πάντα ῥεῖ
Heraklit

Components for Scientific Work Flows

Th Letschert

Background :

The *Parsuite* Project http://www.parasuite.com

PARASUITE

A data flow based system for analyzing engineering data especially for – masses of – product life cycle data

Aim: Reduction of maintenance costs of industrial equipment

- 2007 – 2009 Research Project in Cooperation with Univ. of Marburg, and CogniData
- Now a product of CogniData

This Talk

Some Ideas for a follower project based on the experience with the Parasuite project.

Background – the Parsuite Project

PARASUITE

Architecture

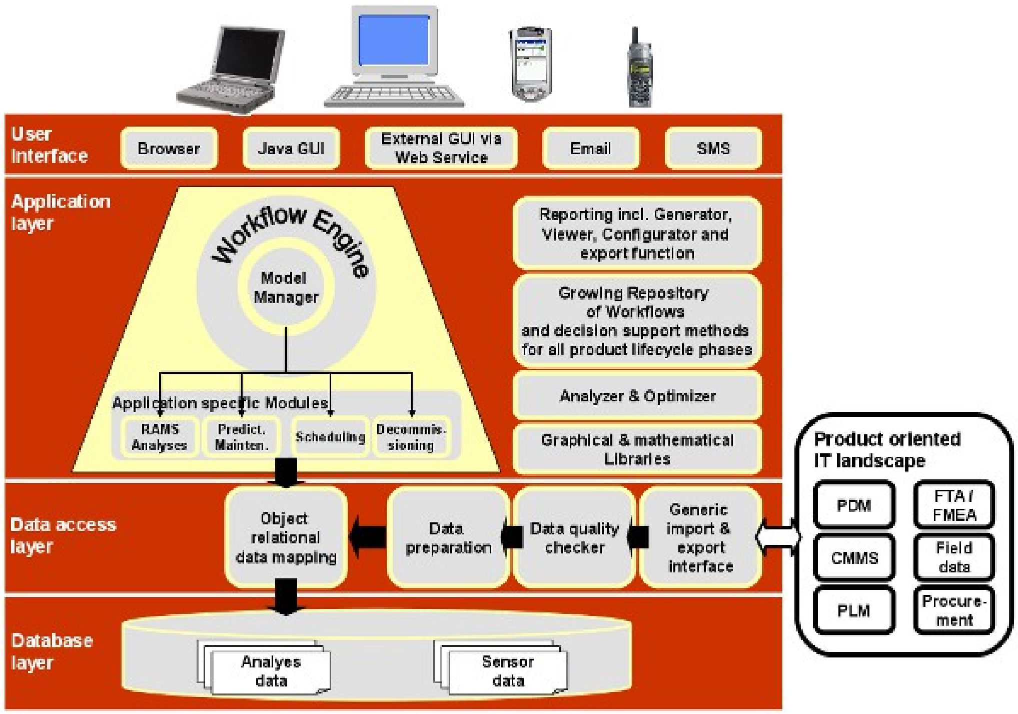
- Frontend
- Data importer
- Data base
- Analyzing modules

Implementation

- Java
- JBoss
- MySQL
- Eclipse RCP

Applications

- Data-flow / work-flow programs analyzing the data



Background – the Parsuite Project

Experience:

Writing Data Flow Programs is unexpectedly hard

Concurrency and algorithmic issues are entangled

Multi-threading means of the implementation platform are easily under- or overused (too much / not enough threads)

Parasuite applications are not intended to be written by end users, but even for IT professionals it is not as easy to write applications as expected.

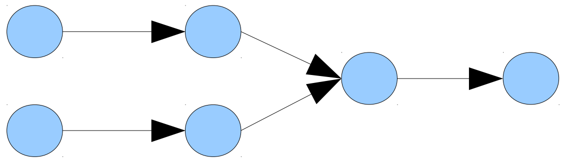
The Flow Based Programming (FBP) Paradigm

Applications are seen as networks of asynchronous processes communicating by means of streams of data chunks that flow through channels of finite capacity.

→ *wikipedia*

Features

- popular in the area of data analysis, data mining
- Based of generic components used as black-boxes
- Well suited for "graphical programming" (by non-IT-lers)
- Typical components: read, count, merge, sort, transform, …

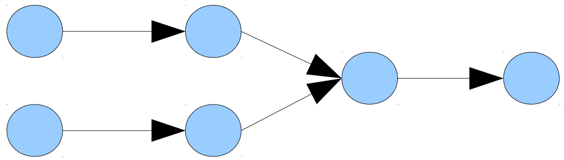


The Flow Based Programming (FBP) Paradigm

Data flow systems are well suited for analyzing scientific data

because they support several goals:

- **Visualization**
  - **Algorithms are presented as systems of computing nodes connected by channels**
- **Parallelism**
  - **Nodes work concurrently / in parallel**
- **Modularity**
  - **Nodes may be reused**



However . . .

