Using business workflows to improve descriptiveness and control of experiments

Tomasz Buchert

Advisors

Lucas Nussbaum

Jens Gustedt

Research in distributed systems

We all know how frustrating experimenting can be.

That's because experiments in distributed systems are:

- time-consuming
- difficult to do correctly
- complex and incomprehensible
- failure-prone

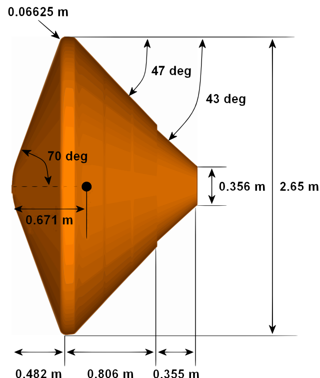
We should **automate experiments**:

- humans make mistakes
- humans tend to forget things
- knowledge dissipates with time

"*The primary decelerator for Phoenix was a rigid capsule with a 70-degree half-angle sphere-cone forebody (...)*"

Why this angle was chosen for Mars?

A simulation showed it is *optimal*.



Automation of system administration

With tools like Chef and Puppet:

- a human factor is nearly removed
- systems are built from modules
- the configuration is reproducible

But reproducibility does not necessarily imply **descriptiveness**.
It does not imply **ease of understanding** either.

Experimentation tools

Many tools to manage experiments exist:

- Expo
- g5k-campaign
- OMF
- Plush
- ... among many others

They are based on different paradigms.

Grid'5000: Bash, oarsub, kadeploy3, ...

```
[ rennes ˜ ] $ oarsub -I -l nodes=2 -t deploy
[ADMISSION RULE] Set default walltime to 3600.
[ADMISSION RULE] Modify resource description with type constraints
Generate a job key...
OAR_JOB_ID=454195
Interactive mode : waiting...
Starting...
Connect to OAR job 454195 via the node frontend.rennes.grid5000.fr
[ 454195@rennes ˜ ] $ kadeploy3 -e squeeze-x64-nfs -f $OAR_FILE_NODES -k
Launching a deployment ...
Grab the key file /home/tbuchert/.ssh/authorized_keys
Performing a deploy step on the nodes: paradent-[17-18].rennes.grid5000.fr
--- switch_pxe (paradent cluster)
  >>>  paradent-[17-18].rennes.grid5000.fr
--- set_vlan (paradent cluster)
  >>>  paradent-[17-18].rennes.grid5000.fr
  *** Bypass the VLAN setting
--- reboot (paradent cluster)
  >>>  paradent-[17-18].rennes.grid5000.fr

... and so on.
```

Repetitive and error-prone if done without attention.

g5k-campaign

```
class MyEngine < Grid5000::Campaign::Engine

    set :environment, "lenny-x64-base"
    set :resources, "nodes=4"
    set :walltime, 7200

    on :install! do |env, *args|
      nodes = env[:nodes]
      ssh(nodes.first, "root") do |ssh|
        ssh.exec!("apt-get install whatever")
        ssh.sftp.upload!("my_engine/some/file", "/destination/path")
      end
      env
    end

    on :execute! do |env, *args|
      env[:nodes].each { |node|
        ssh(node, "root") do |ssh|
          ssh.exec!("./my-program")
        end
      }
      env
    end
  end
```

Manages reservation, but what remains is still low-level.

Expo

```
reserv = ExpoEngine::new(@connection)
reserv.site = [ "bordeaux", "lille", "luxembourg", "nancy", "sophia" ]
reserv.resources = [ "nodes=50", "nodes=10", "nodes=4", "nodes=4", "nodes=30" ]
reserv.name = "Expo Scalability"
reserv.walltime = 600

reserv.run!

sizes = [ 10, 20, 40, 50, 80, $all.length ]

$all.each_slice_array(sizes) do |nodes|

  task_mon = Task::new("hostname", nodes, " Monitoring #{nodes.length} nodes")
  10.times {
    id, res = task_mon.execute
    puts " #{res.length} : #{res.duration}"
  }

end

reserv.stop!
```

Features useful abstractions and efficiency, but not high-level.

