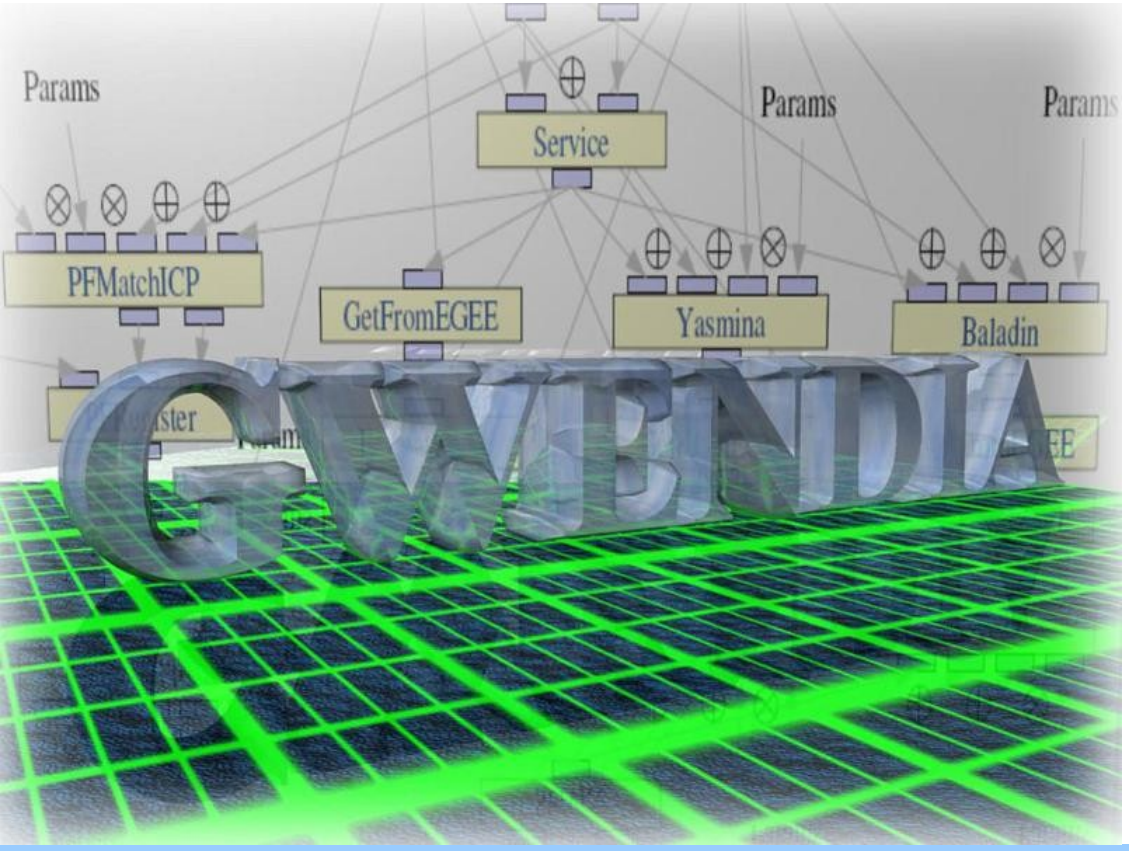
Grid Workflows
Sophia Antipolis, March 23rd 2010



