An introduction to supervision & deployment in Rock

Sylvain Joyeux

DFKI Bremen - Forschungruppe Robotik
& Universität Bremen
Director: Prof. Dr. Frank Kirchner
www.dfki.de/robotics
robotics@dfki.de

Outline

1. Introduction

What will you learn today

- explain what the supervision layer does
- . . . and how to control your robots with it

The main goal is to get you understand the basics of this tool

Building blocks

- single components will often not be usable by themselves
  - example: pose estimation component, image processing component
  - counter-example: devices
- we must describe functional services
  - $\Rightarrow$ a group of components that, together, do something

The goal

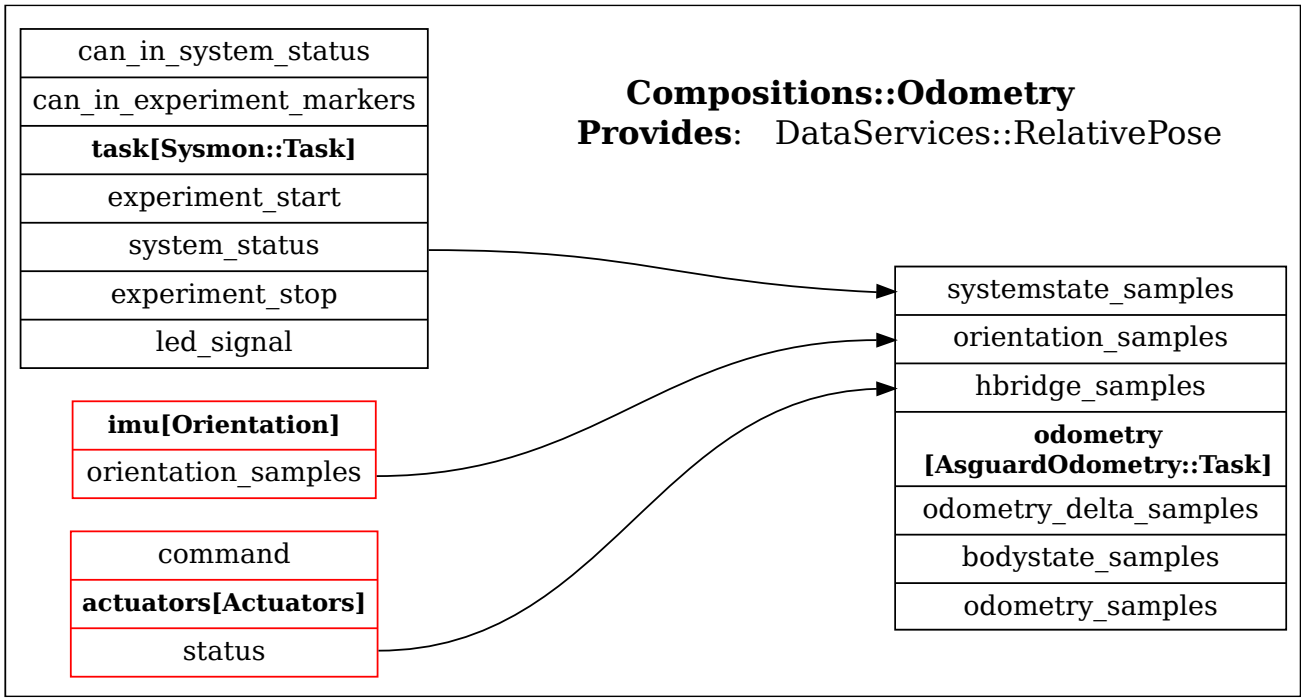
Describe these functional services so that

- one can cherry-pick functionality
- one can recognize identical things done in different ways
- one can track errors

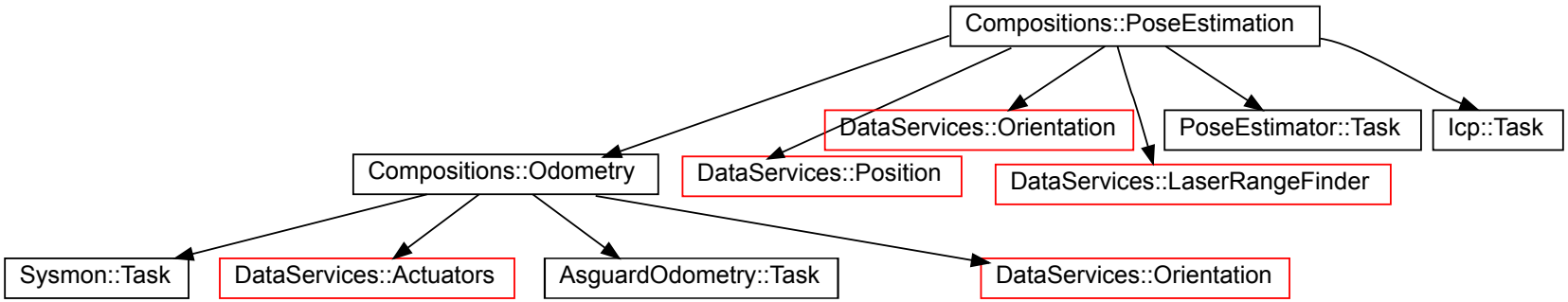
Compositions and data services

compositions group of components that provide a function

data services generic “placeholders” for components and/or other compositions (i.e. Orientation for all orientation providers)



Dependency structure



2. Digging in: concepts

Models and instances

Model gives information about a category of “things”

Instance a “thing” constructed based on information in the corresponding model

Examples

	Models	**Instances**
OO programming	Classes	Objects
Type systems	Types	Values
oroGen	Task context model	Deployed task

Models and instances in Typelib & Roby

Typelib

Type	subclass of Typelib::Type
Value	instance of a subclass of Typelib::Type

Orocos/Roby

Task context definition	subclass of TaskContext
Deployed task	instance of a subclass of TaskContext

What does it mean ?