Gush (Plush)

```
<plush><project name="simple">
    <software name="SimpleSoftwareName" type="none">
      <package name="Package" type="web">
          <path>http://sysnet.cs.williams.edu/~jeannie/software.tar</path>
        <dest_path>software.tar</dest_path>
      </package>
    </software>
    <component name="Cluster1">
      <rspec><num_hosts>2</num_hosts></rspec>
      <software name="SimpleSoftwareName" />
      <resources><resource type="ssh" group="local"/></resources>
    </component>
    <experiment name="simple">
      <execution>
        <component_block name="cb1">
          <component name="Cluster1" />
          <process_block name="p2">
            <process name="cat">
              <path>cat</path><cmdline><arg>software.txt</arg></cmdline><cwd/>
            </process>
          </process_block>
        </component_block>
      </execution>
    </experiment>
  </project>
</plush>
```

Declarative, but rather difficult to write and understand.

OMF

```
defGroup('Sender', "omf.nicta.node28") do |node|
  node.addApplication("test:app:otg2") do |app|
    app.setProperty('udp:local_host', '192.168.0.2')
    app.setProperty('udp:dst_host', '192.168.0.3')
    app.setProperty('udp:dst_port', 3000)
    app.measure('udp_out', :samples => 1)
  end
  w0 = node.net.w0; w0.mode = "adhoc"; w0.type = 'g'
  w0.channel = "6"; w0.essid = "helloworld"; w0.ip = "192.168.0.2"
end

defGroup('Receiver', "omf.nicta.node29") do |node|
  node.addApplication("test:app:otr2") do |app|
    app.setProperty('udp:local_host', '192.168.0.3')
    app.setProperty('udp:local_port', 3000)
    app.measure('udp_in', :samples => 1)
  end
  w0 = node.net.w0; w0.mode = "adhoc"; w0.type = 'g'
  w0.channel = "6"; w0.essid = "helloworld"; w0.ip = "192.168.0.3"
end

onEvent(:ALL_UP_AND_INSTALLED) do |node|
  info "This is my first OMF experiment"; wait 10; allGroups.startApplications
  info "All my Applications are started now..."; wait 30; allGroups.stopApplications
  info "All my Applications are stopped now."; Experiment.done
end
```

Declarative and event-driven but cannot express complex workflows.

Bottom-up vs top-down approach

Most of these tools use **bottom-up design**.

What about a **top-down** approach?

1. Start with high-level description of the experiment.
2. Implement low-level details.
3. Run the experiment.
4. Improve if necessary and reiterate.

There already exists an approach like this.

Business Process Management

Business Process Management is about:

- understanding an organization
- modeling its processes as **workflows**
- **executing** processes and **monitoring** them
- **improving** organizational **activities**
- redesigning **processes** to make them:
  - cheaper
  - faster
  - less defective

XPflow

In this talk I will present **XPflow**:

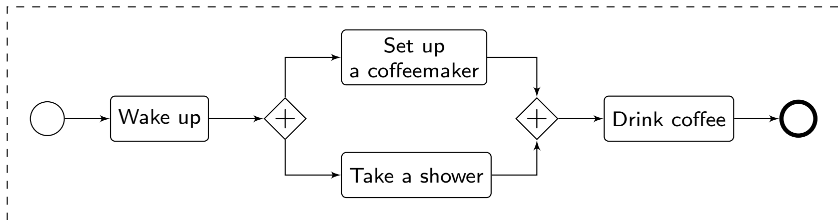
- a new experimentation engine
- based on **Business Process Modeling** and **Management**

It will be illustrated with a Grid'5000-based experiment.

XPflow concepts

There are 2 main concepts in XPflow:

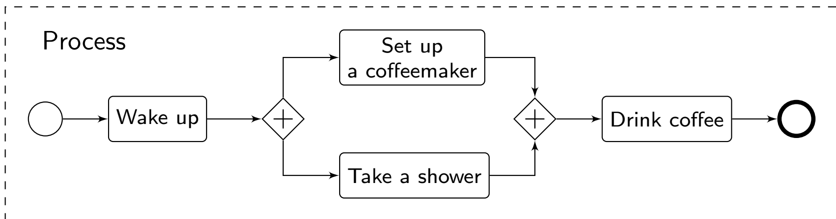
- Processes – high-level description of an experiment:
  - workflows written in a DSL
  - orchestrate other processes and activities
- Activities – low-level building blocks of experiments:
  - do real hard work
  - written in Ruby



XPflow concepts

There are 2 main concepts in XPflow:

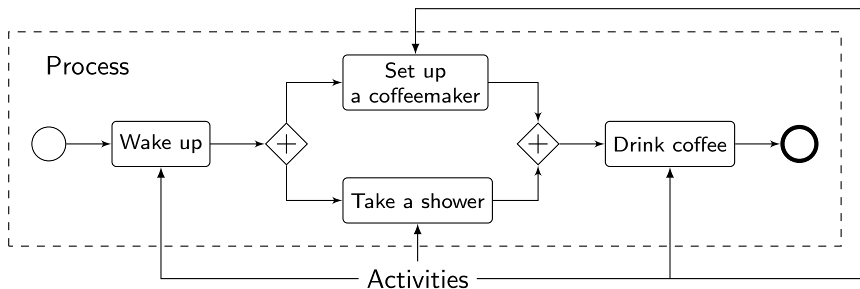
- Processes – high-level description of an experiment:
  - workflows written in a DSL
  - orchestrate other processes and activities
- Activities – low-level building blocks of experiments:
  - do real hard work
  - written in Ruby



XPflow concepts

There are 2 main concepts in XPflow:

- Processes – high-level description of an experiment:
  - workflows written in a DSL
  - orchestrate other processes and activities
- Activities – low-level building blocks of experiments:
  - do real hard work
  - written in Ruby



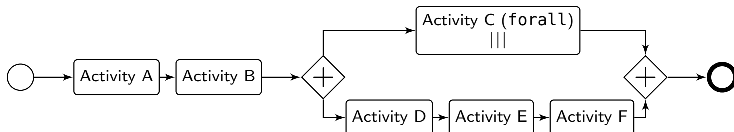
Domain-specific language for processes

The DSL for processes features different **workflow patterns**:

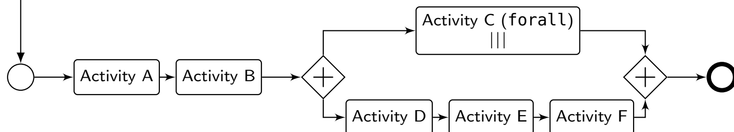
- running activities and other processes (`run`),
- running activities in order or in parallel (`sequence`, `parallel`),
- conditional expressions (`if`, `switch`)
- running sequential and parallel loops (`loop`, `foreach`, `forall`),
- error handling (`try`, `checkpoint`).

Some of them are taken directly from BPM.

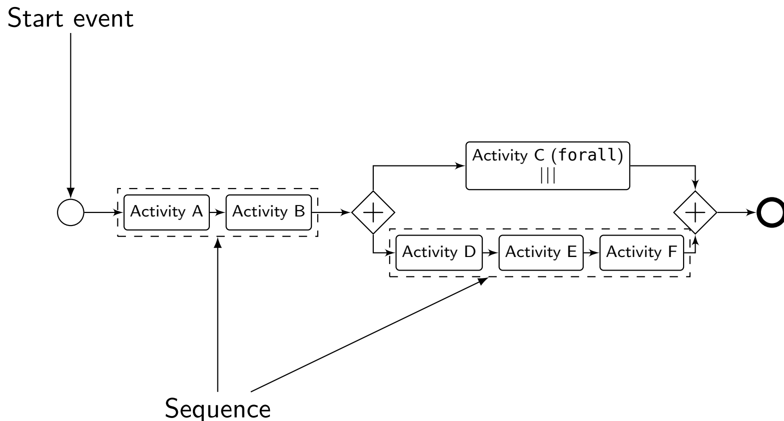
Workflow patterns (example)



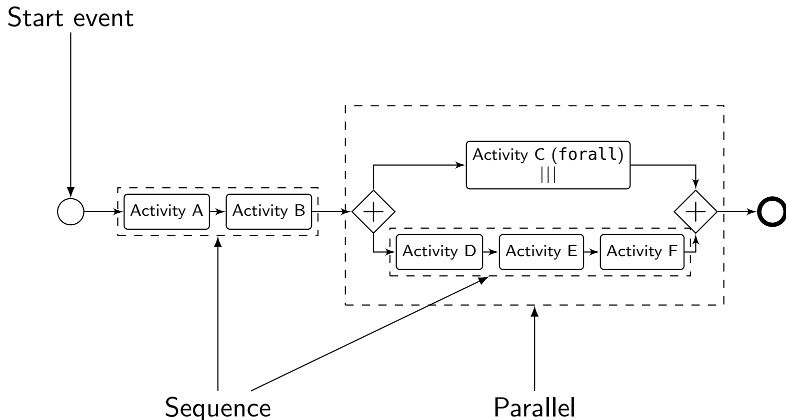
Workflow patterns (example)