Flow Based Programs: Problems

The FBP Paradigm : Modularity and Parallelism

- **Diverging aims**
  - **User friendlessness (Visualization, Modularity):**
    Nodes represent reusable algorithms
  - **Implementation (Parallelism):**
    Nodes represent units of parallel / concurrent work
- **Problem**

  Algorithms as units of work and
  algorithms as units of concurrent / parallel evaluation
  are not the same and thus should not be identified:
  - **Nodes as units of work should be much more coarse-grained than units of parallelism**
  - **Parallelism should in general not be introduced by the user**
  - **Nodes as units of concurrent work should be fine grained and the granularity should be adopted automatically to the platform**

Flow Based Programs: Problems

FBP Paradigm: Data

- **Data flow: Flow of unstructured atomic data**

  However: Scientific data often have a rich structure

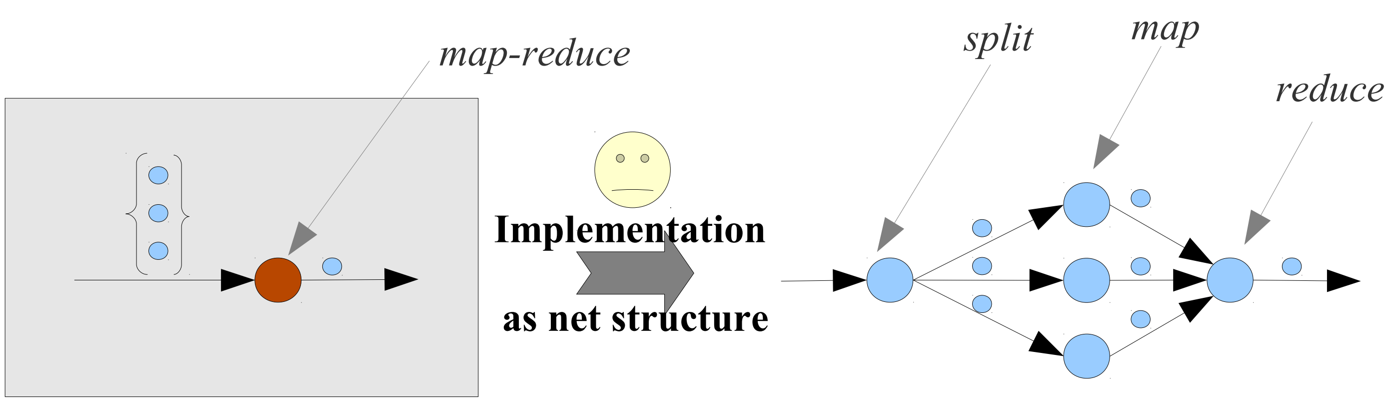
Flow Based Programs: Problems

Problem

Entangled mix of connections and nodes that were introduced for different purposes:

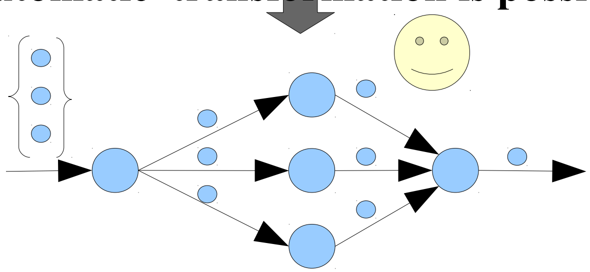
- **Nets structure is used as a remedy for missing data structures**
- **Net structure is a consequence of parallelism that was introduced “by hand”**

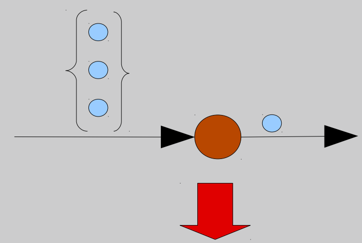
FBP-Problem: Example Map/Reduce



Concept: Map-Reduce-Algorithm

Automatic transformation is possible





The algorithm should appear in this form within the net

Implementation of the node should employ concurrency / parallelism features depending on the actual means and resources of the executing platform

Change of view:

- ***Restrict the FBP paradigm to modularization and visualization.***
- ***Treat implementation as a different issue: The user sees a net of concurrently working nodes, the reality of the implementation however may be different.***

- **Decouple low level aspects of parallelism and concurrency and algorithmic abstraction**
  - **Nodes**

    are used as coarse-grained units of algorithmic abstraction
  - **Parallelism and concurrency**

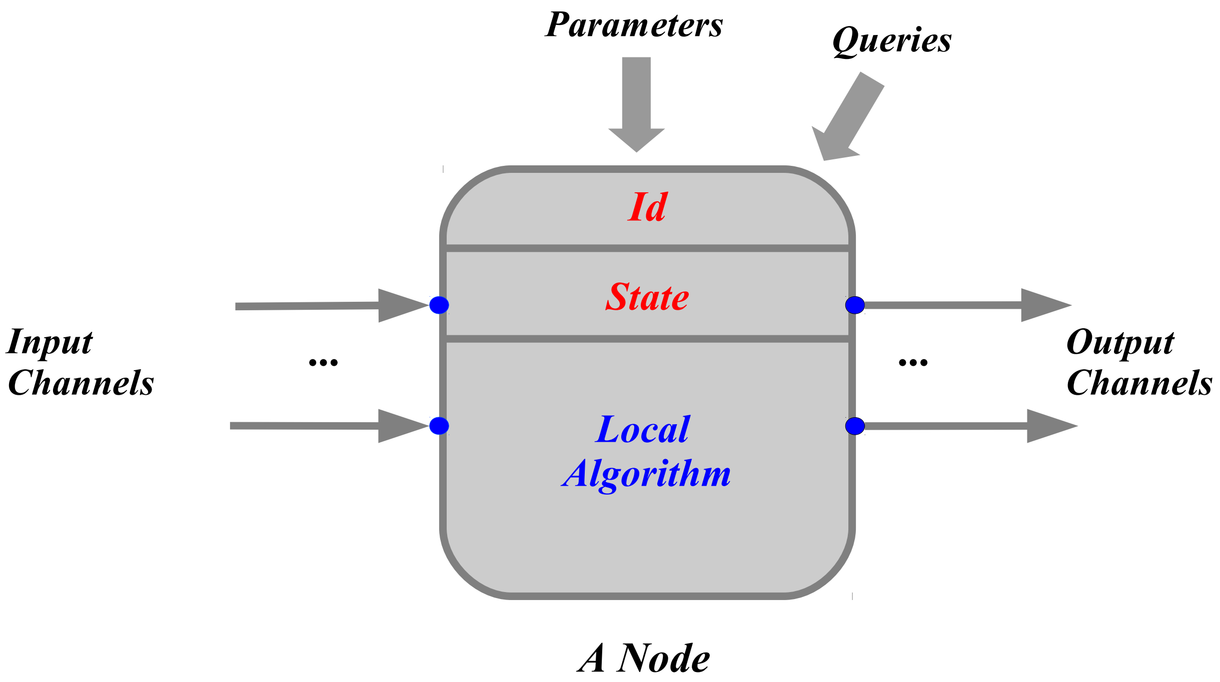
    are dealt with primarily at the node level
    - **Based on the node's task**
    - **Automatically and transparently**
    - **Adapted to the actual platform**
    - **Responsibility of the implementation**
- **Use structured data**
- **Use normalized high level control structures**