University of Winsconsin

University of Southern California

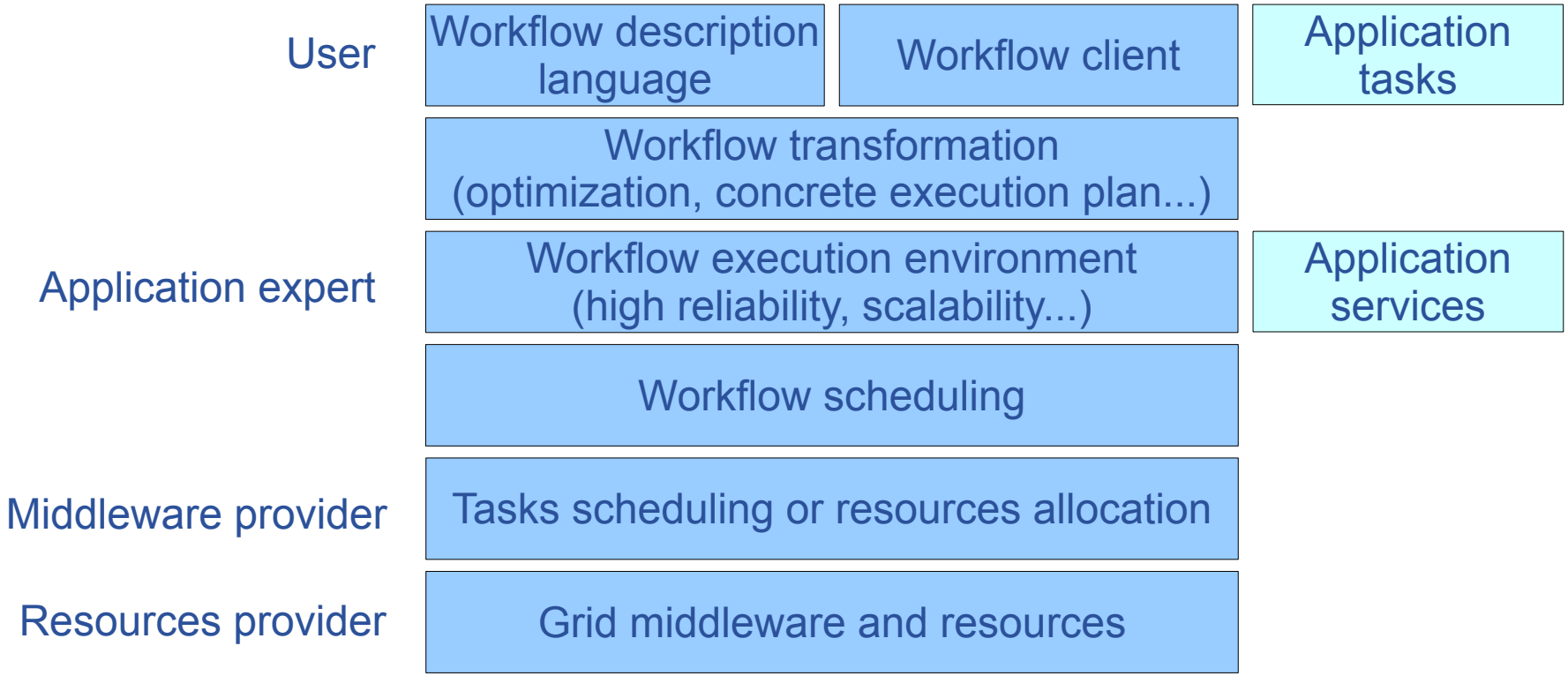
Ecole Normale Supérieure de Lyon

CNRS, I3S, MODALIS

Financé par ANR

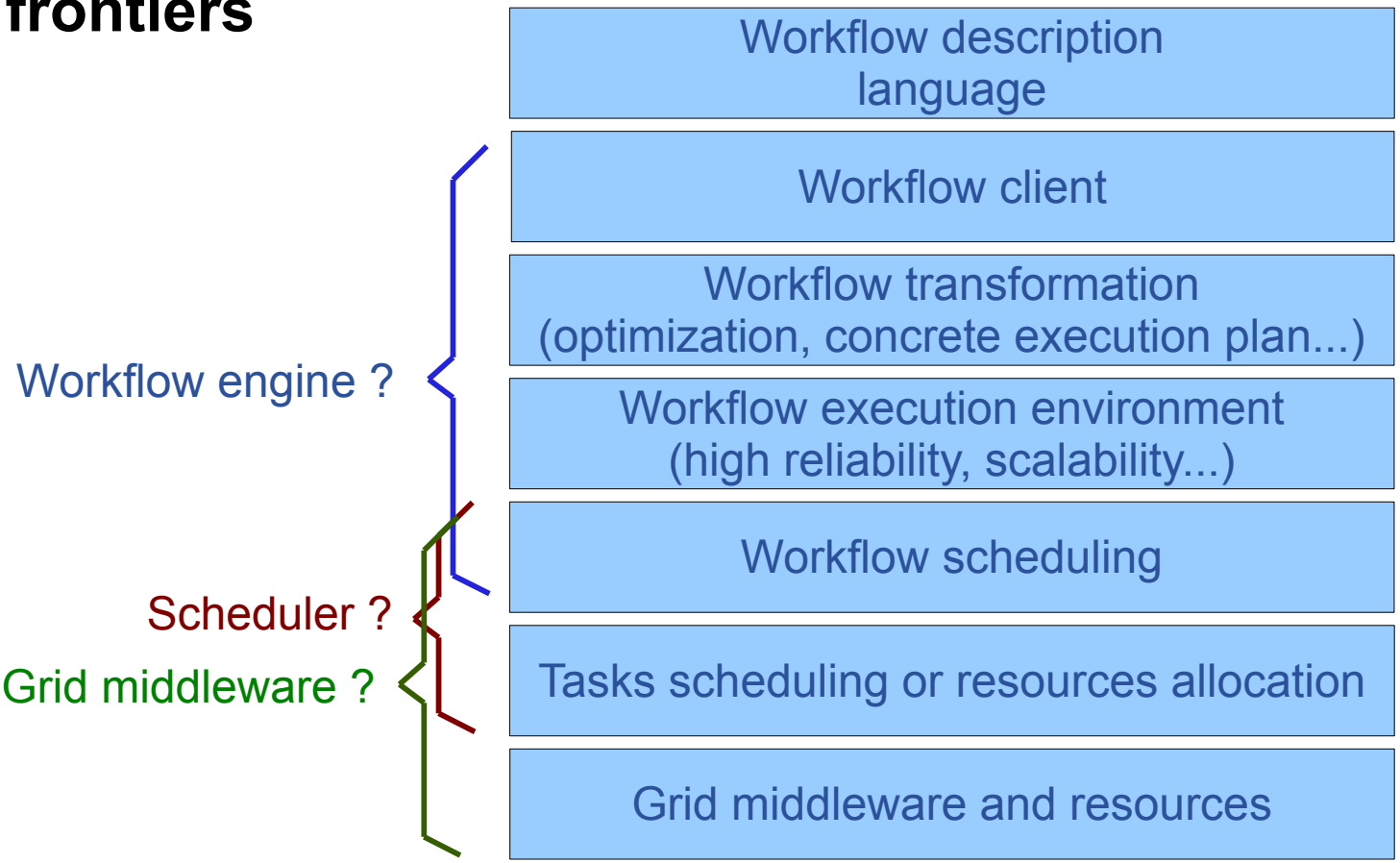
Grid Workflows

- **Grid, HTC, Data Parallelism, Workflows**
  - Grid infrastructure support for HTC
  - Embarassingly parallel (data parallel) applications
  - Coase-grained workflow representation
- **Grid workflow enactment stack**



Grid workflow enactment stack

- **Blurred frontiers**



- **Tightly coupled stages**

Today's agenda

Workflow description language	Workflow client
Workflow transformation (optimization, concrete execution plan...)	
Workflow execution environment (high reliability, scalability...)	
Workflow scheduling	
Tasks scheduling or resources allocation	
Grid middleware and resources	

Johan: *GWENDIA, A data-driven workflow language for grids based on array programming principles*

Ewa: *Pegasus*

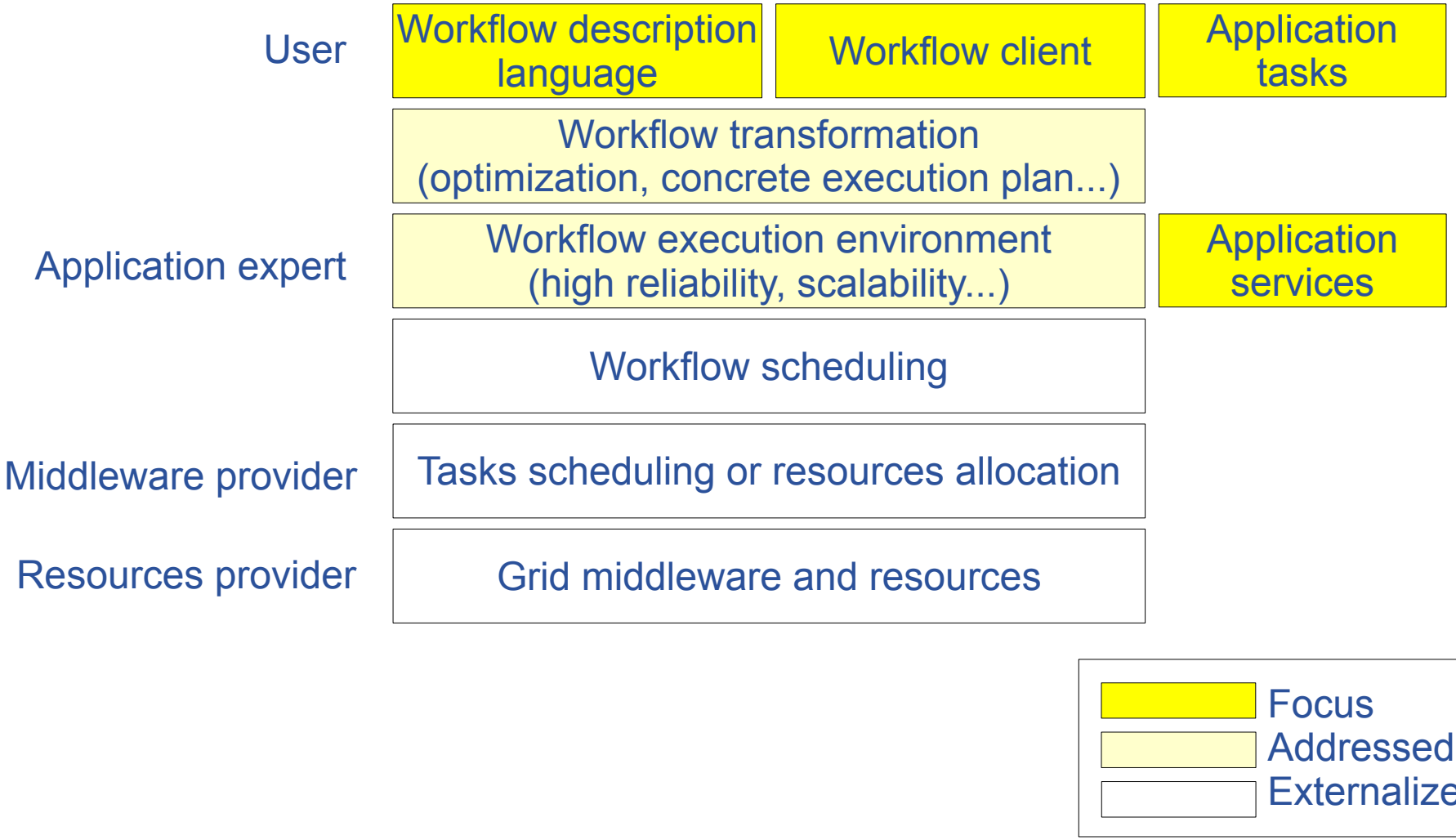
Eddy: *DIET MA-DAG*

Tristan: *A review of operational nightmares and related dreams of grid workflows*

