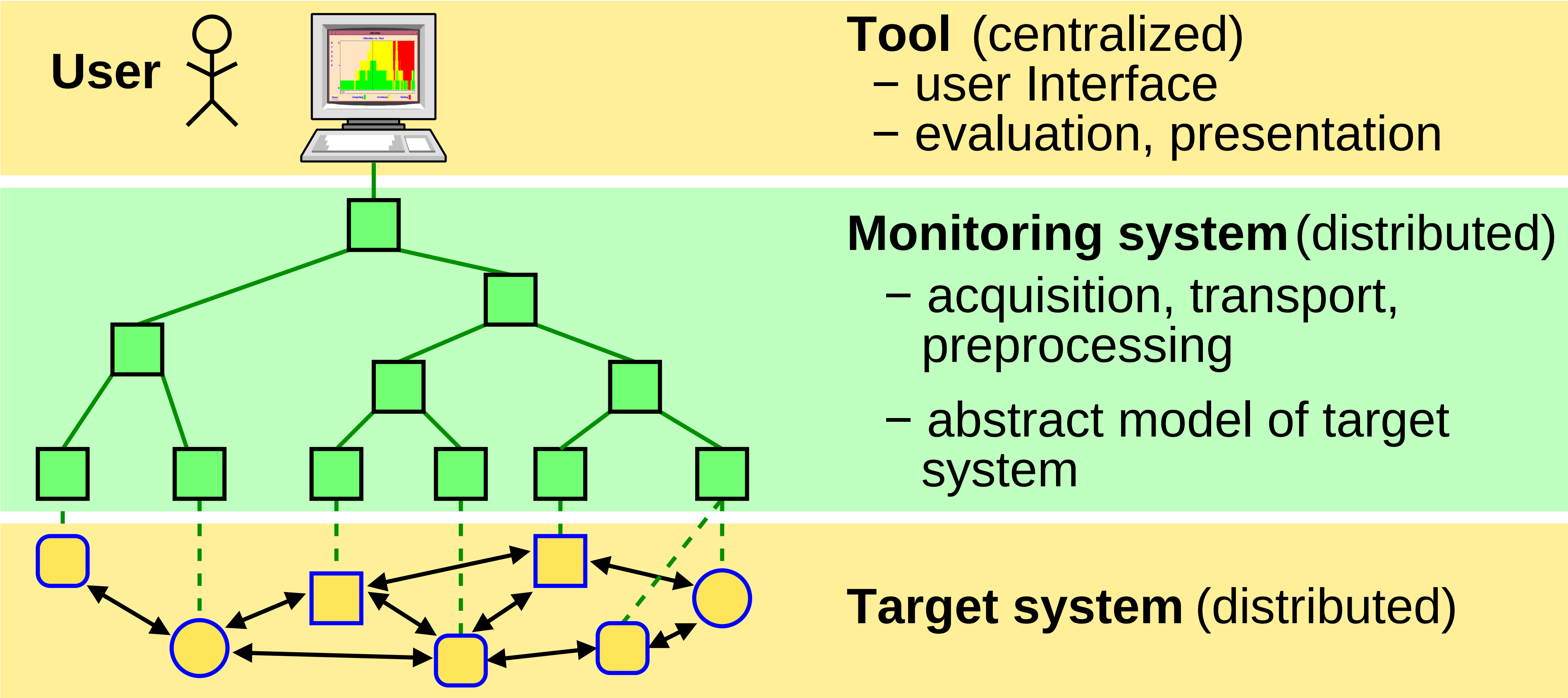
Towards an Automatically Distributed Evaluation of Event Data

Roland Wismüller
University of Siegen
Operating Systems and Distributed Systems
roland.wismueller@uni-siegen.de

Stand: May 5, 2010

Background

- Online monitoring of parallel and distributed software
- Generic (distributed) monitoring system, supporting different tools
  - goals: ease of use, scalability



Lessons Learned

- Provide a well defined interface for the tools
- Provide an object-oriented model of the target system
- Provide for extensibility of the interface
- Use the event / action paradigm
  - i.e. allow the tool to specify arbitrary actions to be executed when an event is detected in the target system
- Support requests on sets of objects

- Make the implementation as asynchronous as possible
- Push execution of actions towards the event sources

First Approach: OMIS / OCM

- Specification and implementation of an online monitoring interface
- Basis: object based model of target and monitoring system
  - system, nodes, processes, threads; counter, timers
- Request language for event / action relations

```
thread_started_lib_call([p_1,p_2], "MPI_Send") :
    pa_counter_increment(pa_c_1, $par8)
```

```
thread_started_lib_call([p_1,p_2], "MPI_Send") :
    thread_stop([a_]) thread_get_backtrace([$thread])
```

- Location transparency: automatic distribution of requests
- Extensibility via plug-in interface for new events, actions and objects

First Approach: OMIS / OCM ...