To add information to a model, you call class methods. To modify the instances, you define methods on the class.

Naming

- Ruby's accepted naming scheme is UpperCamelCase for classes and modules, UPPER_CASE for constant values and snake_case for everything else
- this is used in the instance / model scheme in the supervision (we'll see examples when they appear)

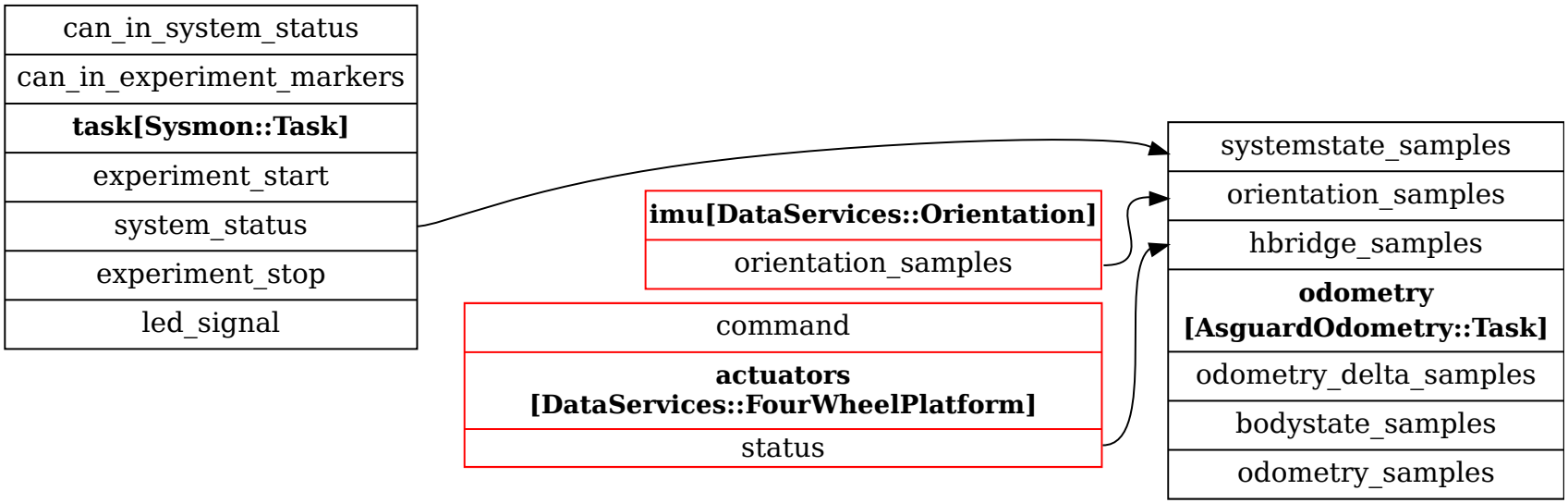
What is it going to be ?

- building compositions (modelling)
- deploying these compositions (running)
- configuration, reconfiguration & tuning

3. Modelling

The goal

We're going to deploy asguard's odometry



Model files

- contain definitions of data services and compositions (and some other things . . .)
- load each other with load_system_model(”filename”)
- if models from oroGen files are needed, use using_task_library ”orogen_project_name”

Naming: oroGen → Roby

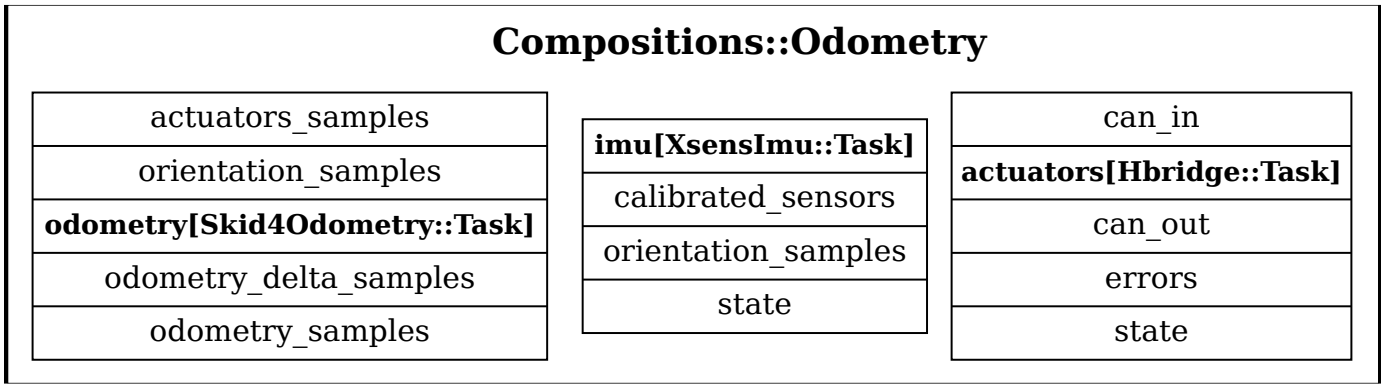
```
name 'xsens_imu'
task_context 'Task' do
end
```

```
using_task_library 'xsens_imu'
# Defines
# – the XsensImu module
# – the XsensImu::Task model
```

Creating compositions

```
using_task_library 'skid4_odometry'
using_task_library 'xsens_imu'
using_task_library 'hbridge'
composition 'Odometry' do
  add Hbridge::Task, :as => 'actuators'
  add XsensImu::Task, :as => 'imu'
  add Skid4Odometry::Task, :as => 'odometry'
end
```