Concept of Nodes and Node Types

Nodes

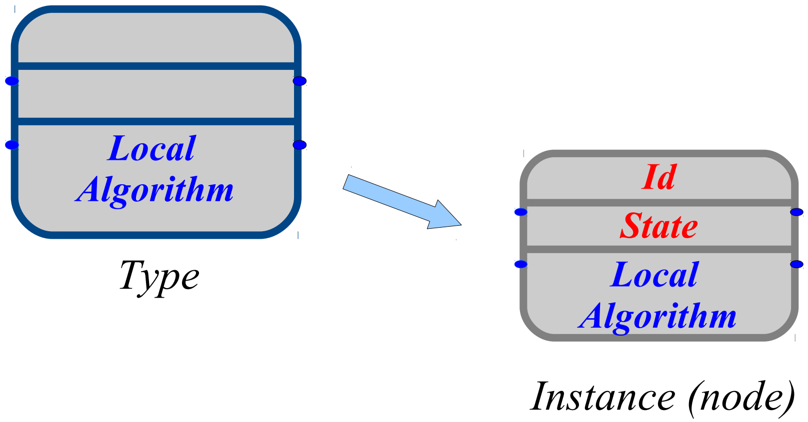
- Primarily a means of modularization
- Algorithmic abstraction
- Embedded in a data flow

- ***Id***: The id of the node (instance of the type)
- ***State*** : current state of the node
- ***Ports***: attach points for channels
- ***Local Algorithm*** executed on input channels
- ***Parameters***: Initialization/Instantiation parameters
- ***Queries***: supply information about the instance



Nodes and Node Types

- **Nodes**
  - are created as instantiation of node types
  - have an actual state
- **Node types**
  - have a name
  - define ports,
    parameters, queries,
    the algorithm, the variables that make up the state



Nodes and Node Types

Example: Node Type and Node

```
class Adder implements Runnable {
  def v1 = 0
  def v2 = 0

  @INPORT(type="Integer")
  InPort inP1

  @OUTPORT(type="Integer")
  InPort inP2

  @OUTPORT(type="Integer")
  OutPort outP

  @QUERY(type="Integer")
  def queryA() {
    return v1 + v2
  }

  void run() {
     while (true)
       if (inP1.isClosed()) break
       if (inP2.isClosed()) break
       v1 = inP1.receive();
       v2 = inP1.receive();
       outP.send(v1+v2);
     }
     inP1.close()
     inP2.close()
     outP.close()
  }
}
```

The node type Adder

- (here) written in Groovy
- Ports and queries are marked by annotations
- Algorithm is defined as implementation of the interface Runnable

```
<tns:node id="adder" type="Adder" />

<tns:connection>
    <tns:from node="producer1" port="outP"/>
    <tns:to node="adder" port="inP1"/>
</tns:connection>

<tns:connection>
    <tns:from node="producer2" port="outP"/>
    <tns:to node="adder" port="inP2"/>
</tns:connection>

<tns:connection>
    <tns:from node="adder" port="outP"/>
    <tns:to node="consumer" port="inP"/>
</tns:connection>
```