- Problems:
  - “unlovely” programming in the tools
    - tool is programmed in C++/Java
    - monitoring system is “programmed” in OMIS language
  - OMIS language is not really object-oriented
    - c.f. `thread_started_lib_call([p_1,p_2], ...)`
  - extensions are difficult to program
    - complex interface to OCM core
    - distribution must be handled explicitly

The Tool Developer's Wish

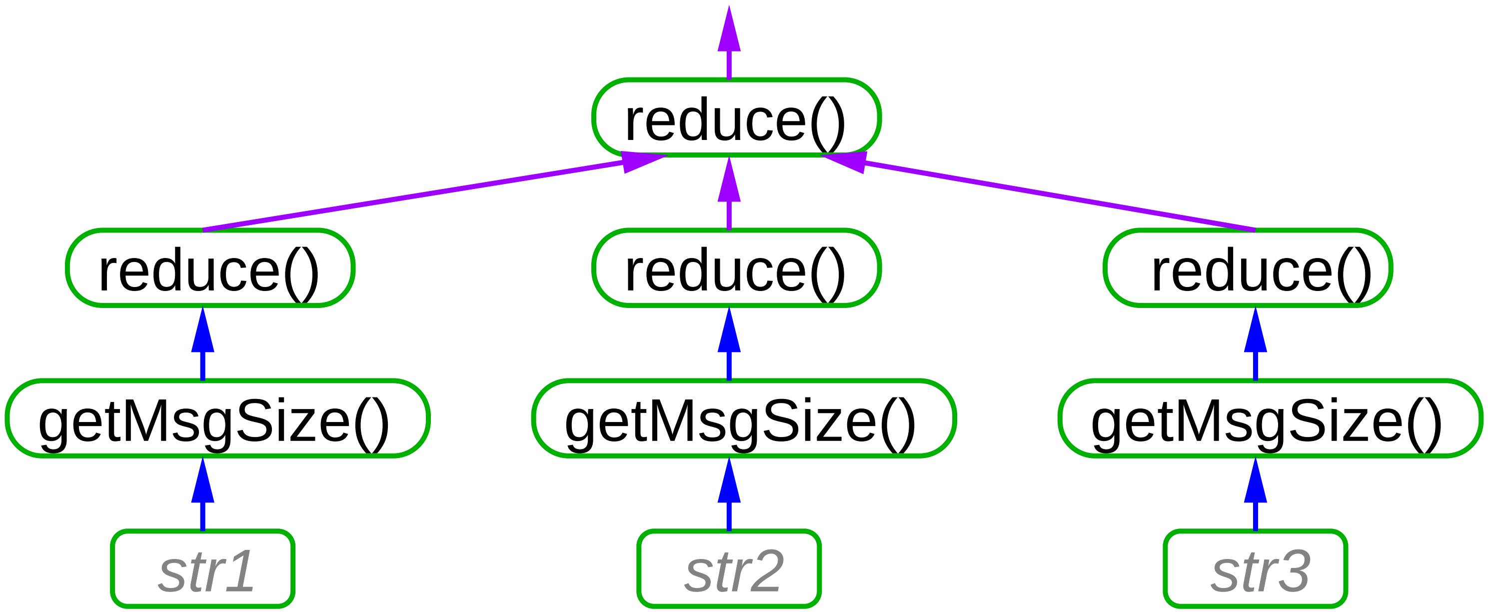
- Object oriented model of target system
  - local (proxy) objects for nodes, processes, ...
- Abstractions for sets and event streams
- Fully integrated into Java / C++:

```
Set<Node> nodes = System.getNodes(...);
Set<Processes> procs = nodes.getProcesses(...);
Set<Stream<SendEvent>> ev = procs.getSendEvents(...);
IntVal tot = Set.reduce(Stream.reduce(ev.getMsgSize(),
                                    SUM),SUM);
...
print(tot.getValue());
```

- Combined with distributed evaluation!

Towards an Implementation (1)

- **Q**: how to map this program to the distributed monitoring system?
- **A**: use data flow graphs as intermediate representation!
  - purely functional model, only explicit (stream) communication
  - easy to (autmatically) distribute them for execution
- Data flow graph for the example (3 processes)