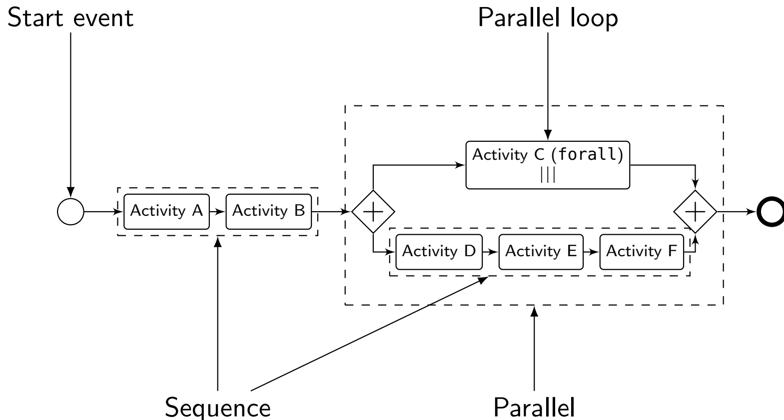
Workflow patterns (example)



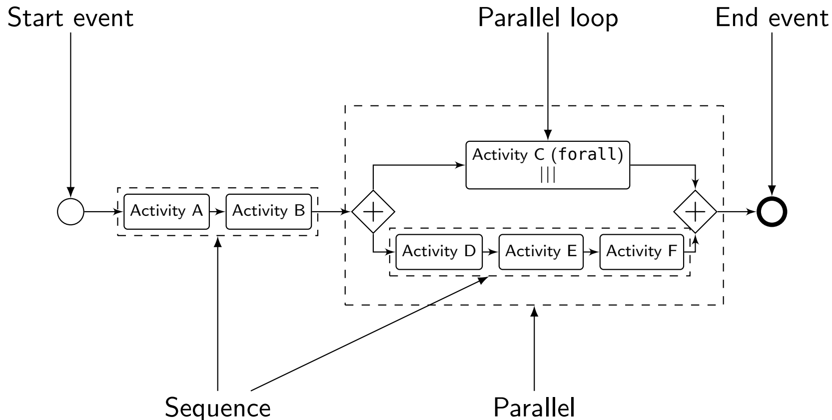
Workflow patterns (example)



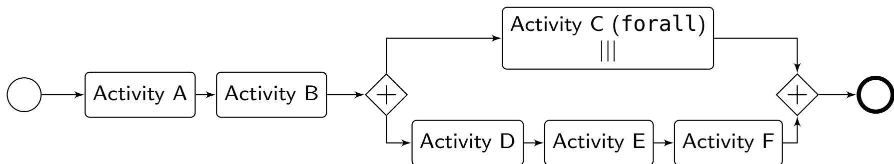
Workflow patterns (example)



Workflow patterns (example)

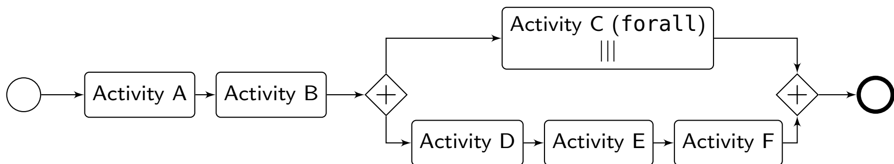


Workflow patterns (example, cont.)



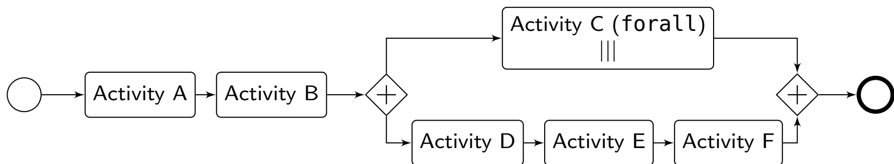

```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)



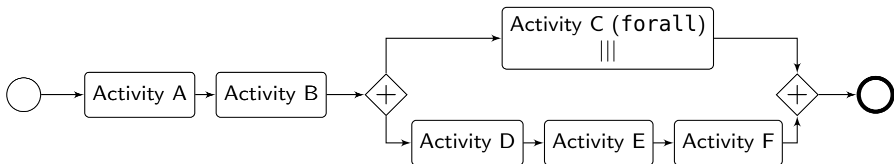