Frédéric x2: *An Overview of Parallel Task Graph and Multi-Workflow Scheduling Algorithms*

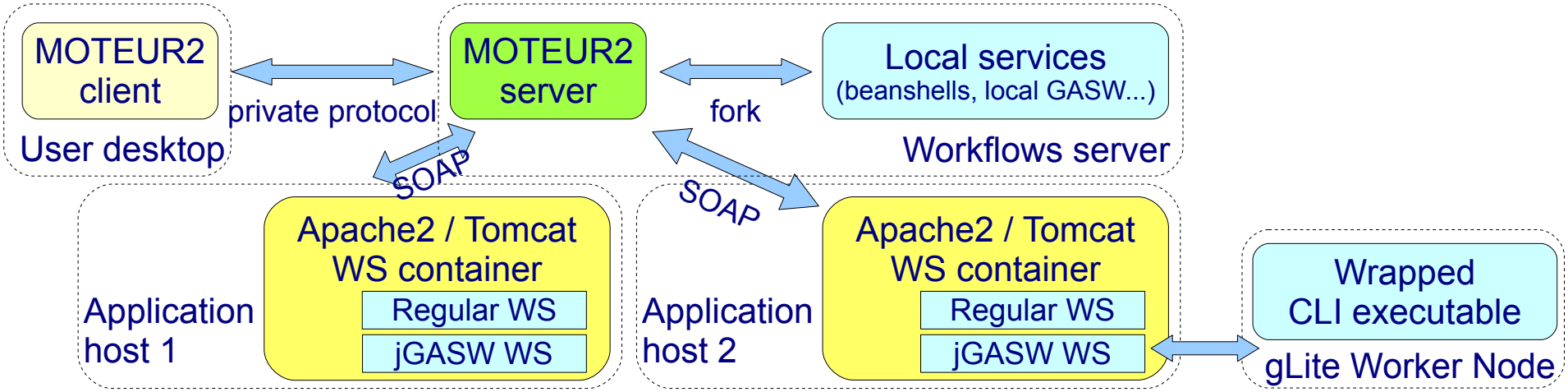
Miron: *Can DAGs do it all - capturing data dependencies with DAGs*

- **Focus on application description**



MOTEUR deployment

- **Typical deployment**

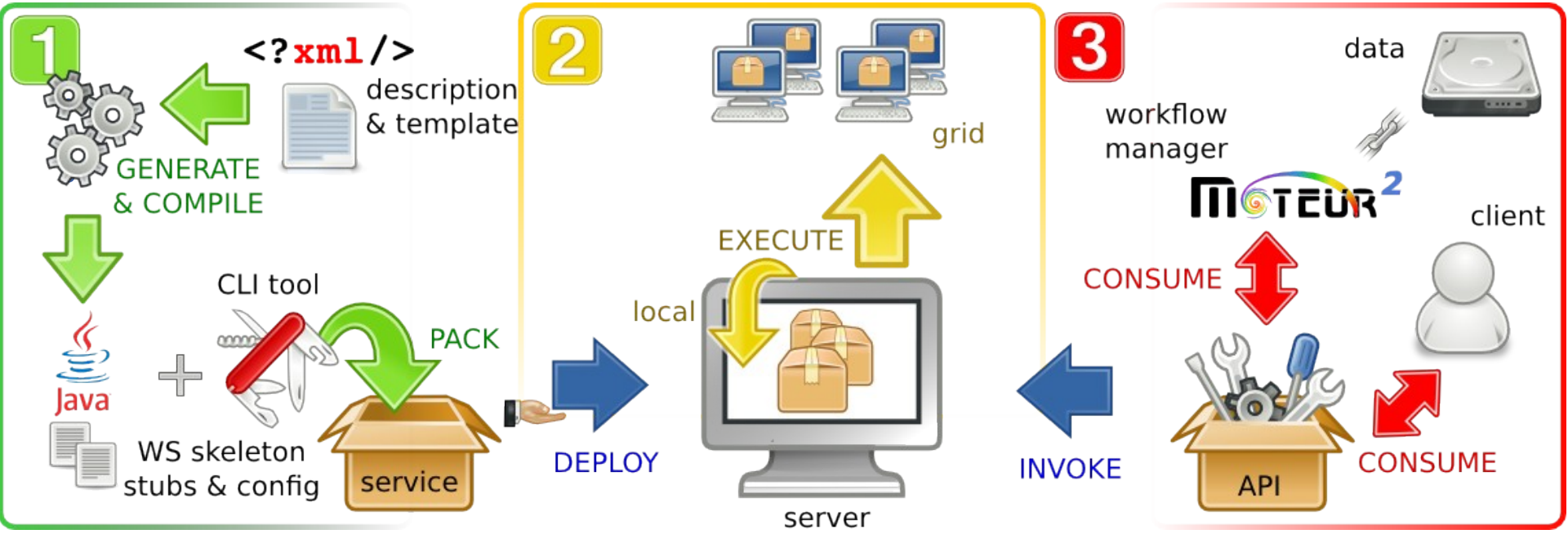


- **gLite middleware interface**
  - CLI wrapped as Web Services
  - WS invocation cause task to be relocated on a grid worker node

Service-Oriented / Tasks

- **jGASW**
  - Code packaging + invocation interface
  - Based on a CLI interface descriptor

Javier Rojas Balderrama

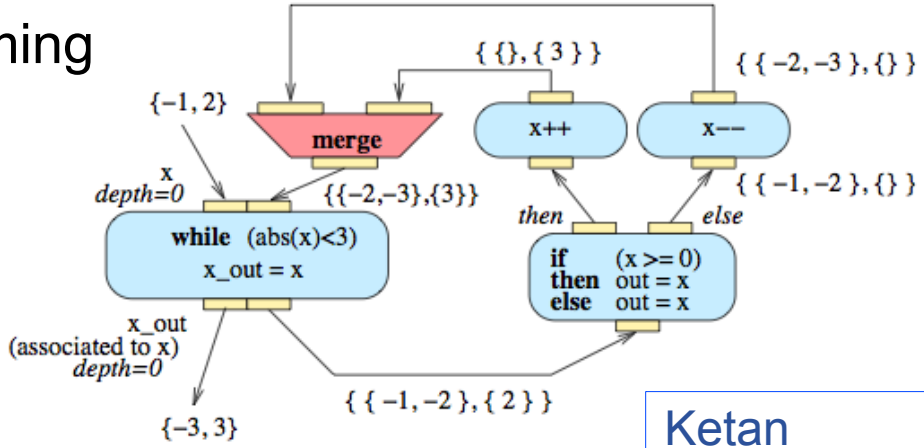


- **Tasks in a Service-Oriented Architecture**
  - Service-Oriented framework; standard Web Service interface
  - Execution unit relocation; embedded legacy CLIs

Workflow language(s)