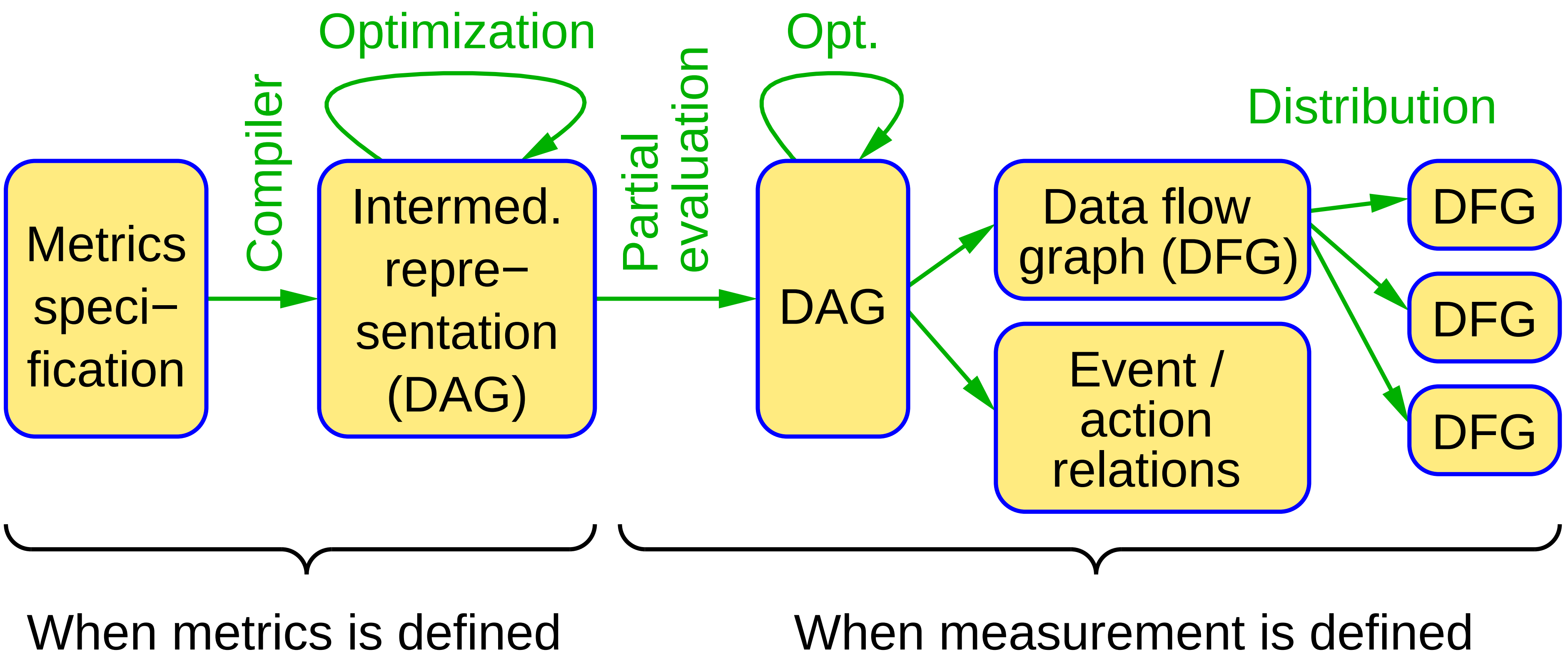
Previous Experience

- EU project CrossGrid: online performance analysis tool G-PM
- Performance metric specification language PMSL
  - allows users to specify new metrics at runtime
  - metrics are evaluated by distributed monitoring system
- Example of PMSL mectrics:

```
Comm_Volume(Process[] procs, TimeInterval ti) {
  PROBE send(Process, VirtualTime, int);
  Value[][] sz; Value[] tmp;
  int size; Process p; VirtualTime vt;
  sz[p][vt] = size AT send(p, vt, size);
  tmp[p] = SUM(sz[p][vt] WHERE sz[p][vt].time IN ti);
  return SUM(tmp[p] WHERE p IN procs);
}
```

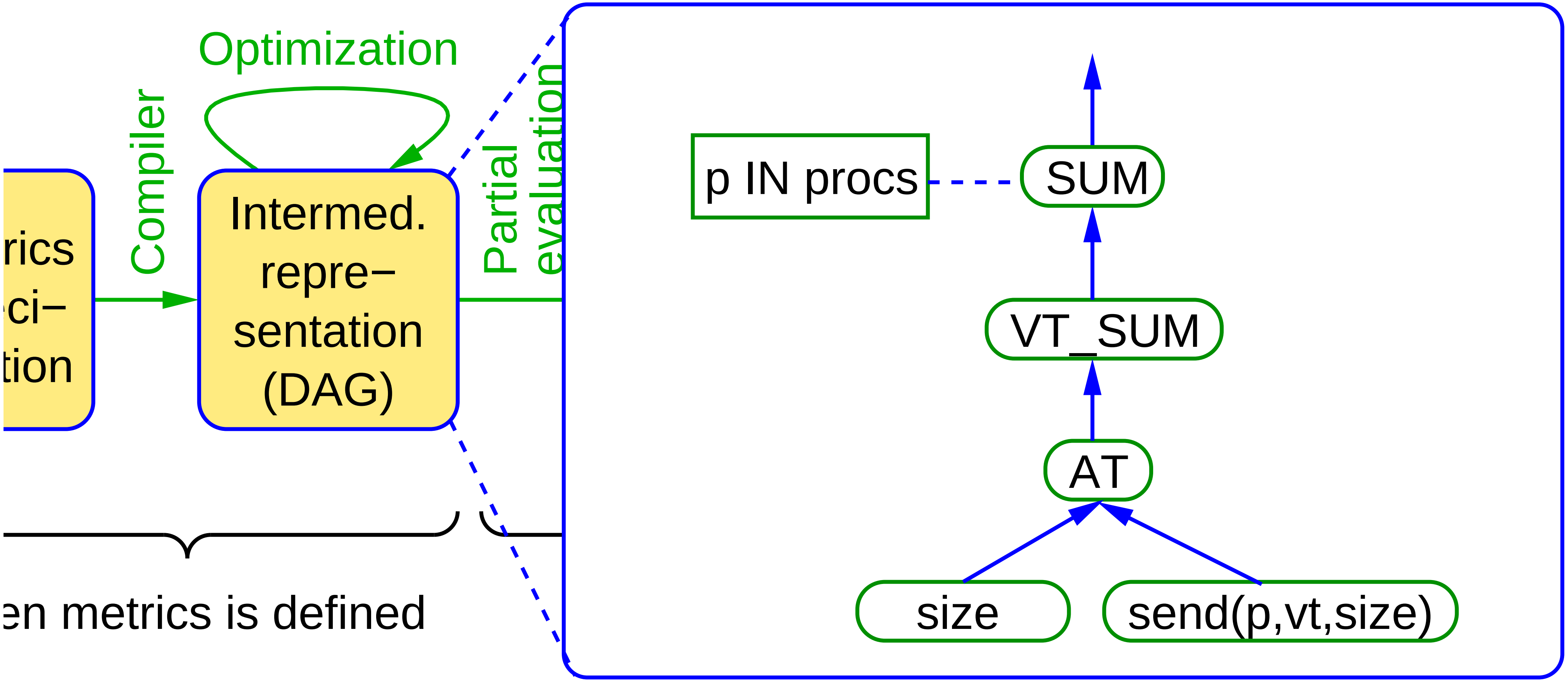
Previous Experience ...