⇒ `Compositions::Odometry` composition model



Child names

```
using_task_library 'skid4_odometry'
using_task_library 'xsens_imu'
using_task_library 'hbridge'
composition 'Odometry' do
 add Hbridge::Task, :as => 'actuators'
 add XsensImu::Task, :as => 'imu'
 add Skid4Odometry::Task, :as => 'odometry'
end
```

- Implicit name generated as snake_case version of the child model name

  `Hbridge::Task` → `task`, `XsensImu::Task` → `task`, ...
- Explicit names given with the :as option

Adding the dataflow connections

- connections are specified per compositions
  - ⇒ the tool makes sure that the dataflow is consistent at a system level

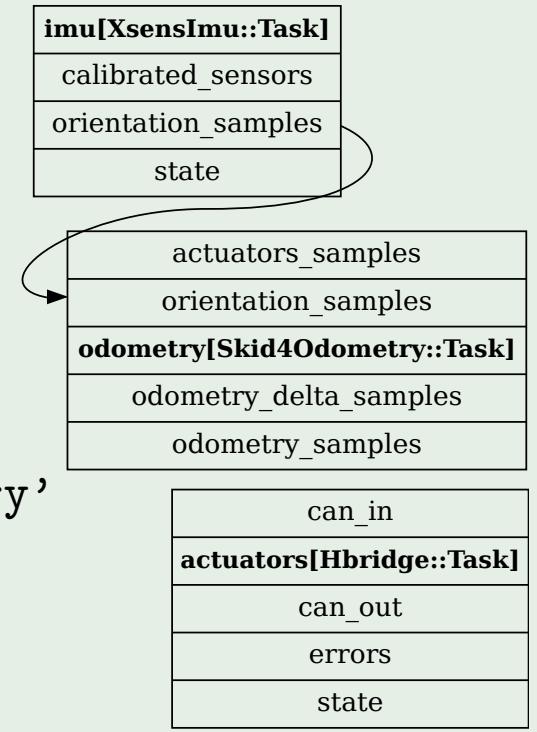
Autoconnections

- candidates are searched by port type and then port name
- ports that are involved in manual connection are automatically excluded from autoconnect
- if an ambiguity exists, an error is generated

Adding the dataflow connections

Autoconnections (contd.)

```
using_task_library 'skid4_odometry'
using_task_library 'xsens_imu'
using_task_library 'hbridge'
composition 'Odometry' do
  add Hbridge::Task, :as => 'actuators'
  add XsensImu::Task, :as => 'imu'
  add Skid4Odometry::Task, :as => 'odometry'
  autoconnect
end
```



Adding the dataflow connections

Manual connections

```
using_task_library 'skid4_odometry'
using_task_library 'xsens_imu'
using_task_library 'hbridge'
composition 'Odometry' do
  add Hbridge::Task, :as => 'actuators'
  imu = add XsensImu::Task, :as => 'imu'
  odometry = add Skid4Odometry::Task, :as => 'odometry'
  connect imu.orientation_samples => odometry.orientation_samples
end
```

What's the issue here ?

- a plain hbridge task has no way to control / read status of motors
- the solution is to
  - use an abstract "actuators" service
  - tell the system that the hbridge on asguard has one of those
  - and let it use the hbridge as the odometry's actuator

Generalization

Two tools: data services and specializations

- data services represent generic placeholders
- specializations allow to refine the definition of compositions to suit specific needs

The goal

- share composition models across systems
- allow to recognize the software structure easily
  - ⇒ common structure *is* represented in a common way
- allow tools to recognize the structure
  - ⇒ a pose estimator is a pose estimator *everywhere*
- factor out runtime management code
  - ⇒ if you a monitoring routine common to all pose estimators, it needs to be defined only once

Data Services

- they describe a functionality
- they describe an interface
- they describe relationships between themselves

```
import_types_from 'base'
data_service_type 'Actuators' do
    input_port("command", "base/actuators/Command")
    output_port("status", "base/actuators/Status")
end
data_service_type 'FourWheelPlatform', :provides => Actuators
```

Data service models are stored in the DataServices namespace, i.e. DataServices::Actuators, DataServices::Status (DataServices:: can be shortened to Srv::)

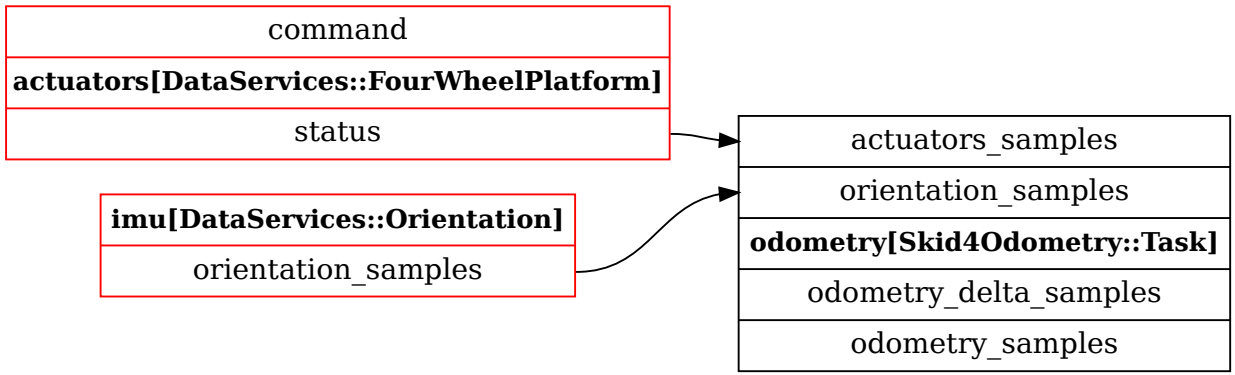
Relationships between services

```
data_service_type 'Position' do
    output_port 'position_samples', '/base/samples/RigidBodyState'
end
data_service_type 'Orientation' do
    output_port 'orientation_samples', '/base/samples/RigidBodyState'
end
data_service_type 'Pose' do
    output_port 'pose_samples', '/base/samples/RigidBodyState'
    provides Position, 'position_samples' => 'pose_samples'
    provides Orientation, 'orientation_samples' => 'pose_samples'
end
```