The node adder as instance of Adder

- defined in XML
- with id and connection of it's ports to channels

Data Concept

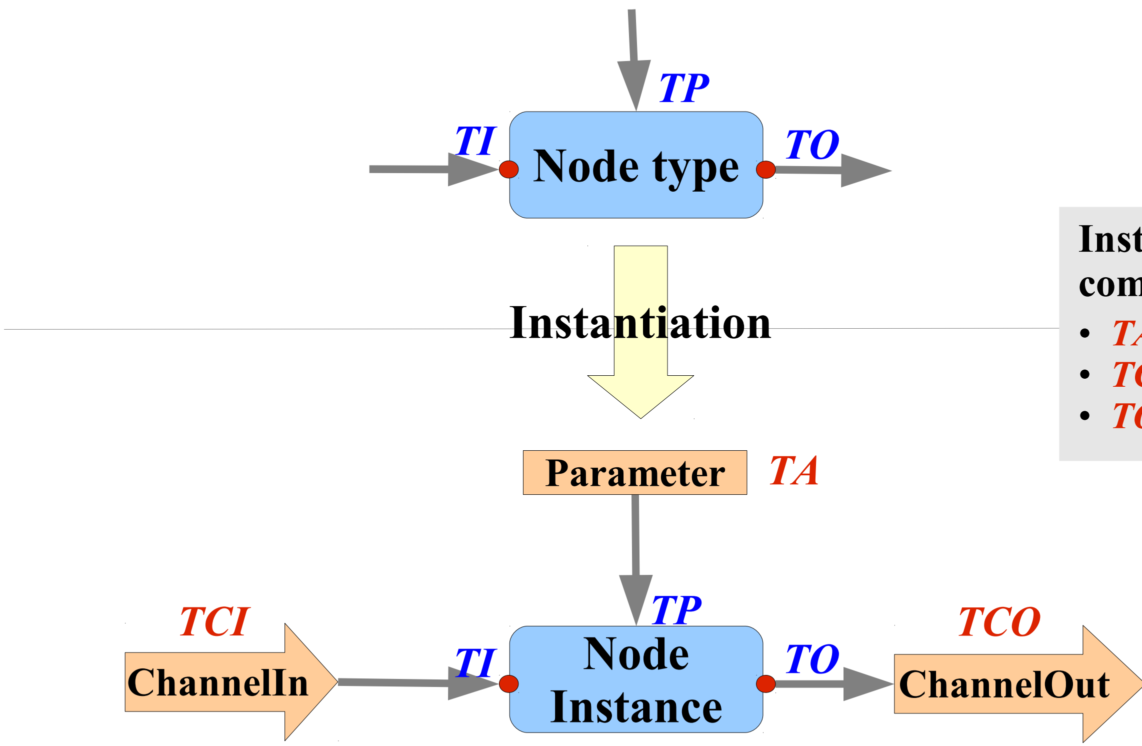
Requirements

- **Simple**
  - Usable by non computer scientists
  - "Natural" data modeling
- **Adequate**
  - Data of the application domain may be represented
- **Static checks**
  - Ports, parameters etc. should have types
  - Connections of nodes via ports should be checked before run-time
- **Flexible**
  - Typing should be flexible to allow for generic node (types) with a wide application range

Requirement : Subtype ordering

Types should have a subtype ordering to allow for a statically checked generic node types

Type



Instantiation is possible if type compatibility holds:

- $TA \leq TP$
- $TCI \leq TI$
- $TO \leq TCO$

Instance

Types and type ordering

The type system should <u>not</u> be based on OO principles

- **OO-based type systems are to provide flexibility combined with checkablity**
- **However: OO-based type systems are unnecessarily complex**
- **A record based type system is sufficient**

Types and values

Values:

- Atomic Values
  - Numeric (int, double, ⋯)
  - String
  - Date
  - ... to be evaluated ...
- Structured values
  - List (ordered sequence of values)
  - Map (Value → Value)
  - Record (Selector → Value)

Types:

- Representing these values

Type Definitions

- **Atomic Types**
  - **Int, Double, String, Date**
- **Type Constructors**
  - **List**: if T is type then so is **List(T)**
  - **Record**: if T1, ··· Tn are Types and s1, ··· sn are Identifiers

    then **Record(s1:T1, ··· sn:Tn)** is a type
  - **Map:** if T1 and T2 are Types

    then Map**(T1, T2)** is a type
  - **Type Identifier**

    Identifiers may be defined to denote types
- **Recursion**
  - Recursive type definitions are not allowed

Subtype relation

- Usual conversion order on atomic types
- Conversion to String possible for any value
- Covariance on Lists:

  $T \leq T' \Rightarrow List(T) \leq List(T')$
- Covariance on Records

  $T \leq T' \Rightarrow Record(\cdots s{:}T \cdots) \leq Record(\cdots s{:}T' \cdots)$
- Covariance on Maps

  $T \leq T' \Rightarrow Map(T \rightarrow T'') \leq Map(T' \rightarrow T)$

  $\Rightarrow Map(T'' \rightarrow T) \leq Map(T'' \rightarrow T')$
- Record extension

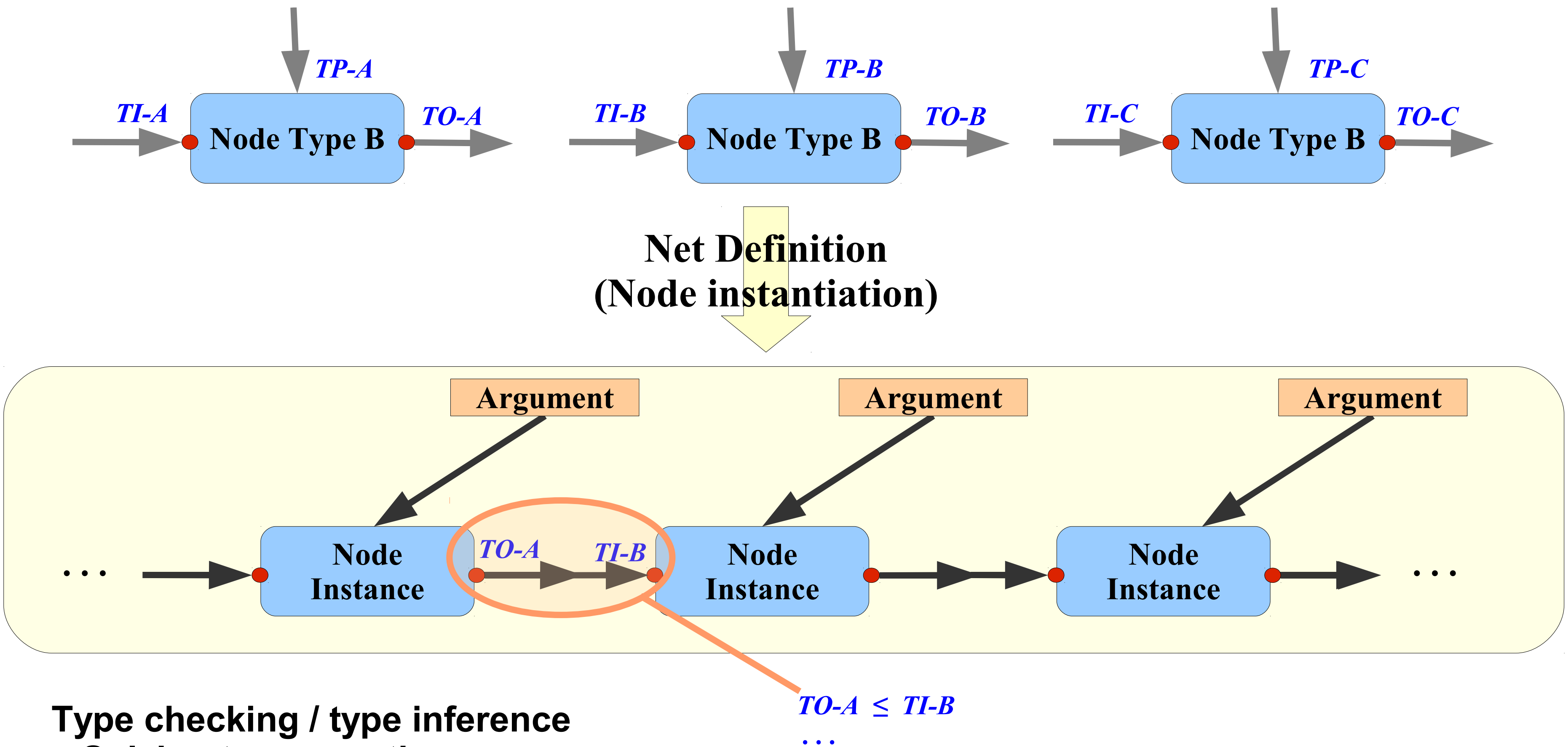
  each extension of a record type is compatible with the not extended record