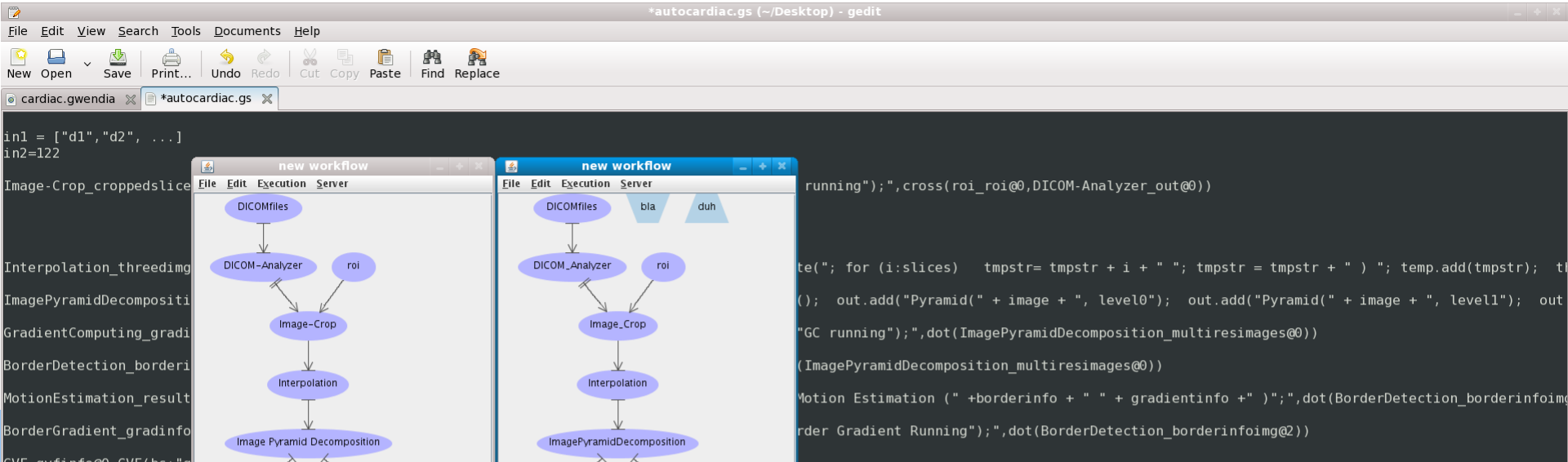
Grid Workflow Efficient Enactment for Data Intensive Applications

- **GWENDIA language**
  - Use array-programming concepts to describe complex data flows
  - Graph-based visual programming



Ketan Maheshwari

- **Dual visual-programming / scripting approach**



GWENDIA language goals

- **Objectives: highly-expressive, user-friendly workflow language for grids**
- **Motivations**
  - Little emphasis on expressiveness of workflow languages nor equivalence between different languages
  - Study how “compatible” different programming models are
- **We strongly believe there is no ideal language anyway**
  - Many different users, use cases and levels of expertise
  - User-friendliness is not an optimizable criterion
- **In practice, many other consideration will prevail**
  - Interface to grid infrastructure
  - Reliability
  - Performance
  - GUI...

These are environment and engines-related considerations. Different implementations could exist.

Context

- **Any programming language could be adopted**
  - From C to Makefiles
- **But all do not fit well a distributed and data-intensive environment**
  - Grid workflows target human-scale activities
  - Prototyping is the rule
  - Scientific applications with heavy user communities and large data sets require high performance and/or high throughput
- **Workflow language for grids**
  - Coarse grain
  - Data intensive
  - Heavy legacy code
  - Interfaced to external Job / Workfload manager(s)

Language

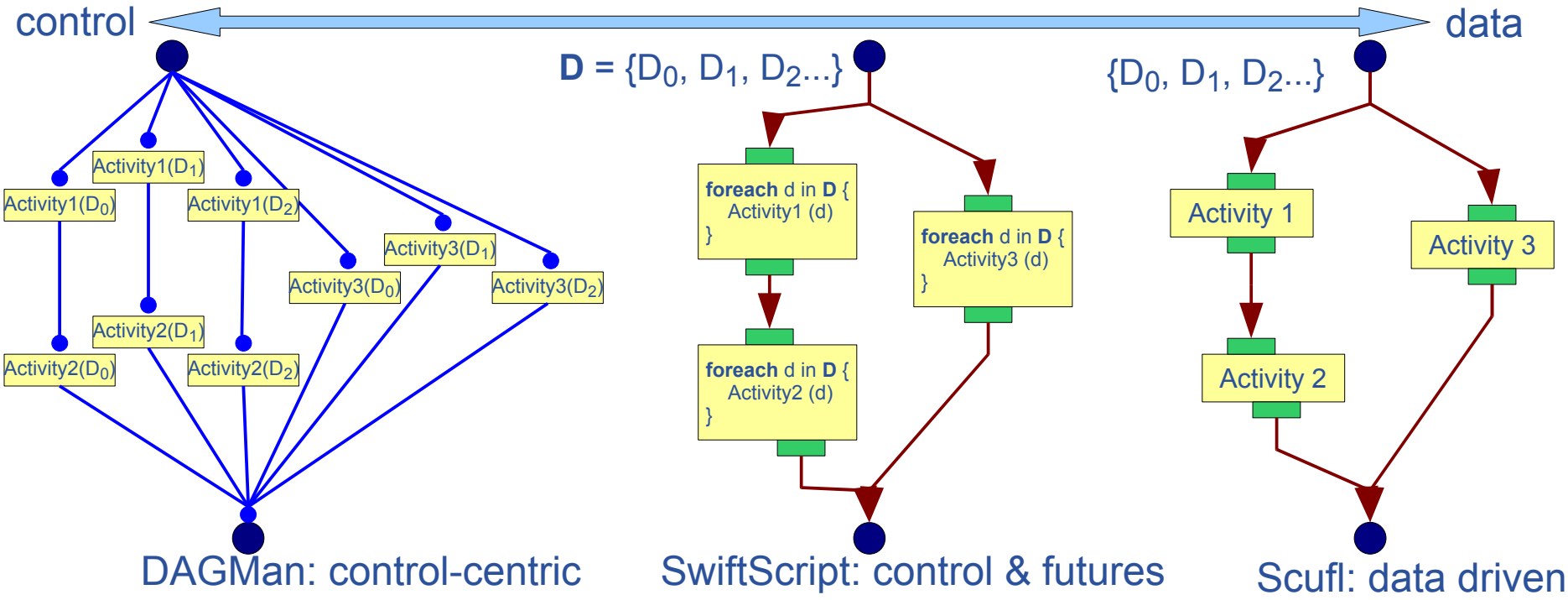
- **Language ↔ Expressiveness**
  - Separate design and enactment phase
  - High expressiveness is not enough: easy access to high expressiveness is a real challenge
- **Not so much emphasis on languages**
  - Workflow environments often use visual programming
  - Internal representations not always clearly specified and documented
  - This impacts the semantics of applications: semantic might change over time or implemented using undocumented, specific assumptions that make understanding of programs difficult in non-trivial cases.