Implementation using distributed evaluation



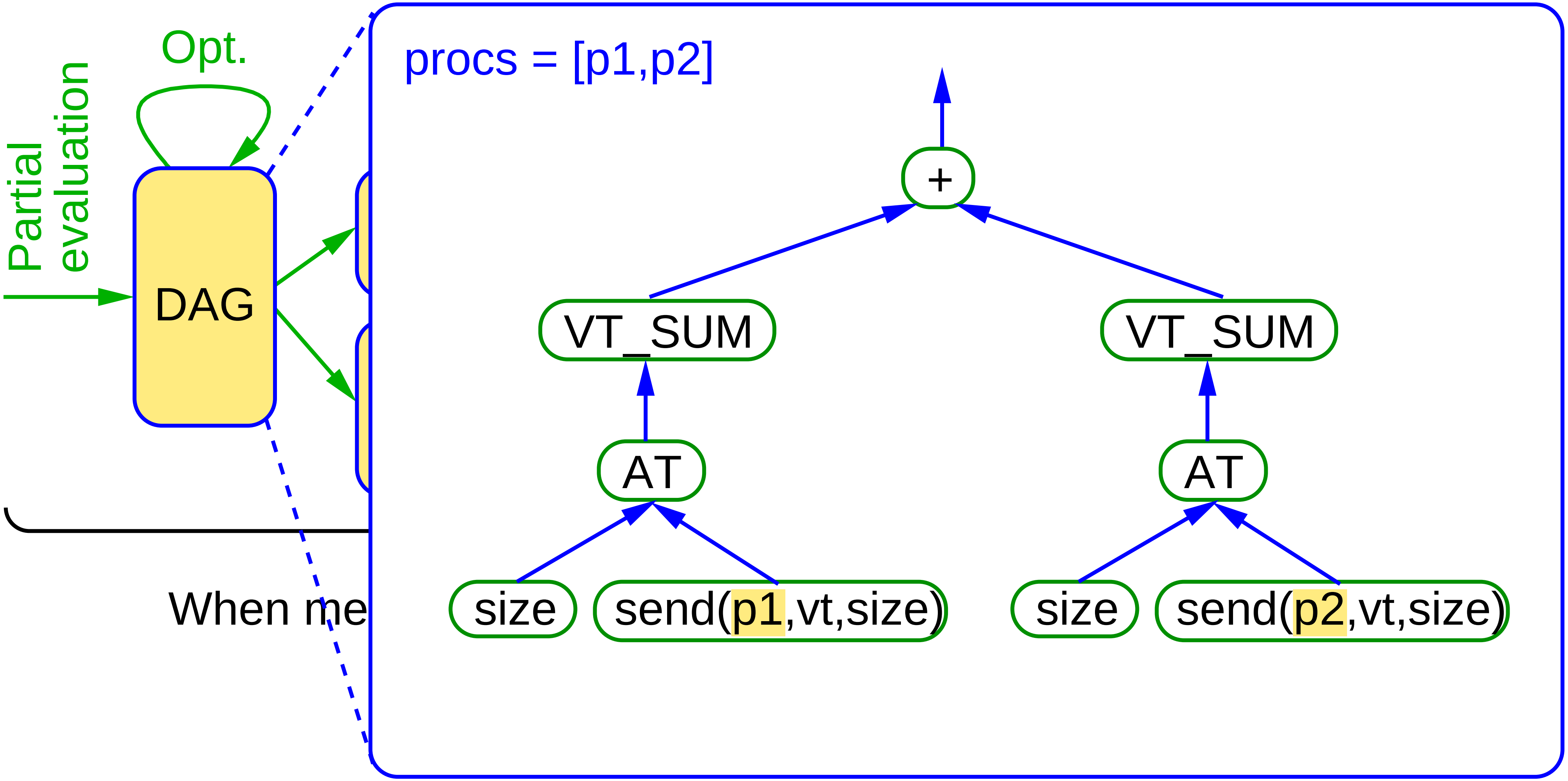
Previous Experience ...