  $Record(\cdots s{:}T \cdots) \leq Record(\cdots \; \cdots)$

Note The type system is structural and not-nominal. I.e. the subtype relation relies only on the structure of the types

Data Concept

Type checking of nets



Type checking / type inference

Solving type equations

Open issue

Two forms of nodes in flow nets

- **Active nodes** (pull / push nodes)
  - Perform (blocking) reads on input-ports
  - Perform (blocking) write operations on output-ports
  - Need a "private" thread for execution
  - Suited for data processing tasks (all data are available)
- **Reactive nodes** (passive / push nodes)
  - Do not read actively
  - React on data available on (sets of) input-ports
  - Do not need a "private" thread
  - Suited to "real time" computations processing data streams
- A net consisting solely of reactive nodes is passive and does nothing: every node waits for ever
- A net consisting solely of active nodes will easily overuse the platform's multi-threading means
- Combining active and passive nodes in one net:
  - Realizable and worthwhile ?

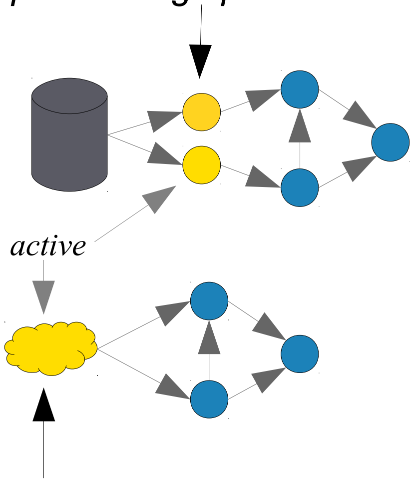
Combining active and passive nodes – is it worthwhile ?

- **No: There are two distinct forms of nets that should not be mixed or confused:**
  - **Pulling Nets (task parallel nets):** a net processes available data

    data reading nodes have to adapt to the speed of the processing nodes
  - **Streaming Nets (data parallel nets):** a net that processes data as they come in real time.

    The processing nodes have to process input by any means as it comes in
- **Yes: Active nodes only is too expensive**
  - Active nodes need a dedicated thread
  - Threads are a limited resource
  - Net structure and threading should be decoupled as much as possible
  - Even in “Pulling Nets” there are a lot of task that do not net a dedicated thread because they are stateless and purely reactive

Reading nodes, pushing data into the net, have to and can synchronize on processing speed



“External world”, will not synchronize on processing speed

Concurrency: 2 Types of nodes

Combining active and passive nodes – Realization options

Method 1: Transparent reactive nodes

- Reactive nodes are introduced automatically
- Their existence is not visible to the user