Enactors in GWENDIA

- **Common language, different enactors**
  - MOTEUR on-the-fly evaluator
  - DIET sub-DAGs generator
- **Different enactment strategies and schedulers**
  - MOTEUR send tasks on-the-fly and relies on underlying middleware to schedule tasks
  - DIET MA-DAG generates DAGs and implement various scheduling strategies
- **Common application descriptor, different service interfaces**
  - MOTEUR uses Web Services
  - DIET uses GridRPC API

Data-driven language

- **Many dependencies are data dependencies**



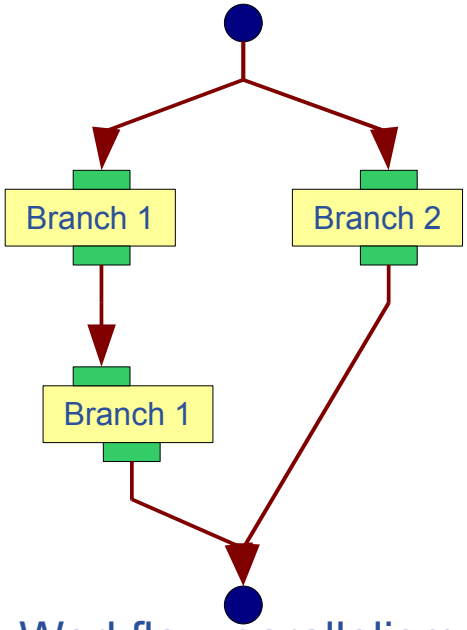
- **Most languages are data-driven in the end**
  - Scufl: by construction
  - MA DAG: by generation
  - SwiftScript: using futures
  - Askalon: mixed data/control flows with coherent, fixed patterns

Data-driven fits data-intensive well

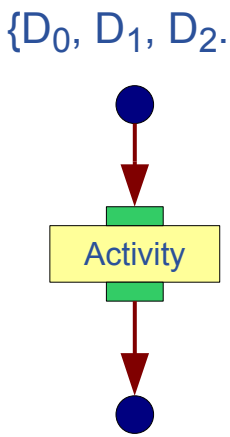
- **Main source of parallelism comes from data sets**
  - Embarrassingly parallel
  - Parameter sweep
  - SPMD
- **Clear separation of application logic and experiment**
  - Experiment = raw data and parameter data
  - This is a clear difference with BPEL or other imperative programming inspired approaches
- **Implicit description of parallelism**

Expression of parallelism

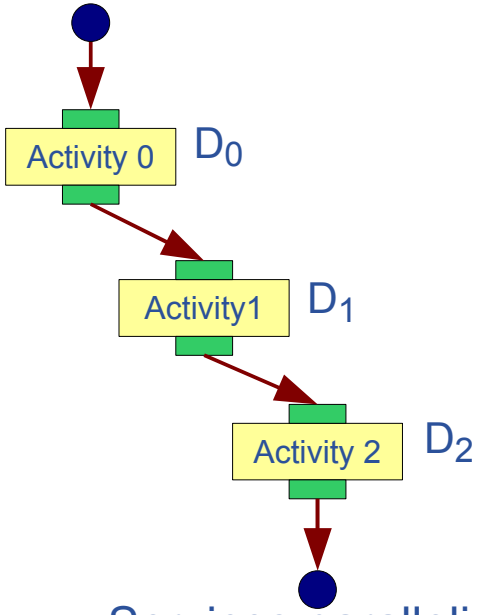
- **Three levels of parallelism**



Workflow parallelism
(implicit in workflow graph)



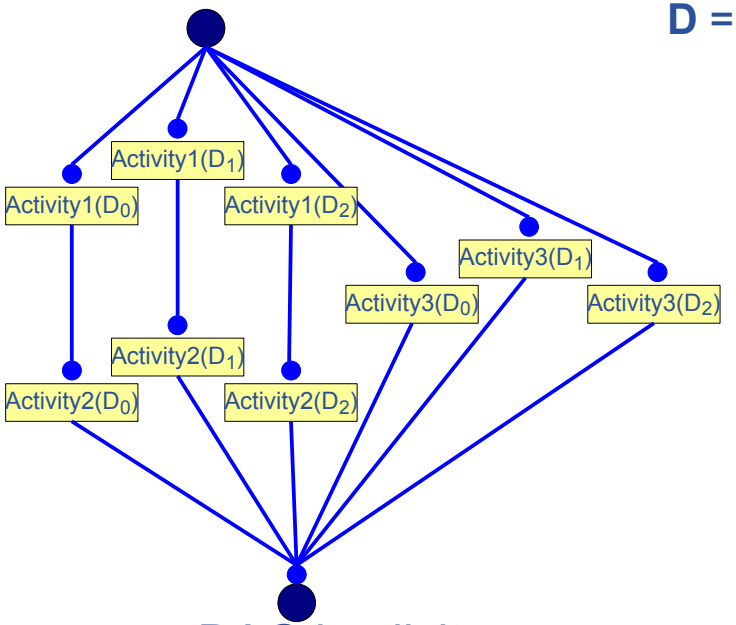
Data parallelism



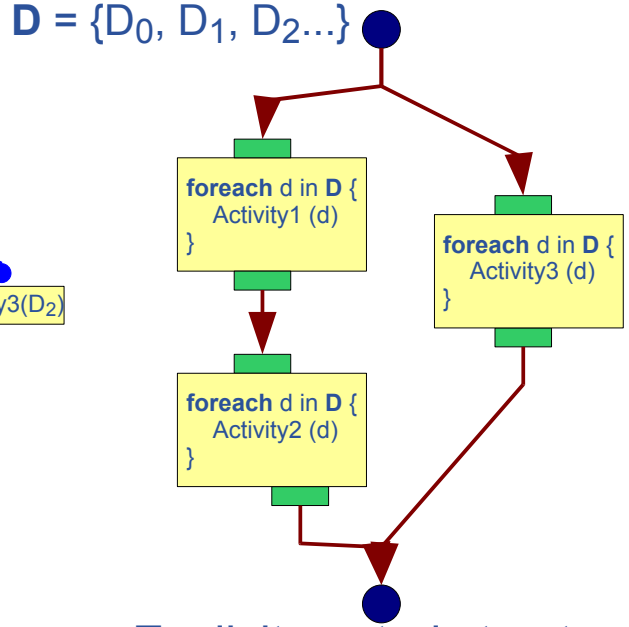
Services parallelism
(pipelining)

- **The expression of data and service parallelisms depend on the workflow language considered**
- **Data parallelism is implicit in data-driven approaches**

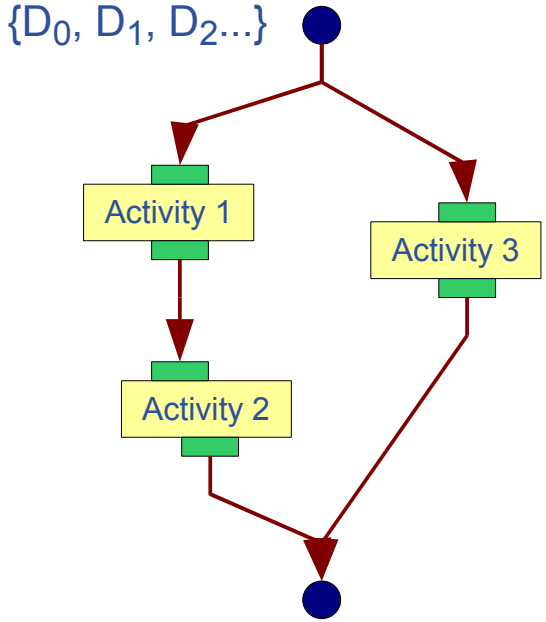
Representations of data parallelism



DAG implicit
(e.g. DAGMan)



Explicit control structure
(e.g. SwiftScript)



Data-driven
(e.g. Scufl)

- **DAGs: implicit in the (large scale) workflow graph**
- **Parallel languages: explicit control structures**
- **Data-driven: transparent**

Parallel languages vs data-driven flows

- **Parallel languages**
  - Bounded: `foreach d in {D0, D1, D2}...`
  - Unbounded: `foreach d in D...`
  - Involves data synchronization at each data parallel service (no pipelining)
- **Data-driven flows (e.g. Scufl)**
  - Independent definition of data flows and data sets
  - Enable completely asynchronous enactment (data parallelism + pipelining): no data synchronization
  - May require explicit data synchronization barriers when this is needed
- **Futures (e.g. SwiftScript)**
  - Non-blocking assignment operations, blocking read
  - Data-centric approach, asynchronous execution

Control structures or not?