Implementation using distributed evaluation

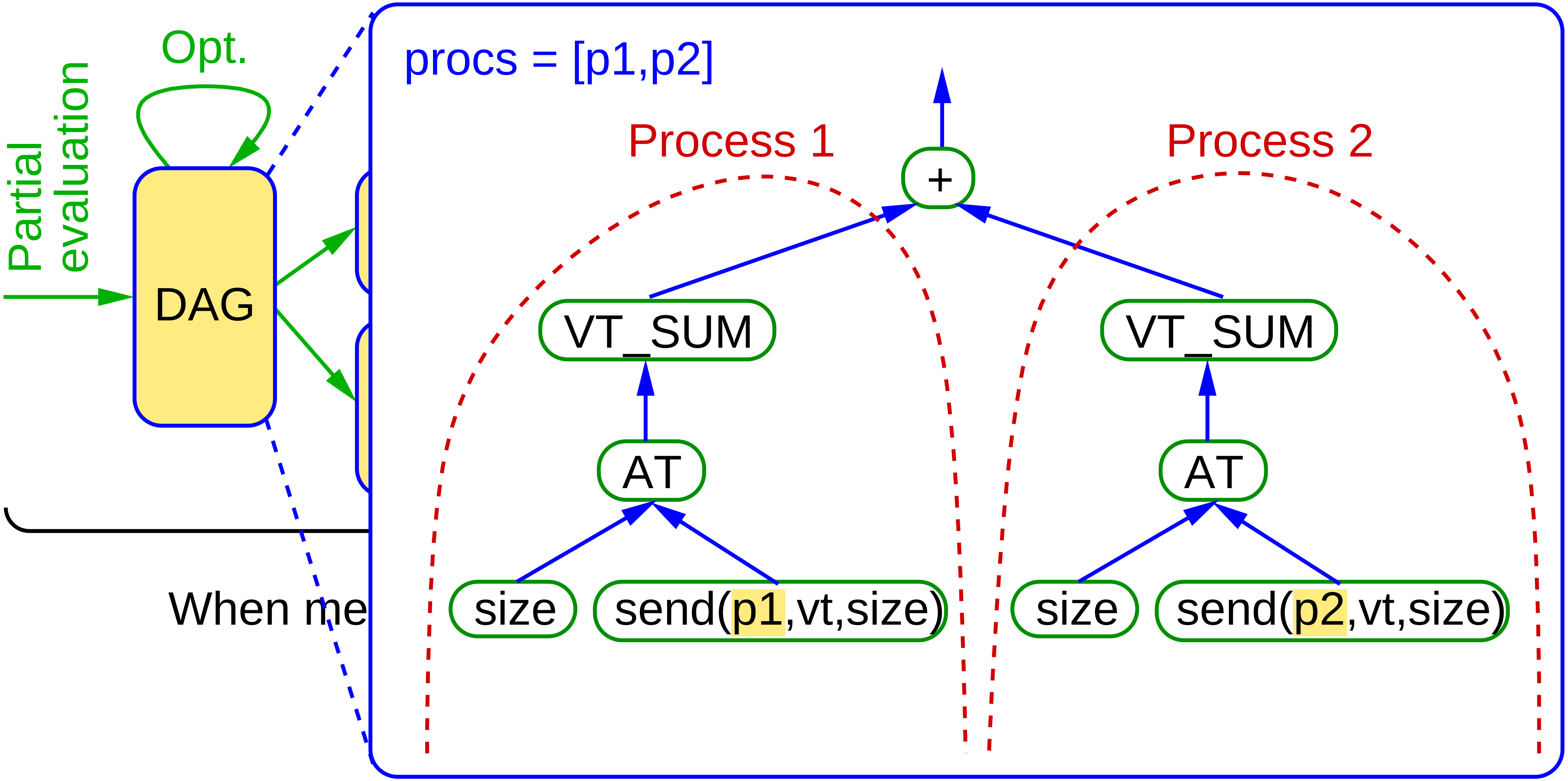


Previous Experience ...

