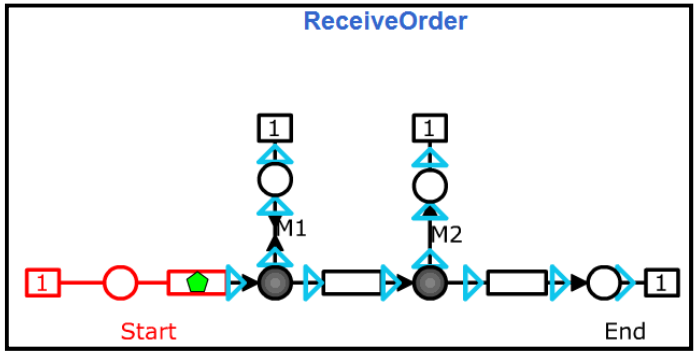
Tool support

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

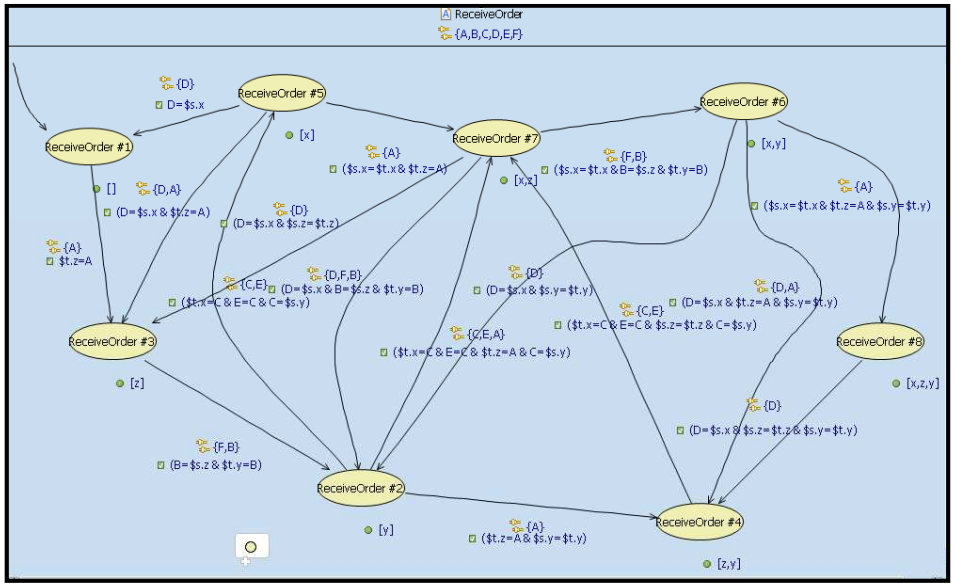
Tool snapshots



Reo graphical editor



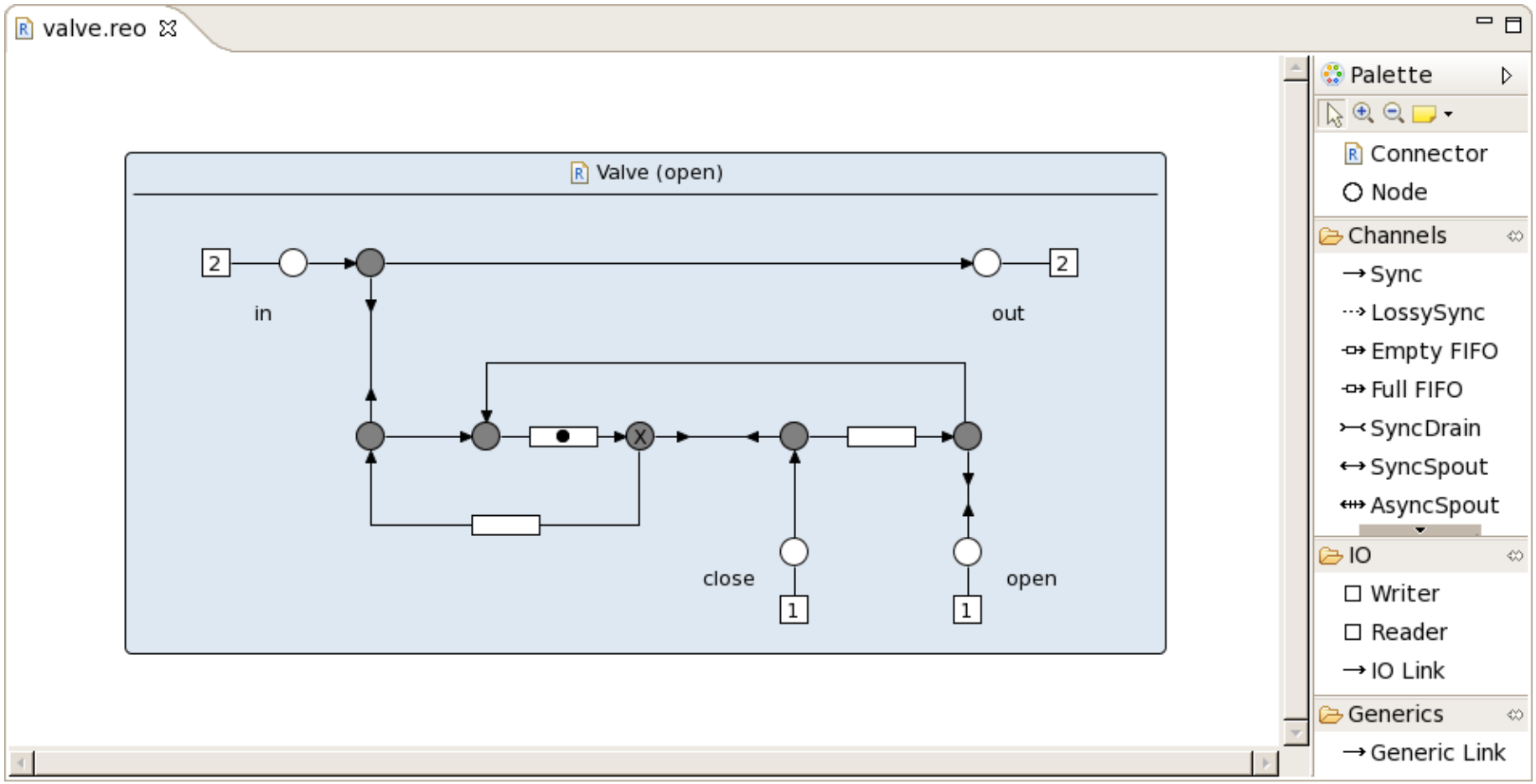
Reo simulation plug-in