Method 2: Non-Transparent reactive nodes

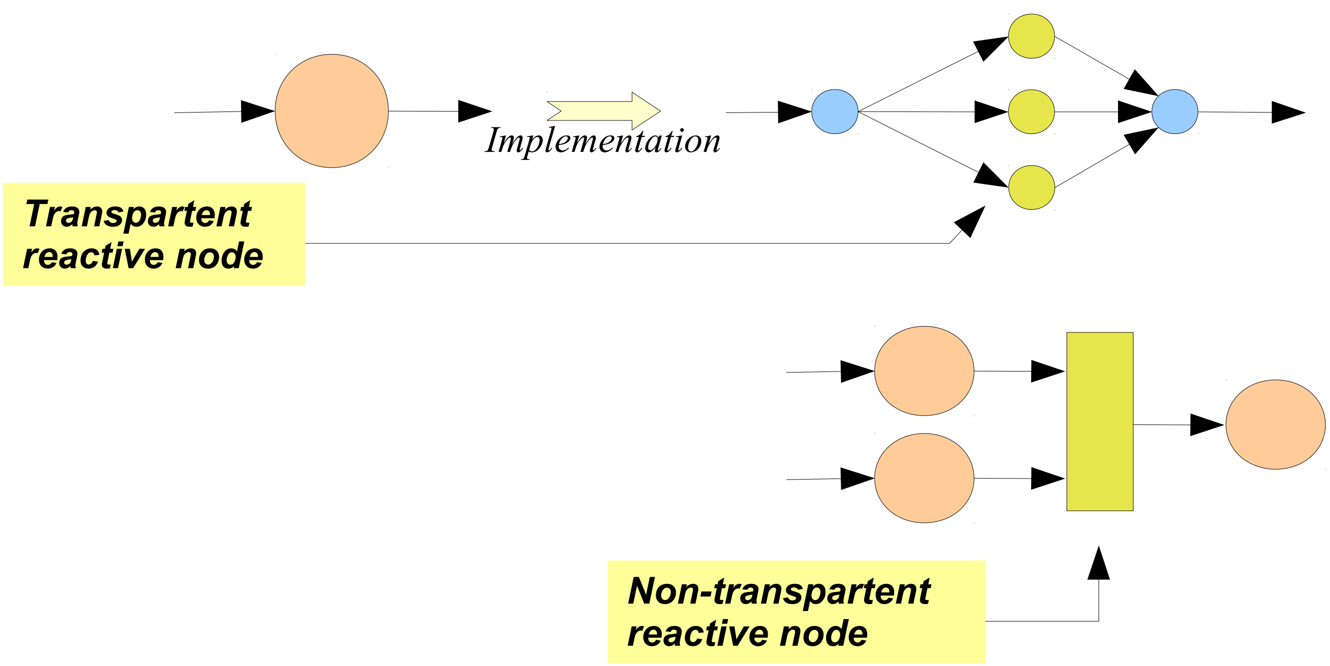
- Reactive nodes are introduced by the user

Combining active and passive nodes – Realization options

Actual Investigation:

Is it worthwhile to provide transparent and non-transparent reactive nodes?

I.e. is it easy and beneficial to use them, and
may their combination be implemented with moderate effort ?



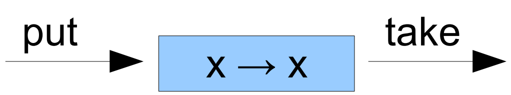
“Heavy” algorithmic nodes are transparently implemented using sets of reactive nodes.

Non-transparent reactive nodes are introduced by the user

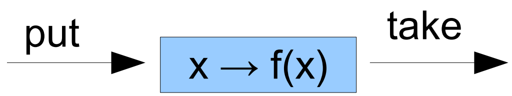
Combining active and passive nodes – Implementation

Observation:

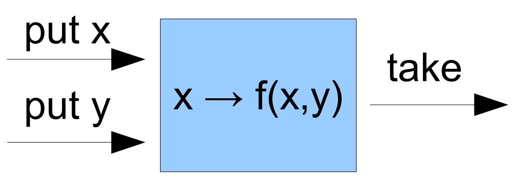
- Channels (synchronized buffers) are passive nodes
- A channel represents the identical function
- They may be extended to compute functions
- They may be extended to compute functions on several input values
- The notion “channel” is not appropriate: we have stream-transformers



A channel = identical stream transformer



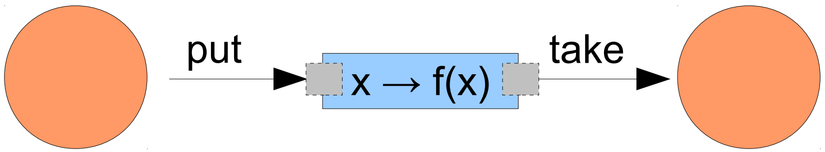
A channel with a function: a stream transforming function



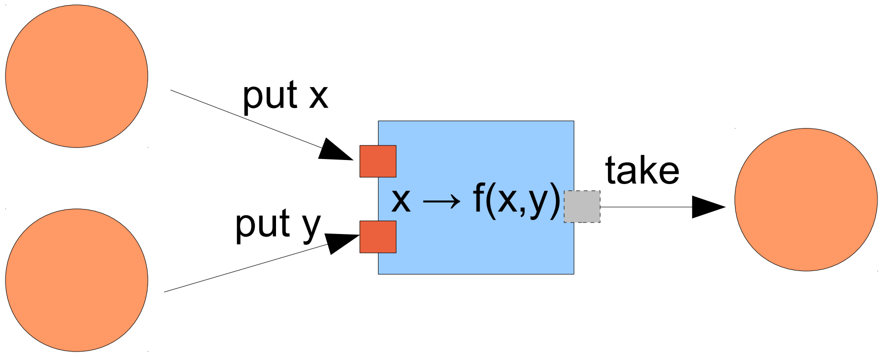
A function that transforms two streams

Combining active and passive nodes – Implementation

- **Transformers with one input just act like synchronized buffers that modify their values**
- **Transformers with more than one input have to synchronize on all inputs e.g. by an internal buffer on each input**