```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)



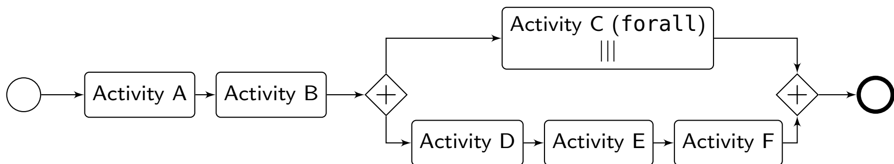

```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)



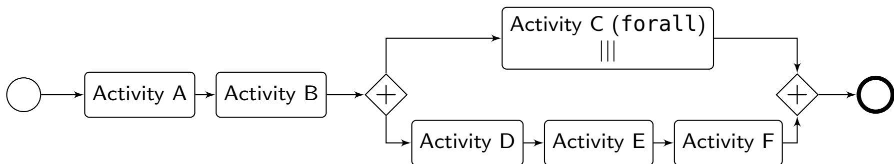

```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)



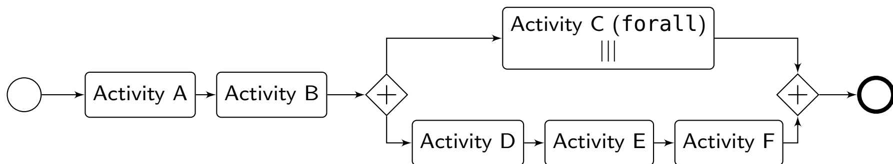

```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)




```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Workflow patterns (example, cont.)




```
engine.process :workflow do |array|
    run :a
    run :b
    parallel do
        forall array do |x|
            run :c, x
        end
        sequence do
            run :d
            run :e
            run :f
        end
    end
end
```

Error handling

XPflow gives some means to cope with failures:

- snapshotting:
  - saves a state of an experiment for future use
  - shortens a development's cycle
- retry policy:
  - retries a failed subprocess execution
  - improves reliability

```
engine.process :snapshotting do
    run :long_deployment
    checkpoint :d
    run :experiment
end
```

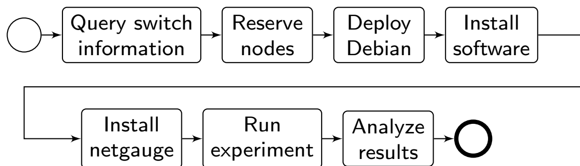
```
engine.process :retrying do
    try :retry => 5 do
        run :tricky_activity
    end
end
```

Both features require **idempotency**.

Example of an experiment

Measure the *effective bisection bandwidth* of a switch.

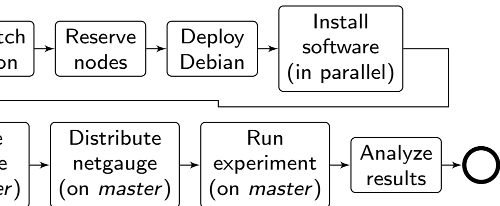
1. Get names of all nodes connected to the switch.
2. Reserve the nodes.
3. Deploy Debian OS.
4. Install necessary software.
5. Compile and install *netgauge*.
6. Run the experiment.
7. Analyze results.



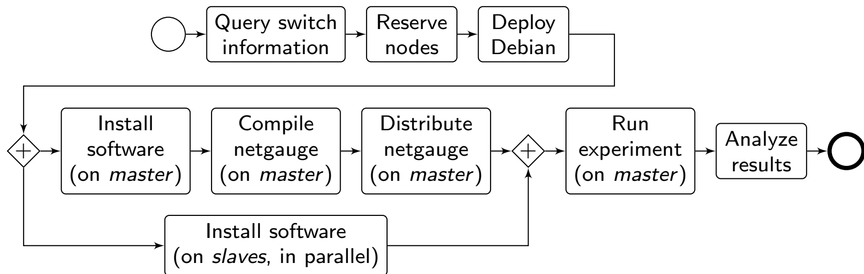
Few notes:

- each node must have some software installed
- each node must have *netgauge* installed ...
- ... but one node is enough to compile it
- one node must launch MPI application

We will introduce a *master* node and *slave* nodes.