- Pose is providing both Position and Orientation
- the pose_samples port of Pose is to be used in place of either the position_samples port of Position or the orientation_samples port of Orientation

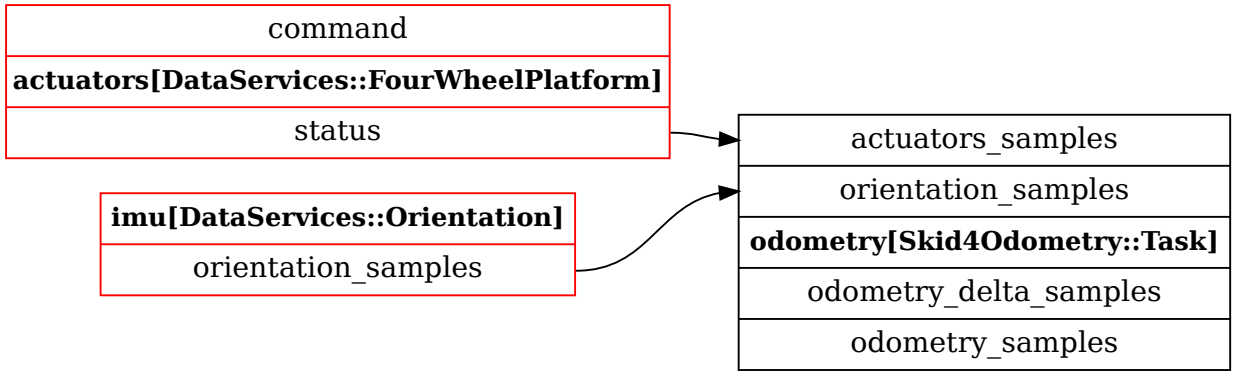
Composition with data service

```
using_task_library 'skid4_odometry'
composition 'Odometry' do
  add Srv::FourWheelPlatform, :as => 'actuators'
  add Srv::Orientation, :as => 'imu'
  add Skid4Odometry::Task, :as => 'odometry'
  autoconnect
end
```



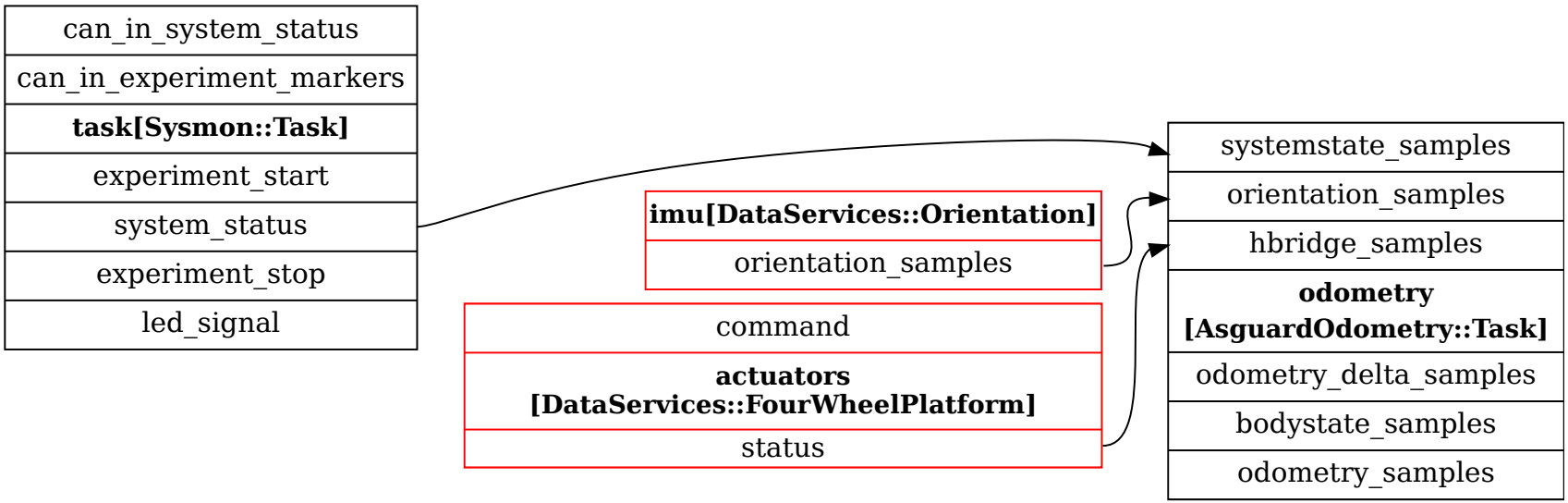
Composition Specialization

How to transform that



Composition Specialization

Into that (asguard's odometry)



Composition Specialization

- central change is that we use Odometry::Task instead of Skid4Odometry::Task
- but Odometry::Task is very much related to Skid4Odometry::Task from a conceptual point of view
- unfortunately, they are not related from an implementation POV

$\Rightarrow$ we first need to use a data service to generalize the odometry task