Implementation using distributed evaluation

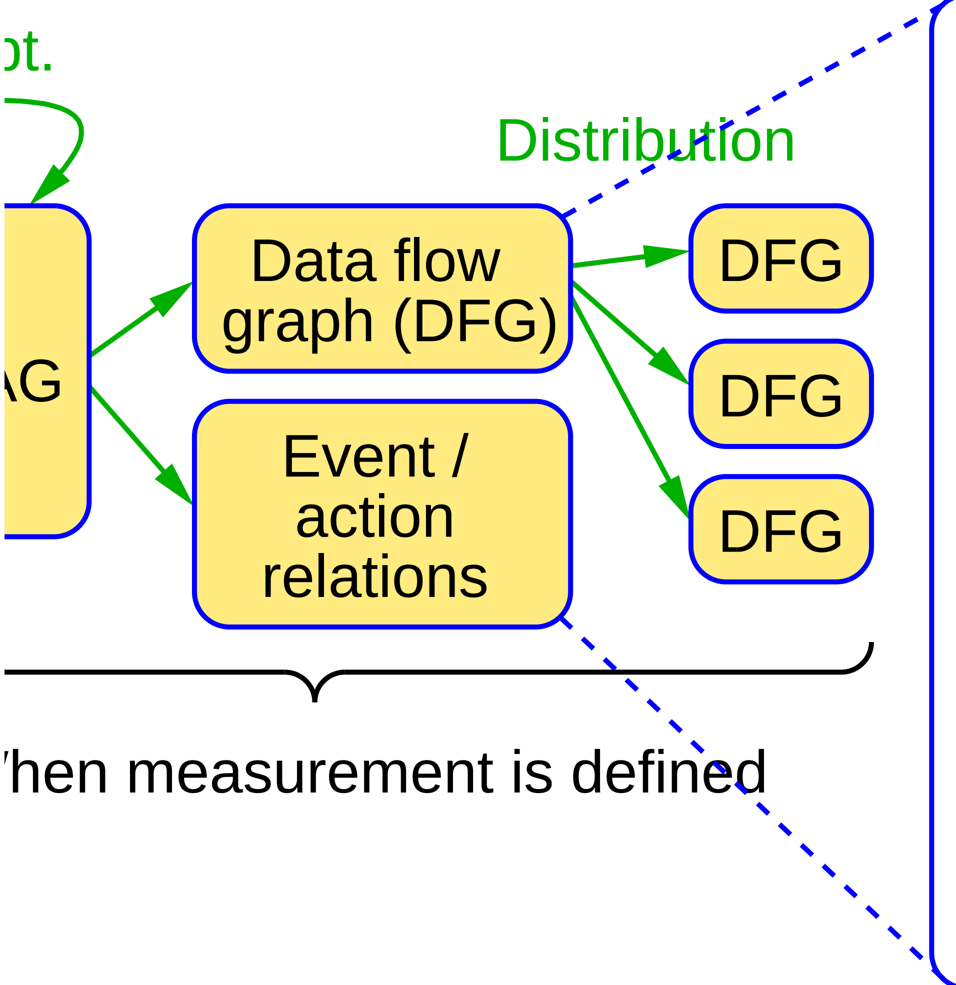


Implementation using distributed evaluation



Previous Experience ...

Implementation using distributed evaluation



Converting the DAG into data structures for execution in OCM

- Data flow graphs
  - distribution to OCM components
  - interpretation by OCM plug−in
- Event/action relations
  - monitoring the events
  - data transport between data flow graphs

Towards an Implementation (2)

- **Q**: how to create the data flow graphs?
- **A**: use transparent proxies!

- Inspiration: ProActive (INRIA)
  - transparent asynchronous RMI (remote method invocation)
    - RMI immediately returns a *future* (proxy object)
    - once the result arrived, method calls are forwarded to it
    - method call blocks, if result is not yet available
  - groups (sets)
    - method called on group is executed for each member
    - method result again is a group
    - also implemented via proxy object
  - proxy classes are generated at run time (using Java reflection)

Towards an Implementation (2) ...

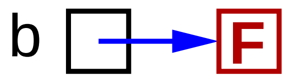
- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    $\Rightarrow$ we end up with a data flow graph

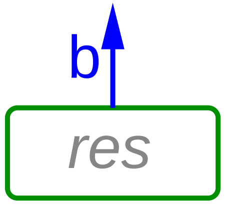
```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    ⇒ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```



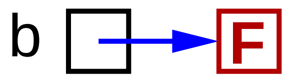


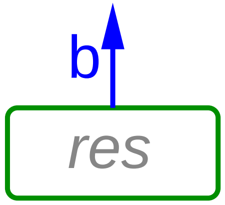
Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed

    ⇒ we end up with a data flow graph

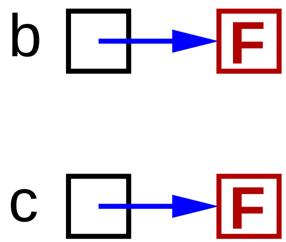
```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

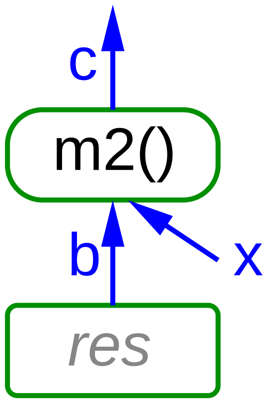




- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    $\Rightarrow$ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

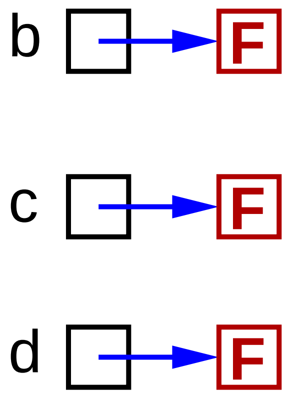


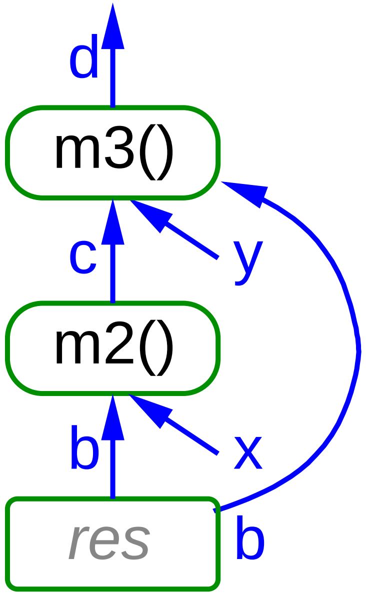


Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    $\Rightarrow$ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