Buffering is optional

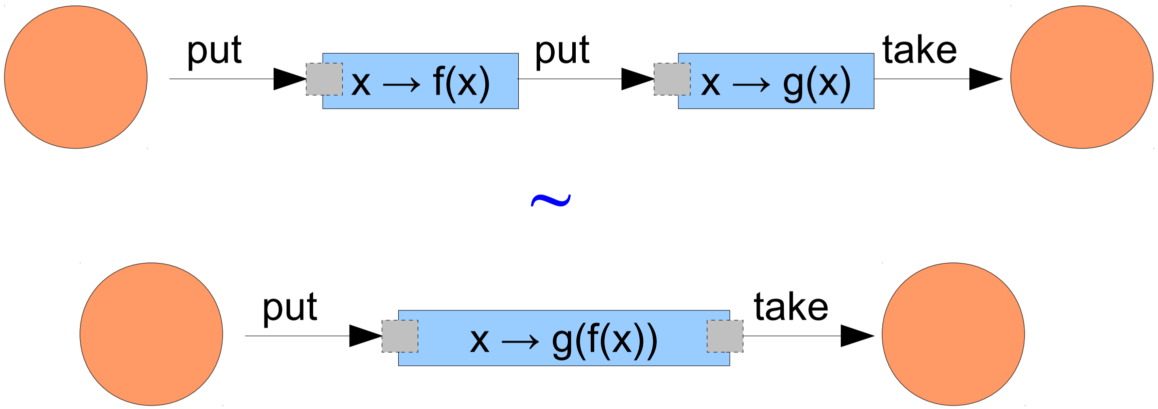


A buffer capacity of at least one on the input side is mandatory

Th Letschert

Chaining of transformers

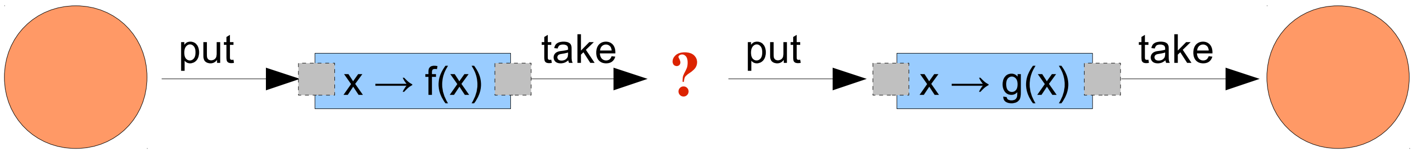
Does chaining of transformers make sense ?



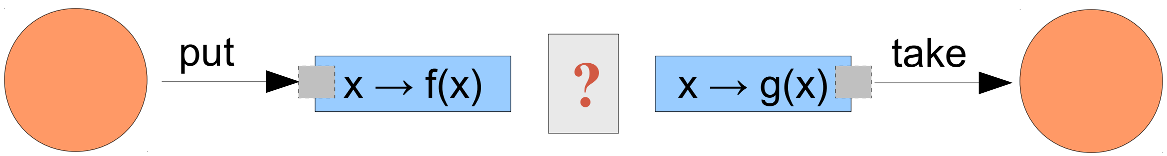
Obviously both versions are computational equivalent, but the first version suggests that f and g may be computed concurrently.

Chaining of transformers

Is chaining possible ?



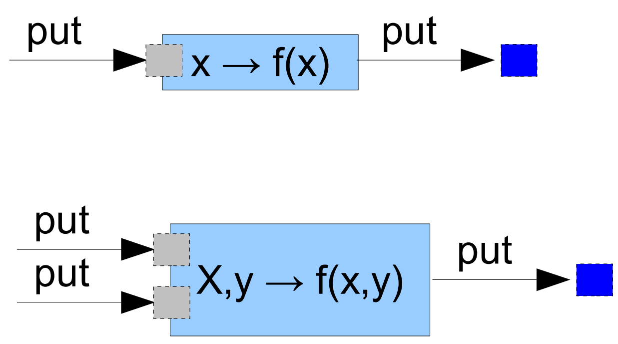
Chaining of transformers is possible, but it's implementation is not obvious. Transformers may not just be concatenated.

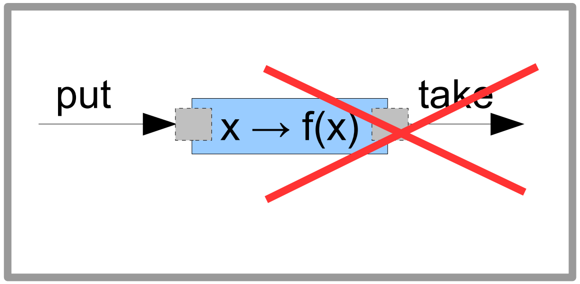


Chaining of transformers suggests a separation of buffering.

Chain-able transformers

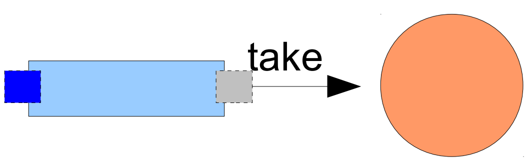
Define transformers without “right side”





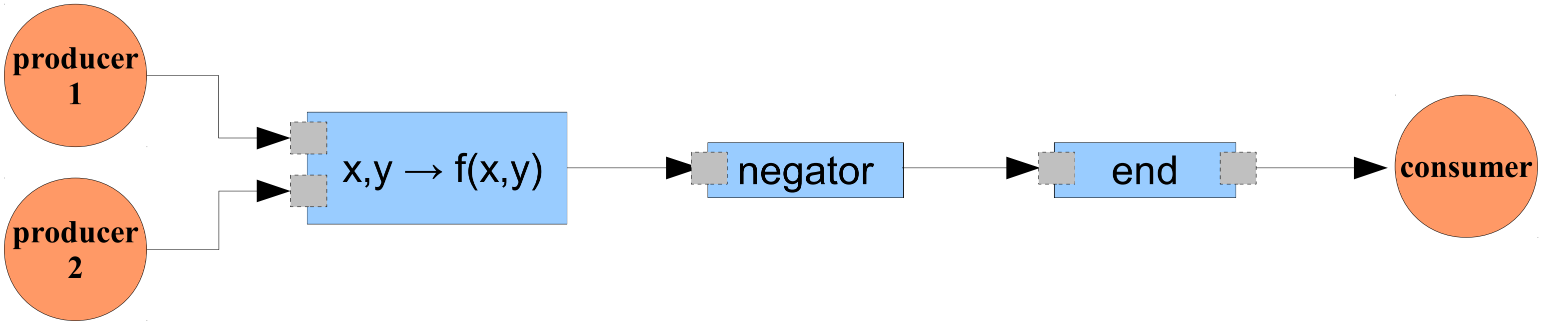
remove “right side” of channels to make them chainable.