Composition Specialization

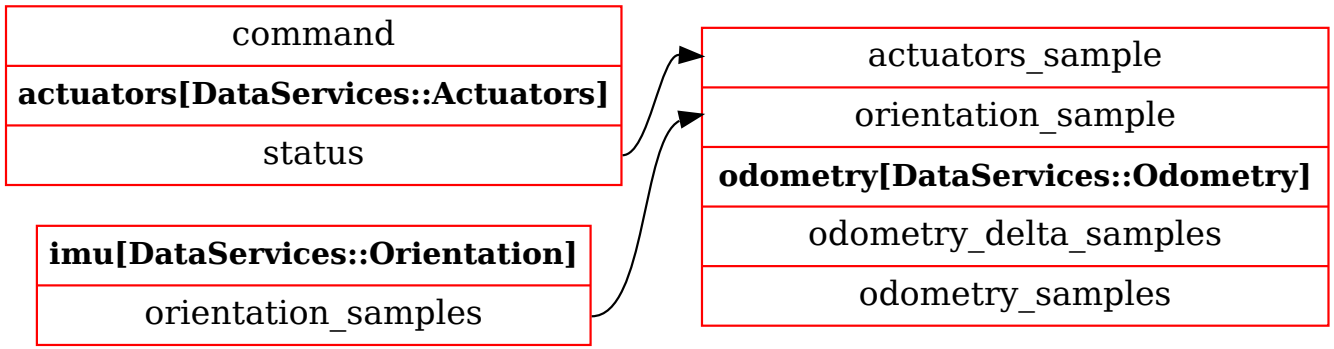
1. define the services

```
data_service_type 'RelativePose' do
  output_port 'odometry_delta_samples', '/base/samples/RigidBodyState'
  output_port 'odometry_samples', '/base/samples/RigidBodyState'
end
data_service_type 'Odometry' do
  input_port 'orientation_sample', '/base/samples/RigidBodyState'
  input_port 'actuators_sample', '/base/actuators/Status'
  provides RelativePose
end
```

Composition Specialization

2 replace Skid4Odometry::Task by Srv::Odometry in the composition

```
composition 'Odometry' do
  add Srv::Actuators, :as => 'actuators'
  add Srv::Orientation, :as => 'imu'
  odometry = add Srv::Odometry, :as => 'odometry'
  autoconnect
end
```



Composition Specialization

3 tell the supervision to add sysmon if odometry is an AsguardOdometry::Task

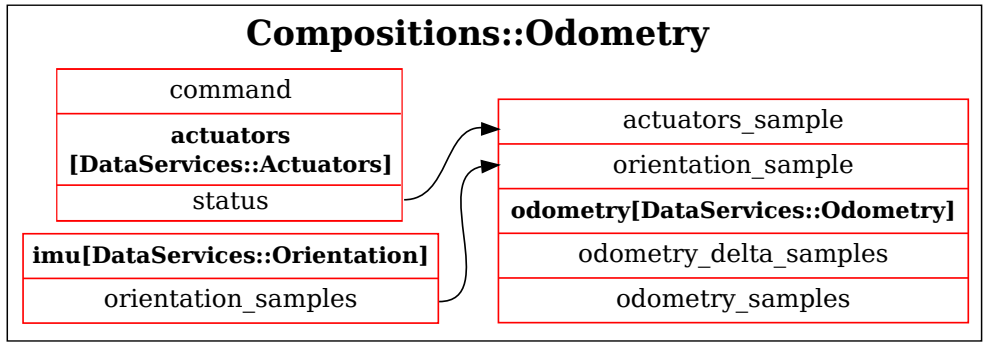
```
using_task_library 'odometry'
class AsguardOdometry::Task
  provides Srv::Odometry
end
Compositions::Odometry.specialize 'odometry' => AsguardOdometry::Task do
  add Sysmon::Task
  autoconnect
end
```

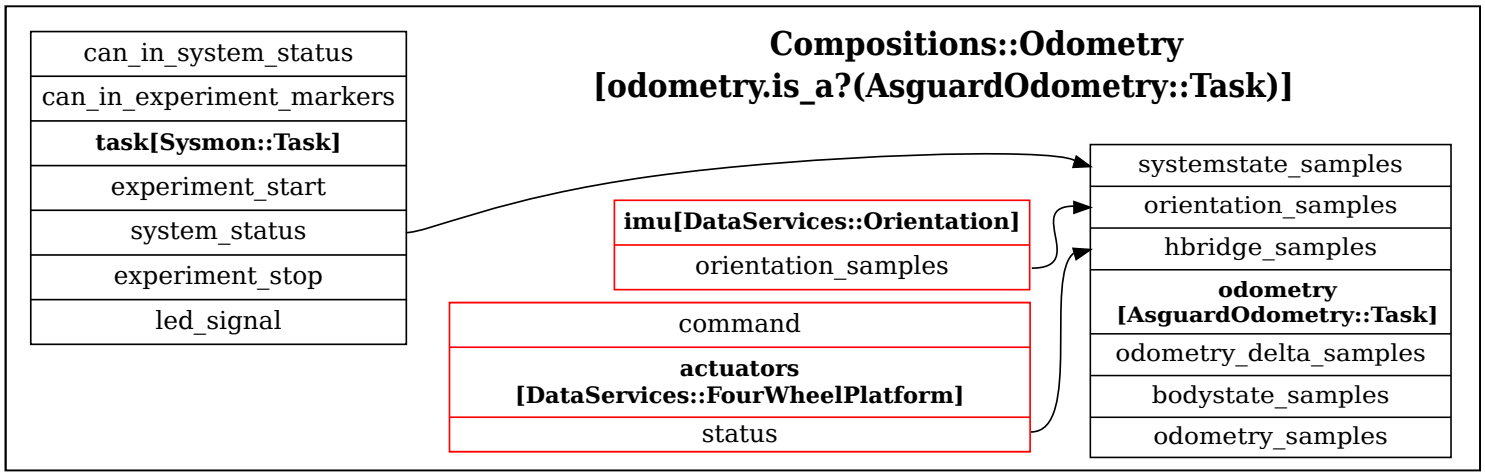
item autoconnect in the parent composition does not apply on the specializations