Another observation: compilation can run in parallel with installation of software on the *slave* nodes.


Query switch information
Reserve nodes
Deploy Debian
Install software (on *master*)
Compile netgauge (on *master*)
Distribute netgauge (on *master*)
Run experiment (on *master*)
Analyze results
Install software (on *slaves*, in parallel)

An experiment workflow - DSL representation

```
engine.process :exp do |site, switch|
    s = run g5k.switch, site, switch
    ns = run g5k.nodes, s
    r = run g5k.reserve_nodes,
        :nodes => ns, :time => '2h',
        :site => site, :type => :deploy
    master = (first_of ns)
    rest = (tail_of ns)
    run g5k.deploy,
        r, :env => 'squeeze-x64-nfs'
    checkpoint :deployed
    parallel :retry => true do
        forall rest do |slave|
            run :install_pkgs, slave
        end
        sequence do
            run :install_pkgs, master
            run :build_netgauge, master
            run :dist_netgauge,
                master, rest
        end
    end
    checkpoint :prepared
    output = run :netgauge, master, ns
    checkpoint :finished
    run :analysis, output, switch
end
```

An experiment workflow - DSL representation

```
engine.process :exp do |site, switch|
    s = run g5k.switch, site, switch
    ns = run g5k.nodes, s
    r = run g5k.reserve_nodes,
        :nodes => ns, :time => '2h',
        :site => site, :type => :deploy
    master = (first_of ns)
    rest = (tail_of ns)
    run g5k.deploy,
        r, :env => 'squeeze-x64-nfs'
    checkpoint :deployed
    parallel :retry => true do
        forall rest do |slave|
            run :install_pkgs, slave
        end
        sequence do
            run :install_pkgs, master
            run :build_netgauge, master
            run :dist_netgauge,
                master, rest
        end
    end
    checkpoint :prepared
    output = run :netgauge, master, ns
    checkpoint :finished
    run :analysis, output, switch
end
```

Activity :install_pkgs

```
engine.activity :install_pkgs do|node|
    log 'Installing packages on ', node
    run 'g5k.bash', node do
        aptget :update
        aptget :upgrade
        aptget :purge, 'mx'
    end
end
```

An experiment workflow - DSL representation

```
engine.process :exp do |site, switch|
    s = run g5k.switch, site, switch
    ns = run g5k.nodes, s
    r = run g5k.reserve_nodes,
        :nodes => ns, :time => '2h',
        :site => site, :type => :deploy
    master = (first_of ns)
    rest = (tail_of ns)
    run g5k.deploy,
        r, :env => 'squeeze-x64-nfs'
    checkpoint :deployed
    parallel :retry => true do
        forall rest do |slave|
            run :install_pkgs, slave
        end
        sequence do
            run :install_pkgs, master
            run :build_netgauge, master
            run :dist_netgauge,
                master, rest
        end
    end
    checkpoint :prepared
    output = run :netgauge, master, ns
    checkpoint :finished
    run :analysis, output, switch
end
```

Activity :build_netgauge

```
engine.activity :build_netgauge do |master|
    log "Building netgauge on #{master}"
    run 'g5k.copy', NETGAUGE, master, '~'
    run 'g5k.bash', master do
        build_tarball NETGAUGE, PATH
    end
    log "Build finished."
end
```

An experiment workflow - DSL representation

```
engine.process :exp do |site, switch|
    s = run g5k.switch, site, switch
    ns = run g5k.nodes, s
    r = run g5k.reserve_nodes,
        :nodes => ns, :time => '2h',
        :site => site, :type => :deploy
    master = (first_of ns)
    rest = (tail_of ns)
    run g5k.deploy,
        r, :env => 'squeeze-x64-nfs'
    checkpoint :deployed
    parallel :retry => true do
        forall rest do |slave|
            run :install_pkgs, slave
        end
        sequence do
            run :install_pkgs, master
            run :build_netgauge, master
            run :dist_netgauge,
                master, rest
        end
    end
    checkpoint :prepared
    output = run :netgauge, master, ns
    checkpoint :finished
    run :analysis, output, switch
end
```

Activity :dist_netgauge

```
engine.activity :dist_netgauge do |m, s|
    master, slaves = m, s
    run 'g5k.dist_keys', master, slaves
    run 'g5k.bash', master do
        distribute BINARY,
            DEST, 'localhost', slaves
    end
end
```