Define “transformer ends”



Chain-able transformers : Example

Define transformer chains enduing in transformer ends may be defined



Consumer consumes stream (-1) * $f(x_i,y_i)$

Concurrency: 2 Types of nodes

Combining active and passive nodes

Is it possible ?

- Yes !

Does it make sense ?

- Arguable !
  - Brings increased expressiveness to the user
  - Introduces additional complexity
  - Is an issue of concurrency: should be as transparent as possible

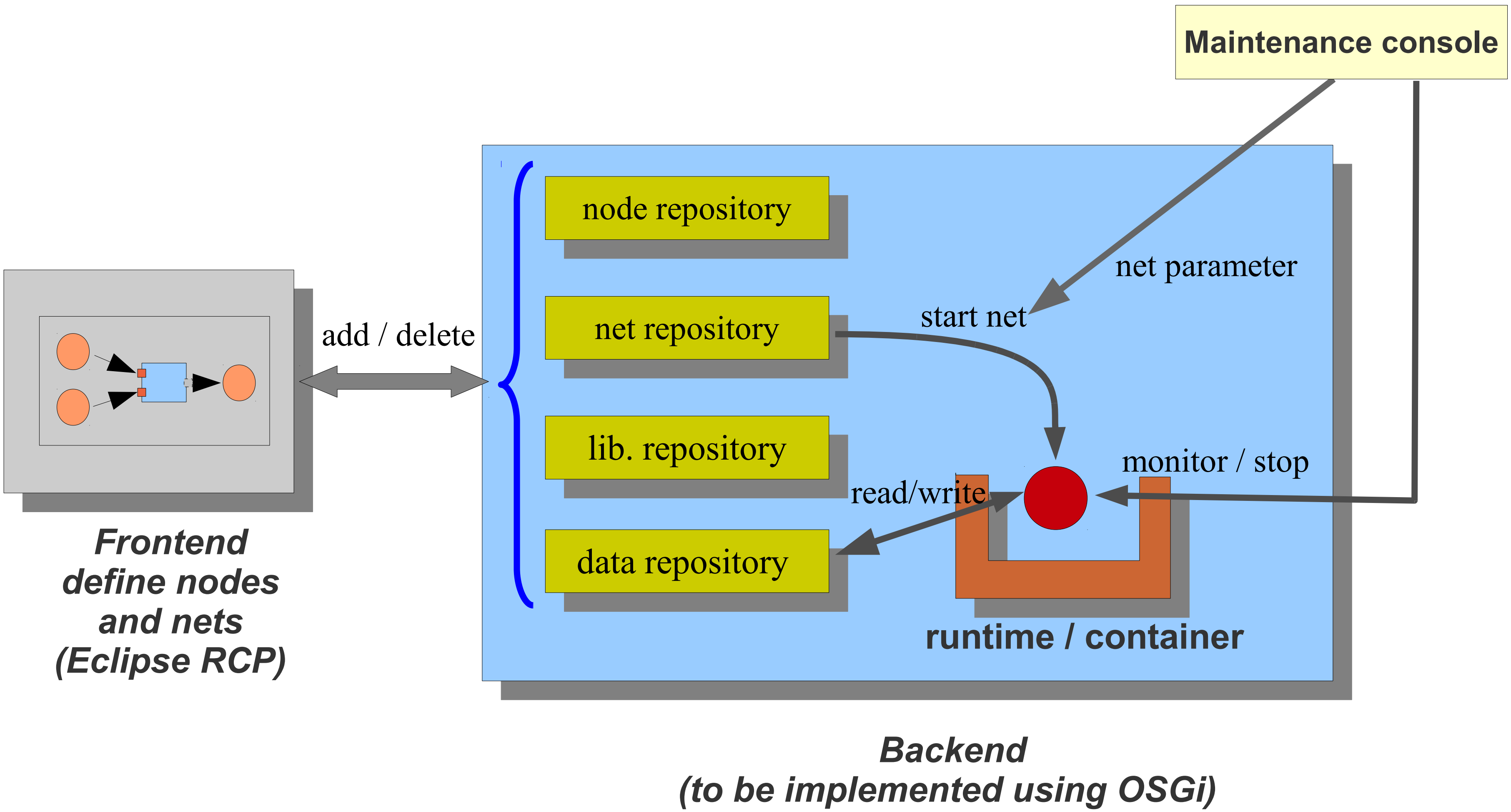
Actual point of view

- Reactive nodes should not be chain-able
- Clear structure of a net with:
  - Active Nodes
  - Connected by channels that
    - may synchronize on several inputs
    - may compute functions

The (Data Analysis) System: conceptual view

- The system consists
  - of repositories containing
    - node definitions / types
    - net definitions / types
    - utility libraries
    - data
  - a container (execution platform like a servlet container)
- The system may
  - accept or delete elements in each repository
  - start or stop the execution of nets within the container
  - import or export data sets

Data Analysis System



Open Issues

- More applications
- Type checking of nets
- Mapping of components and nets to an appropriate component technology (OSGi ?!)