⇒ needs to be repeated

Composition Specialization

3 tell the supervision to add sysmon if odometry is an AsguardOdometry::Task





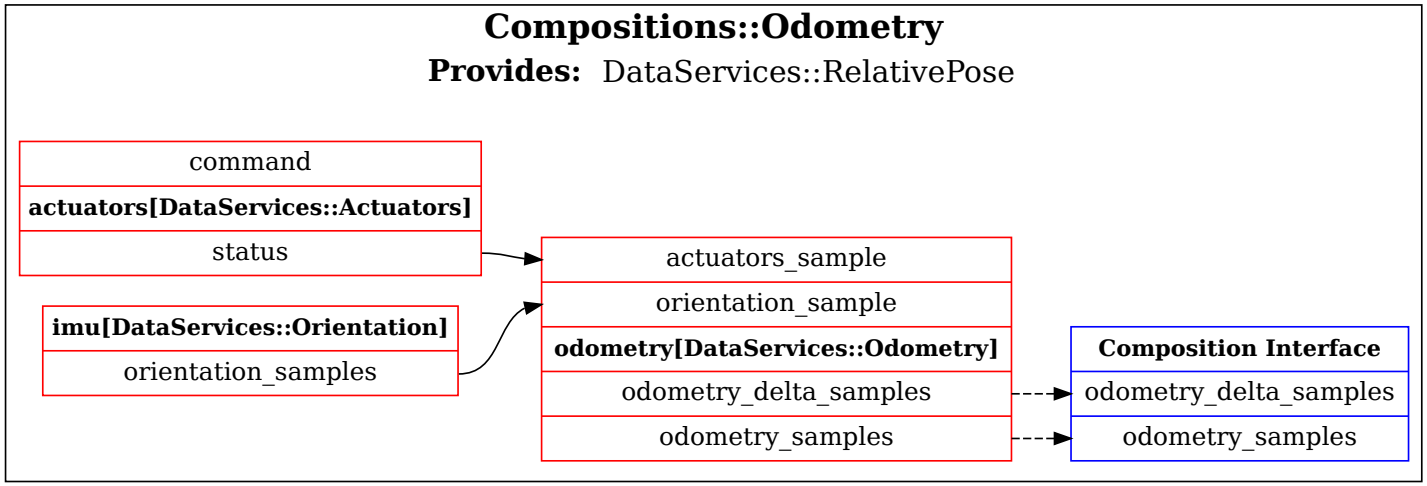
How to use compositions in compositions ?

- you use them directly
- you use them in place of a service
  - ⇒ how to make compositions services ?

```
composition 'Odometry' do
  add Srv::Actuators, :as => 'actuators'
  add Srv::Orientation, :as => 'imu'
  odometry = add Srv::Odometry, :as => 'odometry'
  autoconnect

  export odometry.odometry_delta_samples
  export odometry.odometry_samples
  provides Srv::RelativePose
end
```

How to use compositions in compositions ?



In the system model display

- dotted arrows represent exports
- blue box represents the composition's own interface

Your turn now !

- This was very dense (I know)
- Your turn: think about your system's architecture and what compositions you want to implement
- implement them without data services and without specializations at first

How ?

1. go into a new directory and
   `roby init -p orocos`
2. enable the orocos plugin by editing config/init.rb and add
   `Roby.app.using 'orocos'`

Standard file layout

tasks/data_services/ service definitions

tasks/compositions/ composition definitions

System model display tool

`scripts/orocos/system_model -o svg`

⇒ system_model.svg file, can be displayed by e.g. inkscape

A bit more on the file layout

tasks/orogen/ contains per-oroGen-project definitions

⇒ if the project ”xsens_imu” is loaded inside the supervision, then tasks/orogen/xsens_imu.rb will be loaded too

- used to bind tasks to generic services
- used to define specializations of generic compositions

Reusing stuff !

you can clone imoby's models from
`git://spacegit.dfki.uni-bremen.de/imoby/supervision.git`

- have a look at the complete system model with
  `scripts/orocos/system_model -r asguardv3 -o svg`
- what can you reuse / what patterns are common with your system ?
- try reusing them by defining e.g. specializations