- **Many data-driven / visual programming approaches do not define control structures**
  - Hardly any, yet some (fail-if-false, fail-if-true)
- **DAGs do not include any loop**
  - Because complex application logic is described inside the workflow activities
- **Yet control on data flows is sometimes needed**
  - Different execution conditions
  - Exceptions / Retry on errors at the application level
  - ...

Gwendia data-driven workflow languages

- **Search for a language integrating**
  - Data-driven approach (transparent parallelism)
  - Data arrays manipulation
  - Control structures
  - Maximum asynchronous execution
  - Compact representation
- **Applying array programming principles**
  - Arrays are first class entities
  - Operators apply both to scalar and arrays
- **Indexing scheme to preserve computation coherence in an asynchronous environment**
- **Arrays nesting to represent data-sets decomposition / recomposition**

Arrays, activities and ports

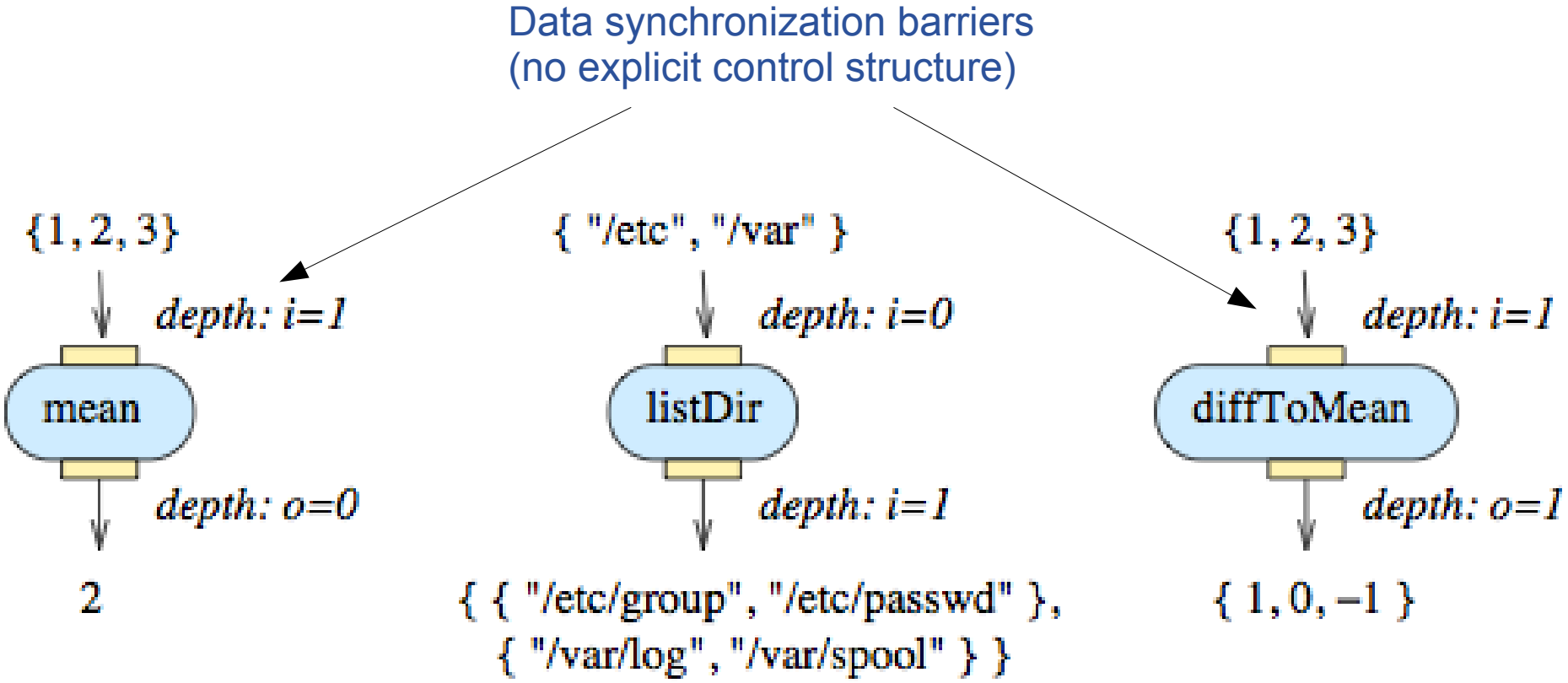
- **Array programming principles**
  - Lightweight syntax for handling arrays:

```
X+Y ≡ foreach i in indices(X) do
          Xi + Yi
      done
```

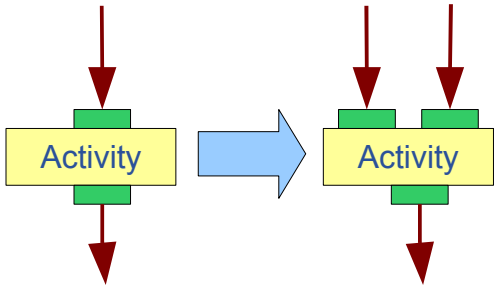
  - Convenient representation for vectorial processors
  - Extendable to any operation on arrays of values
  - Use nested arrays `X={{1, 2},{-1,-2}}` is a 2-nesting levels array
- **Activities with array parameters**
  - `A(X) ≡ { A(X0), ..., A(Xn) }`
  - `{{1, 2},{-1,-2}} ≡ { {A(1), A(2)}, {A(-1), A(-2)} }`
- **Ports depth**
  - Activity ports have a defined depth that corresponds to the number of array nesting levels being synchronized:
  - `Mean({1, 2, 3}) = 2` if `Mean` has input port depth 1

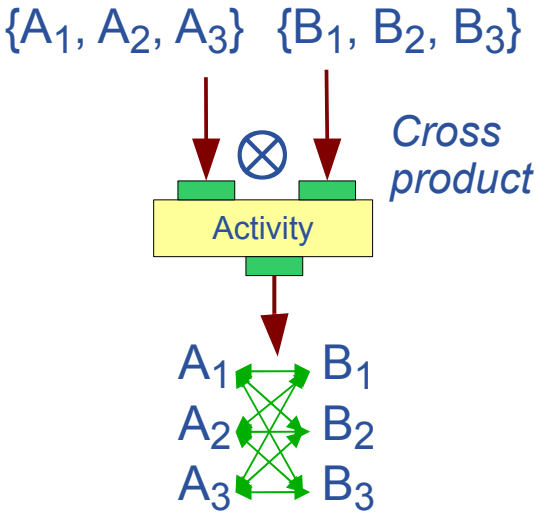
Nested arrays and ports depth

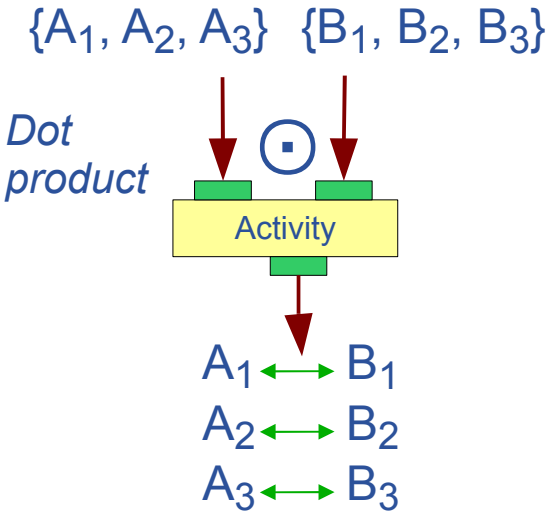
- **Input arrays nesting levels are combined with activities ports depth**