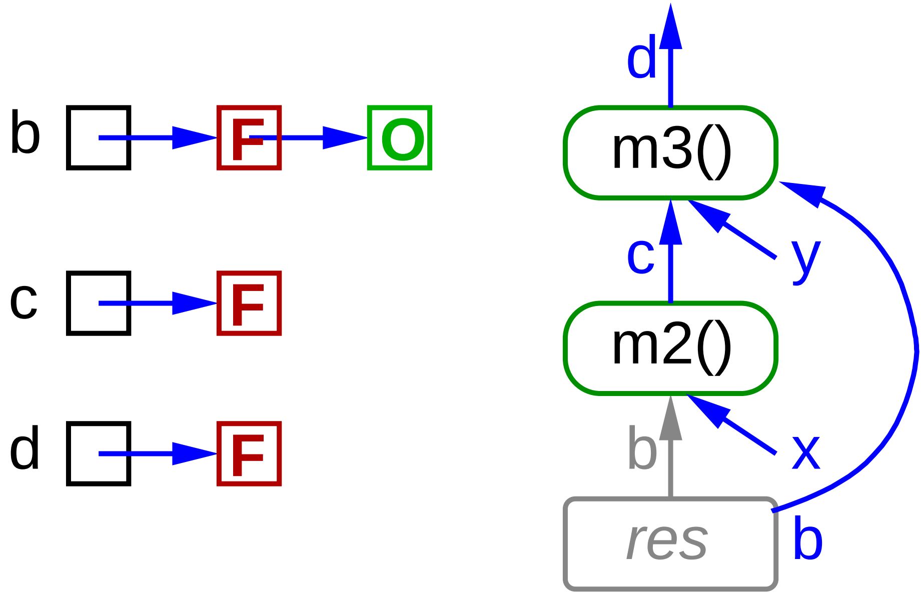




Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    $\Rightarrow$ we end up with a data flow graph

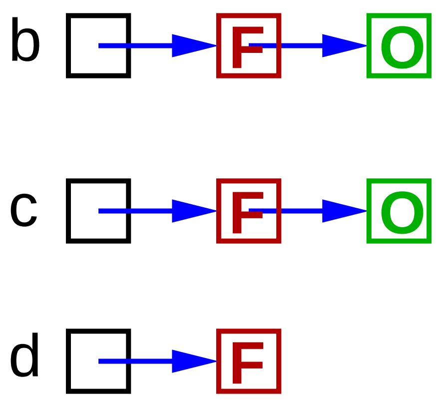
```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

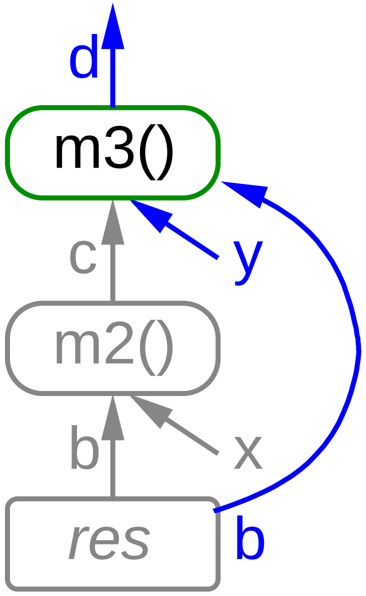


Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    ⇒ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

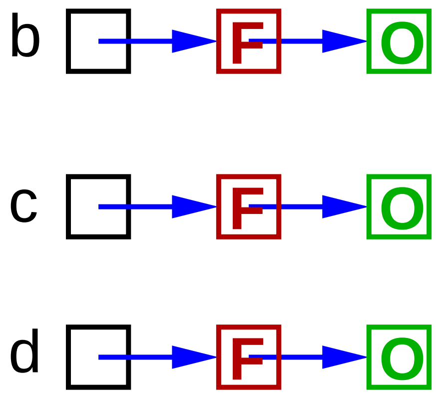


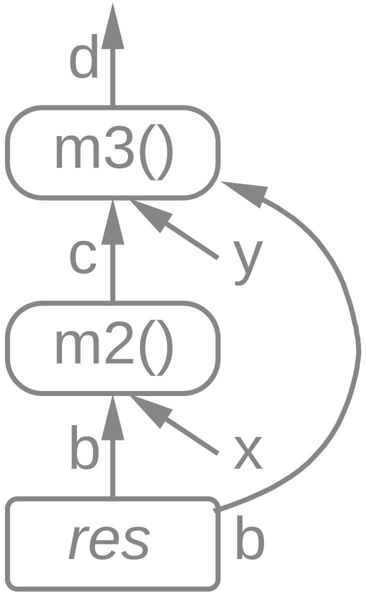


Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    $\Rightarrow$ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

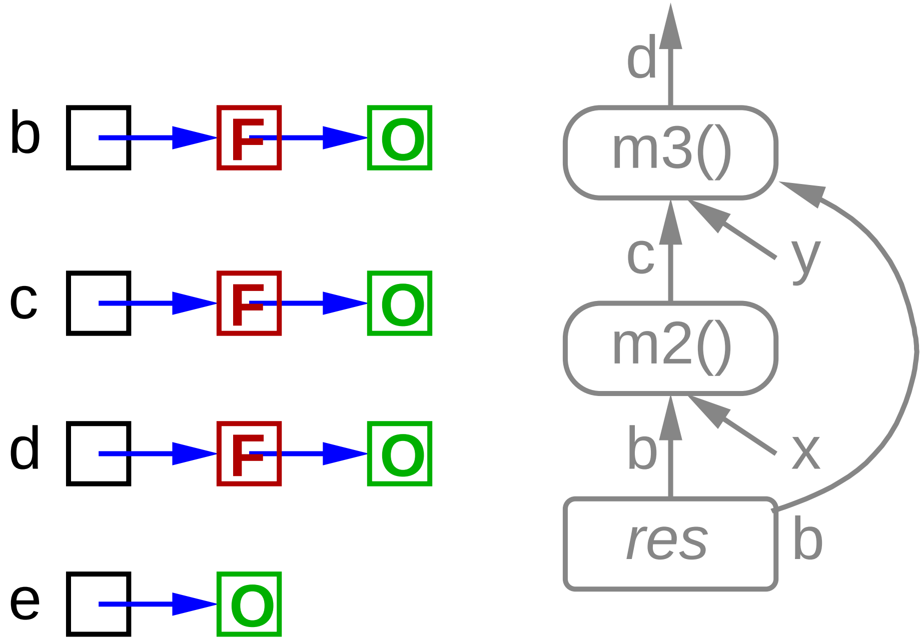




Towards an Implementation (2) ...

- Observation: invoking a method on a future doesn't have to block
  - we can immediately return another future as the result
  - but we have to remember to method to be executed
    ⇒ we end up with a data flow graph