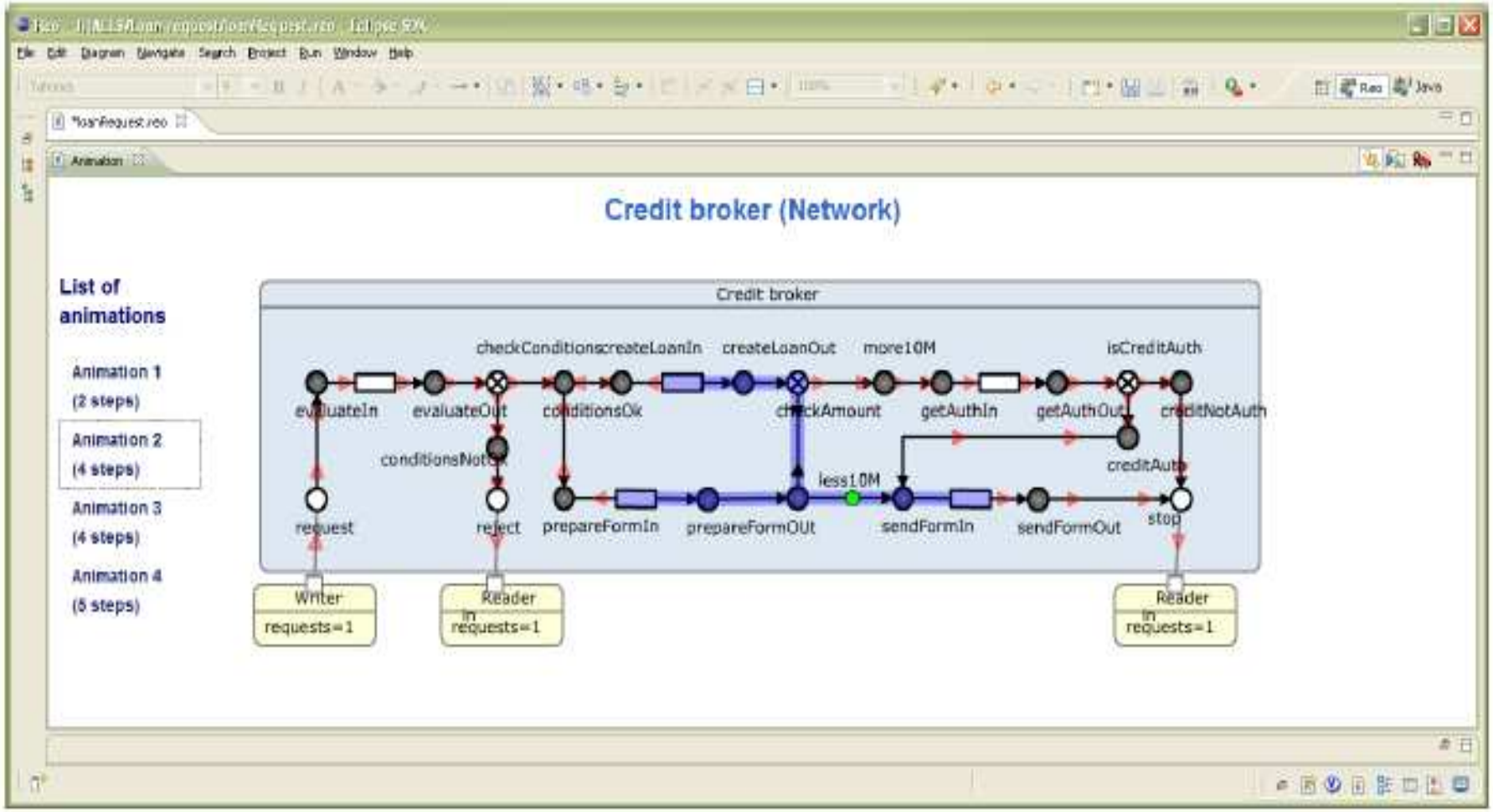
Reo to constraint automata converter

http://reo.project.cwi.nl

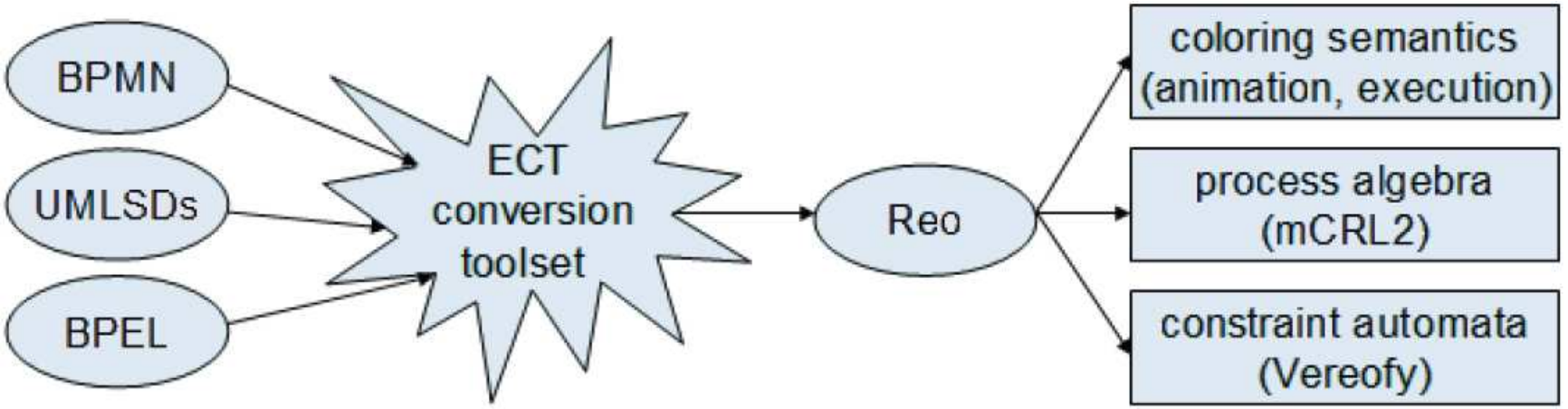
Snapshot of Reo Editor



Reo Animation Tool



ECT Converter Toolset

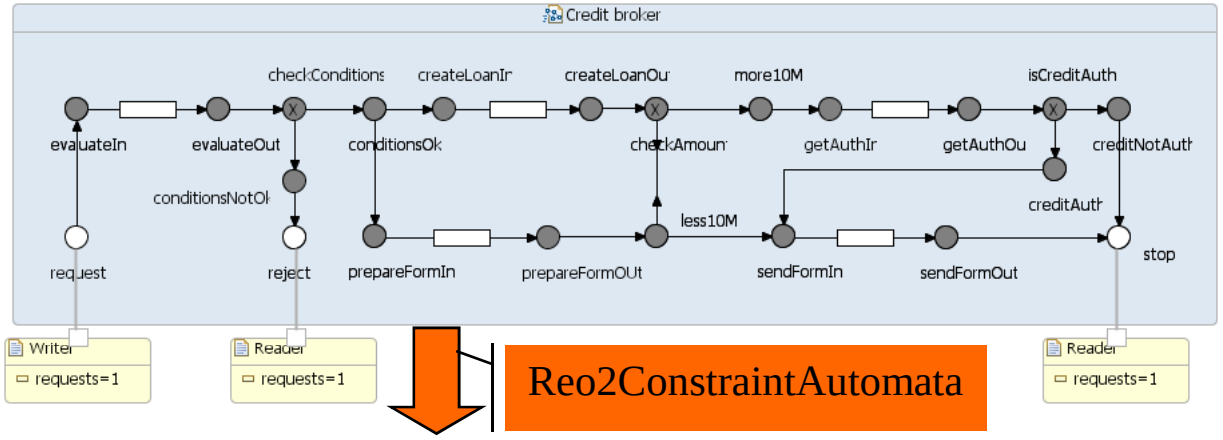