4. Deployment

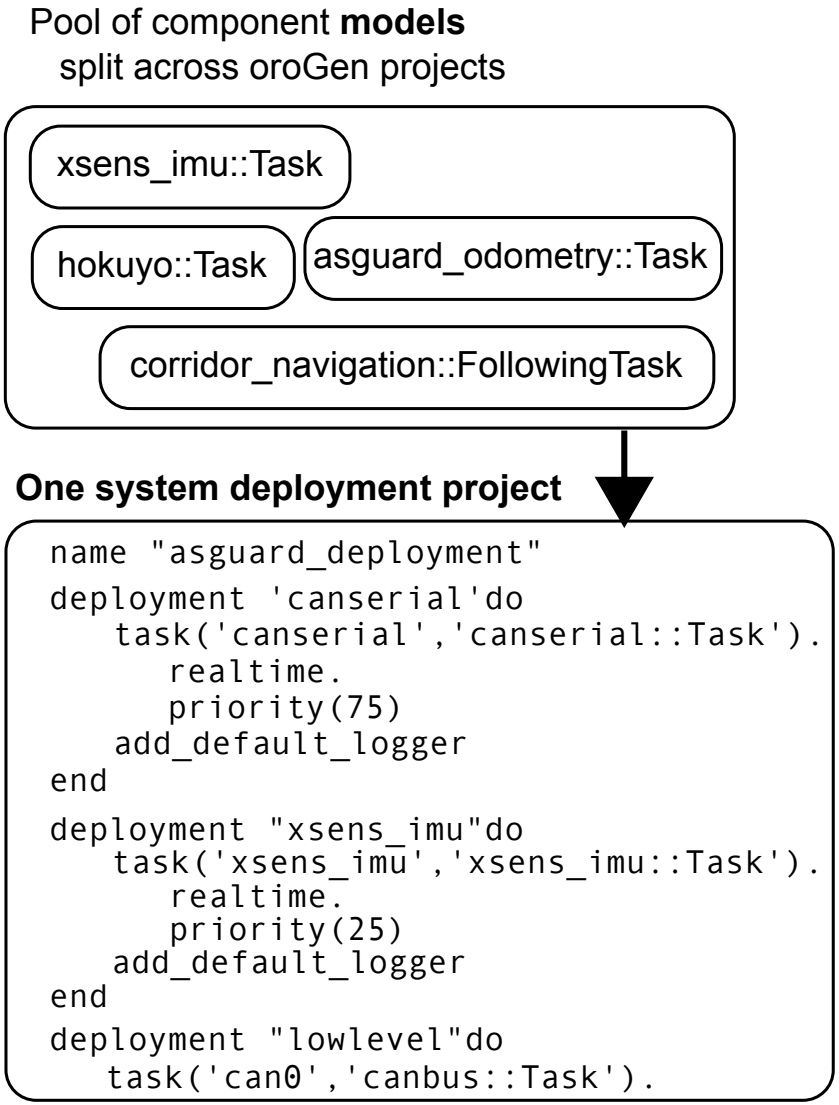
Introduction

- we have all those “nice” compositions
- some of them are completely abstract (refer to services)
- need to deploy
  - $\Rightarrow$ tell the system what you really want to run

Loading sequence

- most of the roby-related commands can take a “robot name” as argument
- this name defines what needs to be loaded
- for the time being, what you need to know is:
  - config/init.rb
  - config/robot_name.rb

oroGen workflow



- all deployed tasks for a single system are defined in a single oroGen deployment
  - ⇒ provides a good overview of all the available tasks
- the other oroGen projects can have test deployments

This is a recommended workflow, which works well with the supervision. You're free to do otherwise, though !

oroGen & Roby loading sequence

1. **Roby** loads the base config files

config/init.rb
config/asguardv3.rb

```
Roby.app.use_deployments_from "asguard_deployment"
```

2. **oroGen** loads the spec for the requested deployment

asguard_deployment.orogen

```
using_task_library 'canserial'
using_task_library 'canbus'
using_task_library 'xsens_imu'
deployment 'canserial'do
  task('canserial','canserial::Task').
    realtime.
    priority(75)
  add_default_logger
end
deployment "xsens_imu"do
  task('xsens_imu','xsens_imu::Task').
    realtime.
```

3. **oroGen** loads depended-upon specifications

xsens_imu.orogen

```
name 'xsens_imu'
task_context 'Task'do
```

4. **Roby** creates the corresponding task model classes (convertion from snake_case to CamelCase !)

XsensImu::Task

5. **Roby** loads the extension file in tasks/orogen/, if there is one

tasks/orogen/xsens_imu.rb

```
class XsensImu::Task
  provides Orientation
end
```

A note of Ruby

When, in `tasks/orogen/xsens_imu.rb`, we do

```
class XsensImu::Task
    provides Srv::Orientation
end
```

we reopen the `XsensImu::Task` class

$\Rightarrow$ we add “stuff” to an already existing class (a Ruby feature)

The role of devices

- central role: that's where data comes from (and goes to) !
  - ⇒ they are part of the *robot description*

What we are going to do is . . .

- learn how the device descriptions work
- learn how to list our robot's devices

Devices

Tasks are declared as drivers

tasks/orogen/hokuyo.rb

```
class Hokuyo::Task
  driver_for "Devices::Hokuyo"
end
```

tasks/orogen/xsens_imu.rb

```
class XsensImu::Task
  driver_for "Devices::XsensImu" do
    provides Srv::Orientation
  end
end
```

Robot description block
(in config/robot_name.rb)

```
Robot.devices do
  device(Devices::Hokuyo).
    additional_configuration.
    more_configuration.
    ....

  device(Devices::XsensImu).
    additional_configuration.
    more_configuration.
    ....
end
```

Devices

- device models are defined in Devices (Dev in short)
- devices are data sources
- one task can be a driver for multiple devices simultaneously
- the driver_for 'Dev::ModelName' form defines both the device model and says that the task is a driver for this. The "Dev::" (or Devices::) prefix can be omitted.
- if you have multiple available drivers for a given device, define the device model separately with
  device_type 'ModelName'
  and declare the driver without the quotes with
  driver_for Dev::ModelName

Small note

I'm cheating !

The FourWheelPlatform used before does not exist anymore.
There are only Actuators from now one

Ask the system to deploy !

More layout !

config/deployments/ predefined deployment files

edit config/deployments/odometry.rb

add_mission Compositions::Odometry

What do we have . . . for now ?

`tasks/orogen/xsens_imu.rb`

```
class XsensImu::Task
    driver_for 'XsensImu' do
        provides Orientation
    end
end
```

`tasks/orogen/hokuyo.rb`

```
class Hokuyo::Task
    driver_for 'Hokuyo'
end
```

`config/tutorial.rb`

```
Robot.devices do
    device XsensImu
    device Hokuyo
end
```

`config/deployments/odometry.rb`

```
add_mission Compositions::Odometry
```

run

```
scripts/orocos/instantiate odometry -o svg
```

and get

```
|= cannot find a concrete implementation for 1 task(s)
| for DataServices::Actuators:0x7f6c54473f48[]
|   child actuators of Compositions::Odometry/[odometry.is_a?(AsguardOdometry::
```

run

```
scripts/orocos/instantiate odometry -o svg
```

and get

```
|= cannot find a concrete implementation for 1 task(s)
| for DataServices::Actuators:0x7f6c54473f48[]
|   child actuators of Compositions::Odometry/[odometry.is_a?(AsguardOdometry::
```

Ahh . . . yes . . .