Iteration strategies







- **In Scufl**

A⊗B

A⊙B

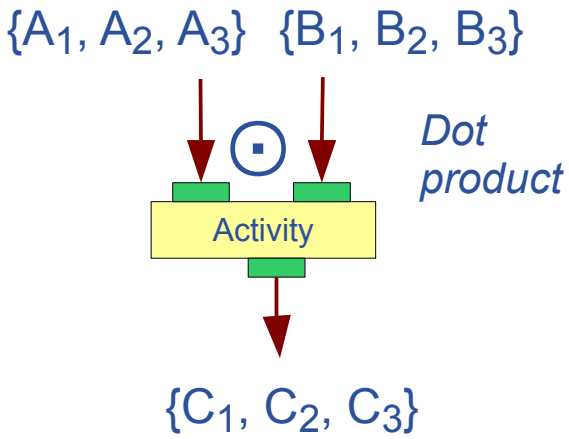
- **Parallel language**

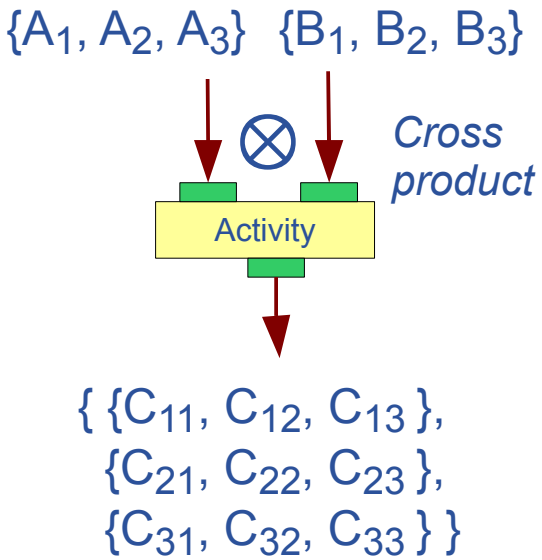
```
foreach a in A
  foreach b in B
    fire Activity(a,b)
```

```
foreach i in indices(A)
  fire Activity(Ai,Bi)
```

Iteration strategies and depths

- **Iterations strategies define output indexing strategies**
- **They may change data nesting levels**



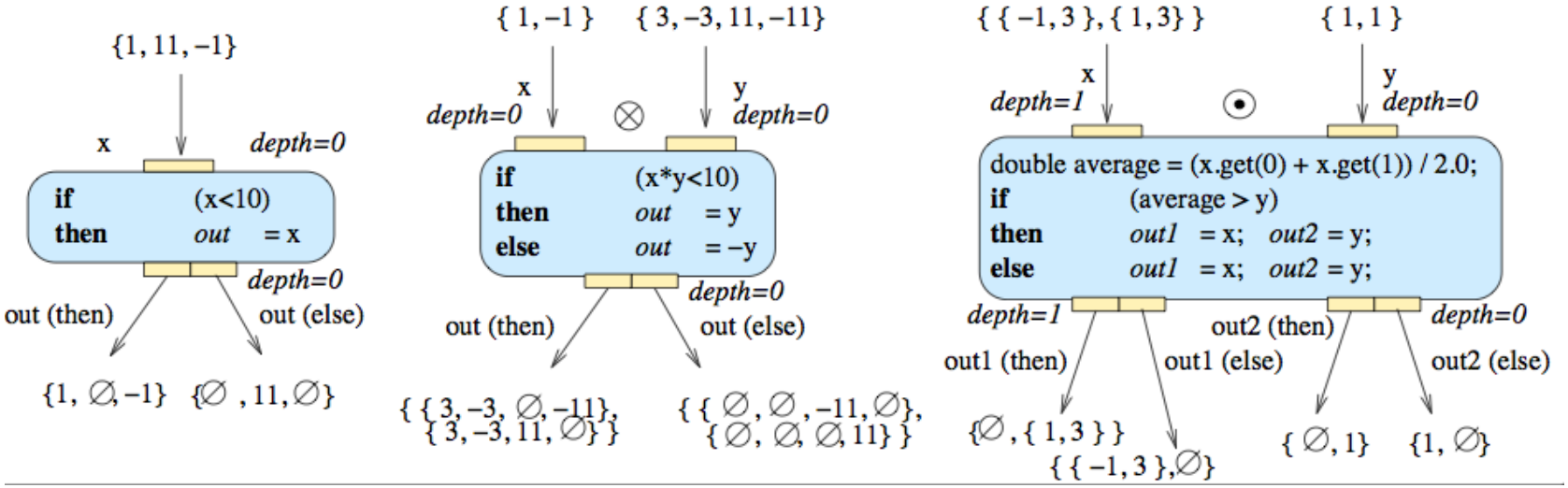


Empty data element

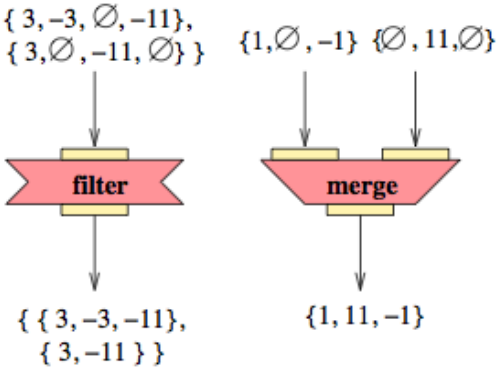
- **Special *void* value ∅**
- **A(∅) ≡ ∅**
  - No evaluation of the activity (∅ is only known from the workflow engine)
- **Iteration strategies semantics**
  - **x ⊙ ∅ ≡ ∅ ⊙ x ≡ ∅**
  - **x ⊗ ∅ ≡ ∅ ⊗ x ≡ ∅**
- **∅ has an index in the array it belongs to**

Conditionals

- Test expression variables mapped to input ports
- Dual “then / else” output ports
- Use of the special “void” data element to preserve indexing scheme coherence