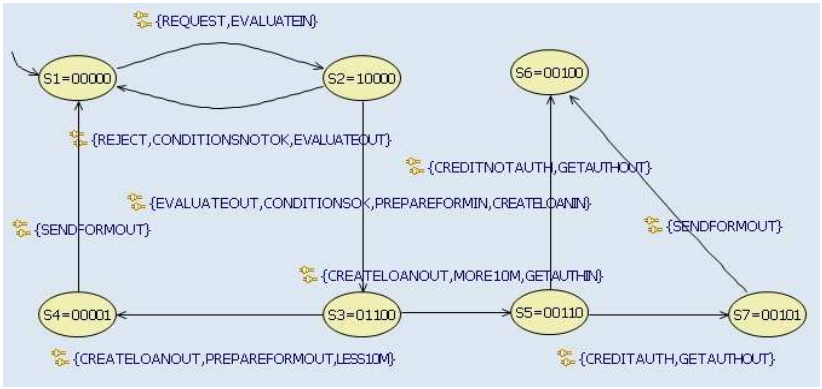
Changizi, B., Kokash, N., Arbab, F.: *A Unified Toolset for Business Process Model Formalization*, International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA'2010), tool demonstration paper

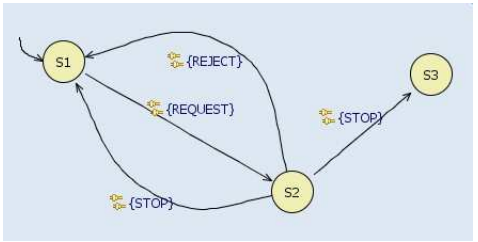
Model Checking

- Constraint automata are used for model checking of Reo circuits
- Vereofy model checker (http://www.vereofy.de)
  - Developed at University of 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
  - Developed at TU Eindhoven (http://www.mcrl2.org)