An experiment workflow - DSL representation

```
engine.process :exp do |site, switch|
    s = run g5k.switch, site, switch
    ns = run g5k.nodes, s
    r = run g5k.reserve_nodes,
        :nodes => ns, :time => '2h',
        :site => site, :type => :deploy
    master = (first_of ns)
    rest = (tail_of ns)
    run g5k.deploy,
        r, :env => 'squeeze-x64-nfs'
    checkpoint :deployed
    parallel :retry => true do
        forall rest do |slave|
            run :install_pkgs, slave
        end
        sequence do
            run :install_pkgs, master
            run :build_netgauge, master
            run :dist_netgauge,
                master, rest
        end
    end
    checkpoint :prepared
    output = run :netgauge, master, ns
    checkpoint :finished
    run :analysis, output, switch
end
```

Activity :netgauge

```
engine.activity :netgauge do |master, nodes|
    log "Running experiment..."
    out = run 'g5k.bash', master do
        cd PATH
        mpirun nodes, "./netgauge"
    end
    log "Experiment done."
end
```

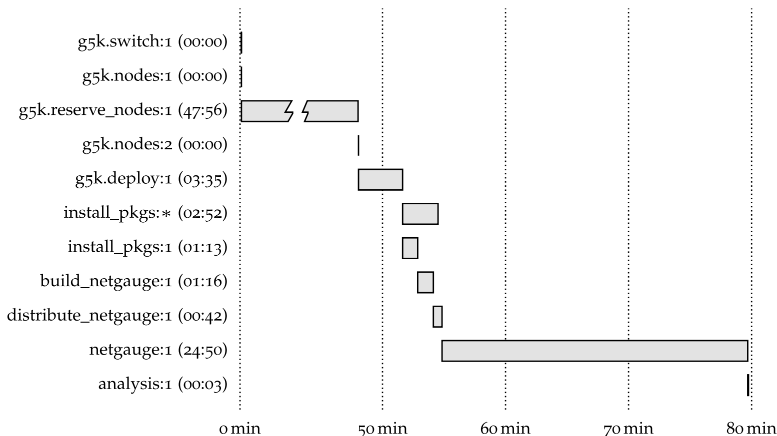
Running the experiment

The experiment runs on Grid'5000 frontend or on your local machine.

```
[ 11:15:52.940 ] Started activity g5k.switch:1.
[ 11:15:53.418 ] Finished activity g5k.switch:1 (0.478 s).
[ 11:15:53.419 ] Process exp: Experimenting with switch: sgraphene2
[ 11:15:53.419 ] Started activity g5k.nodes:1.
[ 11:15:53.419 ] Finished activity g5k.nodes:1 (0.000 s).
[ 11:15:53.419 ] Started activity g5k.reserve_nodes:1.
[ 11:15:55.837 ] Waiting for reservation 408387
[ 11:16:02.452 ] Reservation 408387 should be available in 12 mins
[ 11:16:02.452 ] Reservation 408387 ready
[ 11:16:02.453 ] Finished activity g5k.reserve_nodes:1 (9.022 s).
[ 11:16:02.453 ] Started activity g5k.nodes:2.
[ 11:16:02.453 ] Finished activity g5k.nodes:2 (0.000 s).
[ 11:16:02.453 ] Started activity g5k.deploy:1.
[ 11:22:09.427 ] Finished activity g5k.deploy:1 (366.968 s).
[ 11:22:09.429 ] Started activity install_pkgs.
[ 11:22:09.429 ] Started activity install_pkgs:1.
[ 11:22:09.430 ] Activity install_pkgs: Installing packages on graphene-96
[ 11:22:09.430 ] Started activity install_pkgs:2.
[ 11:22:09.430 ] Activity install_pkgs: Installing packages on graphene-60
```

The execution is monitored and errors reported if necessary.

Monitoring features - Gantt chart of the execution



Each activity is monitored during its execution.

Notice that `build_netgauge:1` runs in parallel with `install_pkgs:∗`.

Conclusions

In this talk I presented XPflow.

Current features include:

- improved descriptiveness
- modularity and flexibility
- monitoring and support for common patterns
- integration with Grid'5000

In the future we will work on:

- integration with experimentation tools
- human interaction during the experiment
- efficient data broadcast and collection

Thank you for your attention. Questions?