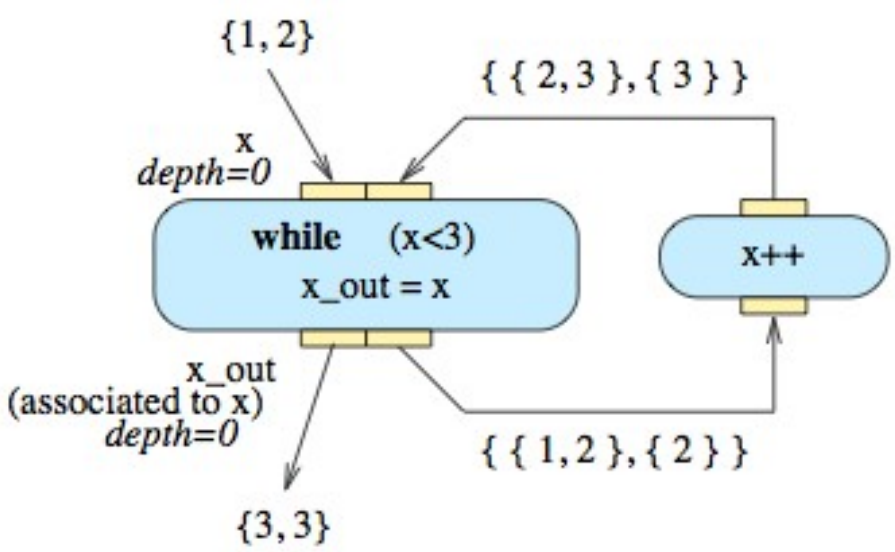


- Special filter / merge operations

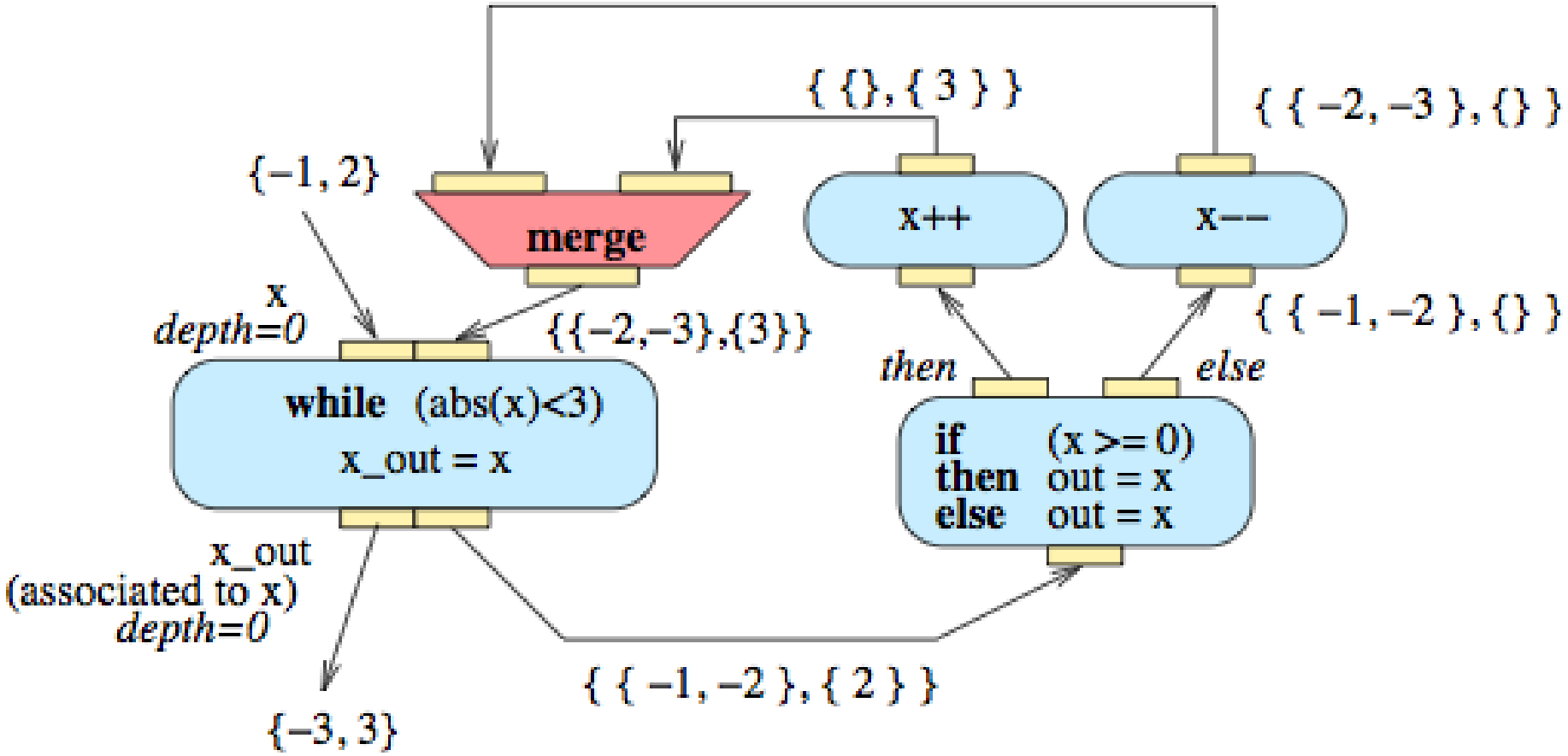


Loops

- **Test expression variables mapped to input ports**
- **Dual "inner / outer" ports**
  - Input ports: initialization / iterated values
  - Output ports: iterated values / end of loop



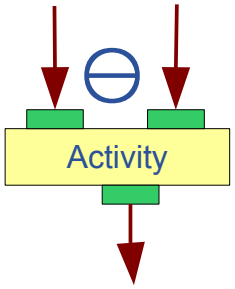
Example


{ {}, { 3 } }
{ { −2, −3 }, {} }
{−1, 2}
merge
x++
x−−
x
depth=0
{{−2,−3},{3}}
{ { −1, −2 }, {} }
then
else
while (abs(x)<3)
x_out = x
if (x >= 0)
then out = x
else out = x
x_out
(associated to x)
depth=0
{ { −1, −2 }, { 2 } }
{−3, 3}

Additional iteration strategies

- **Flat-cross product**
- **Match product**

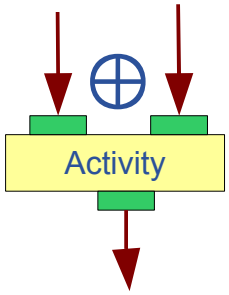
$\{A_1, A_2, A_3\}$ $\{B_1, B_2, B_3\}$



Flat cross product

$\{ C_{11}, C_{12}, C_{13}, C_{21}, C_{22}, C_{23}, C_{31}, C_{32}, C_{33} \}$

$\{A^a_1, A^b_2, A_3\}$ $\{B_1, B^{ab}_2, B^b_3\}$



Match product

$\{ \{\varnothing, C_{12}, \varnothing\},$
$\{\varnothing, C_{22}, C_{23}\},$
$\{\varnothing, \varnothing, \varnothing\} \}$

Implementations

- **Control structures**
  - Forseeable: data flow generated statically from inputs (dot, cross, flat-cross and for loops)
  - Unforseeable: dynamic generation (match, conditionals, while loop, output array with unknown size)
- **Complete asynchronous execution**
  - With best effort to deal with unknown indexes (e.g. flat cross)
- **Moteur2 workflow engine**
  http://modalis.polytech.unice.fr/moteur/start
  - On-the-fly interpreter
  - All structures processed dynamically
- **MA-DAG engine http://www.ens-lyon.fr/graal/~diet**
  - Partial DAGs generation, greedy approach
  - Multi-DAGs scheduling heuristics

Conclusions

- **Coherent integration of:**
  - Data-driven approach: Scufl-based
  - Array programming principles: Nested arrays as first-class entities
  - Asynchronous execution: coherence through data items indexing
  - Control structures
- **Transparent parallelism**
  - No *foreach* nor *fork/join* constructs
- **Compact**
  - Only application logic described, no redundancy
  - No *for* kind of loop over data sets
  - Iteration strategies
- **Reusability**
  - Decoupled data and application logic