```
b = a.m1();
...
c = b.m2(x);
...
d = c.m3(y,b);
...
e = d.m4();
```

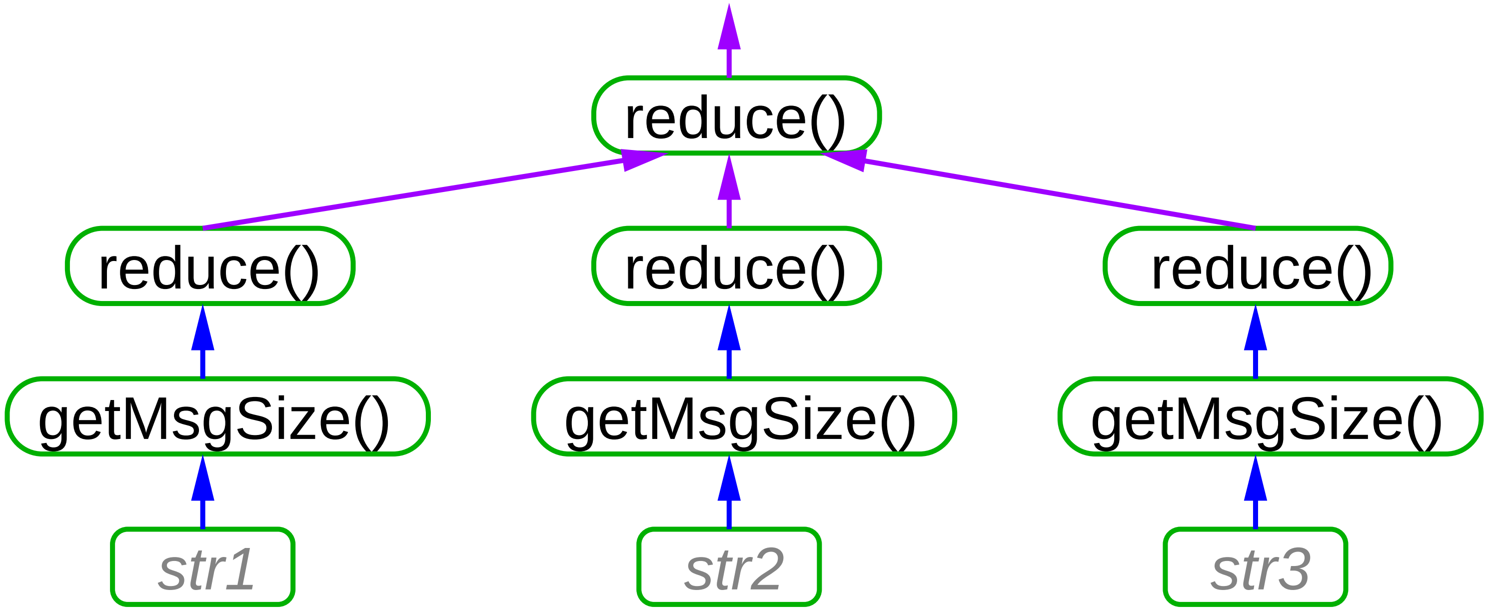


Proof-of-Concept Implementation

- Using ProActive run time code generator for proxy classes
  - can generate a proxy class for every non-final class
- Three kinds of proxies:
  - future proxy for asynchronous RMIs
    - method call results in creation of a data flow node, if object is not yet available
  - group proxy for sets
    - basically identical to ProActive
  - stream proxy
    - invokes method on each object in the stream, as it arrives
    - method result again is a stream
    - implemented using a data flow node, similar to future proxy
- Plus all kinds of combinations (e.g.: future group of streams)

The Example Revisited

```
Set<Node> nodes = System.getNodes(...);
Set<Processes> procs = nodes.getProcesses(...);
Set<Stream<SendEvent>> ev = procs.getSendEvents(...);
IntVal tot = Set.reduce(Stream.reduce(ev.getMsgSize(),
                                      SUM),SUM);
...
print(tot.getValue());
```



Conclusions / Status

- A “natural” object oriented model for online analysis is feasilbe
  - use transparent proxies to create data flow graphs
  - distribute the data flow graphs (and the code of the required classes) to the target system for execution
- Still many issues open for research:
  - semantics (method parameters, excution order, ...)
  - implementation of special functions
    - reductions, scatter, ...
  - best way to generate proxy classes
    - currently: set / stream of A is subclass of A
  - distribution of data flow graphs
    - esp. distribution of reduction methods