Verification with Vereofy



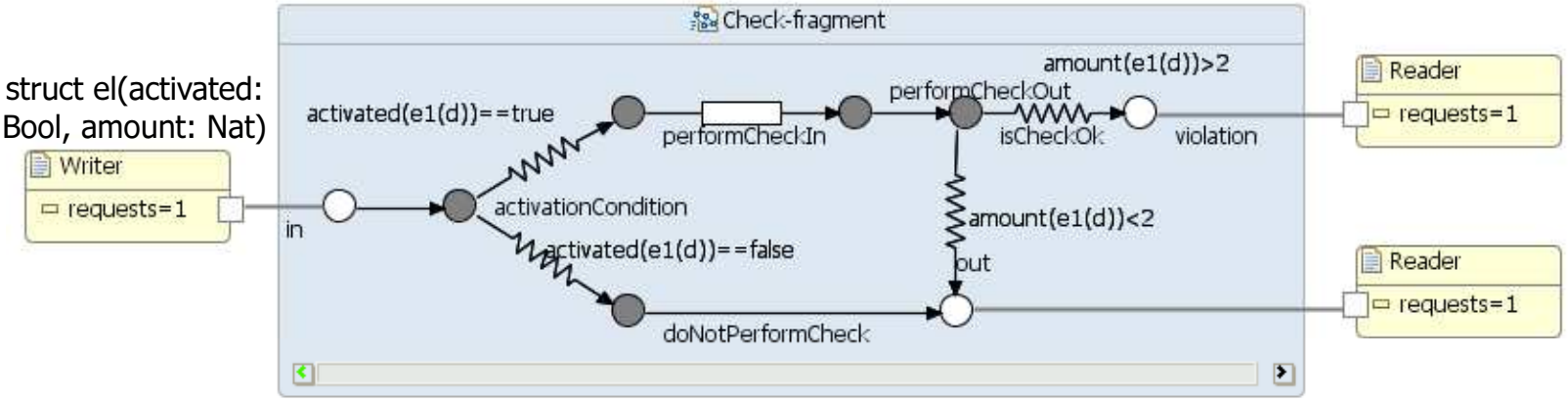


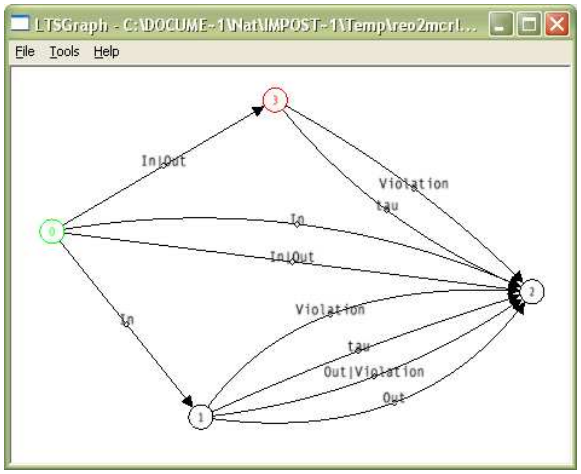


- Modal formulae
  - Branching time temporal logic: *AG*[EX[*true*]] – check for deadlocks
  - Linear temporal logics: **G**(*request* → **F** (*reject* ∪ *sendFormOut*)) – check that admissible states *reject* or *sendFormOut* are reached

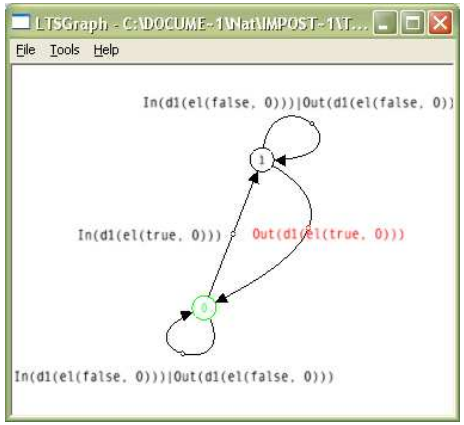
Data Dependent Control Flow

struct el(activated: Bool, amount: Nat)

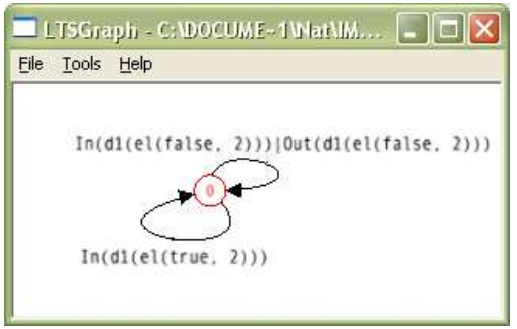




No data



(amount(d)<1)



(amount(d)==2)

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