Our device list contains

- a single Srv::Odometry provider (Odometry::Task)
- a single Srv::Orientation provider (the IMU)
- but no Srv::Actuators provider
  - $\Rightarrow$ remember that issue with the hbridge task ?

The hbridge deployment

- the hbridge multiplexes/demultiplexes
- it creates the “right” ports at configuration time
- but we need to tell the module what to do !

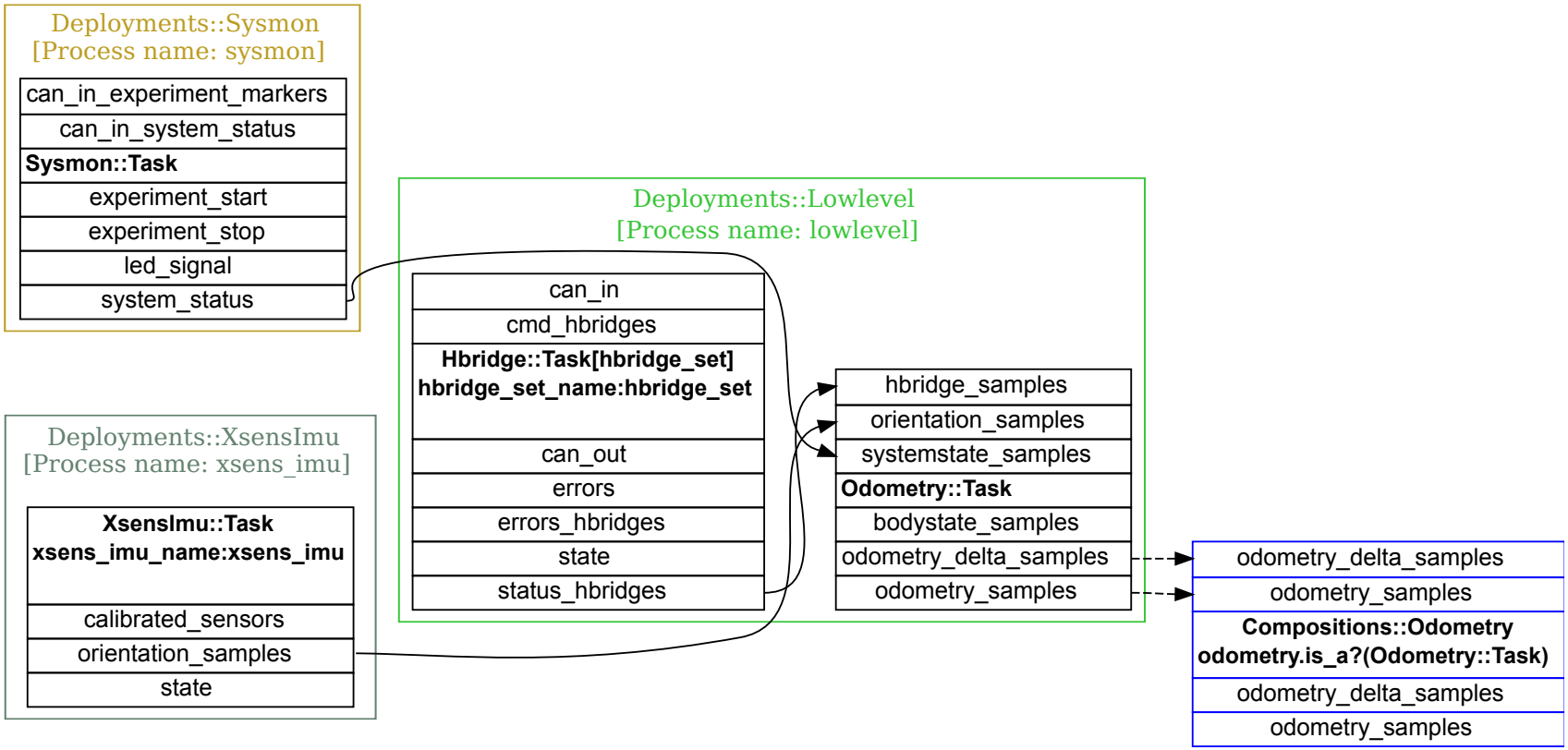
```
Robot.devices do
    hbridges = device(Dev::HbridgeSet)
    hbridges.slave(Dev::Hbridges).
        select_ids(−1, 2, 3, −4)
end
```

Defining dynamic services

```
class Hbridge::Task
  hbridge_set = driver_for('HbridgeSet')
  hbridges = hbridge_set.dynamic_slaves 'Hbridges' do
    output_port "errors_#{name}", "/hbridge/Error"
    input_port "cmd_#{name}", "/base/actuators/Command"
    output_port "status_#{name}", "/base/actuators/Status"
    provides Srv::Actuators,
      "status" => "status_#{name}",
      "command" => "cmd_#{name}"
  end
```

- one can define Hbridges devices on HbridgeSet devices
- Hbridges devices are Actuators

Finally . . .



Colored boxes that enclose tasks represent deployments (i.e. processes), the blue boxes represent the composition's interfaces (exported ports)

What about multiple devices ?

```
Robot.devices do
    device Dev::XsensImu, :as => 'xsens_imu1'
    device Dev::XsensImu, :as => 'xsens_imu2'
    device Dev::Hokuyo
end
add_mission Cmp::Odometry
```

```
= cannot find a device to tie to 1 task(s)
| for XsensImu::Task:0x7f558e8cd710[]
|   child imu of Compositions::Odometry/[odometry.is_a?(Odometry::Task)]:0x7f55
```

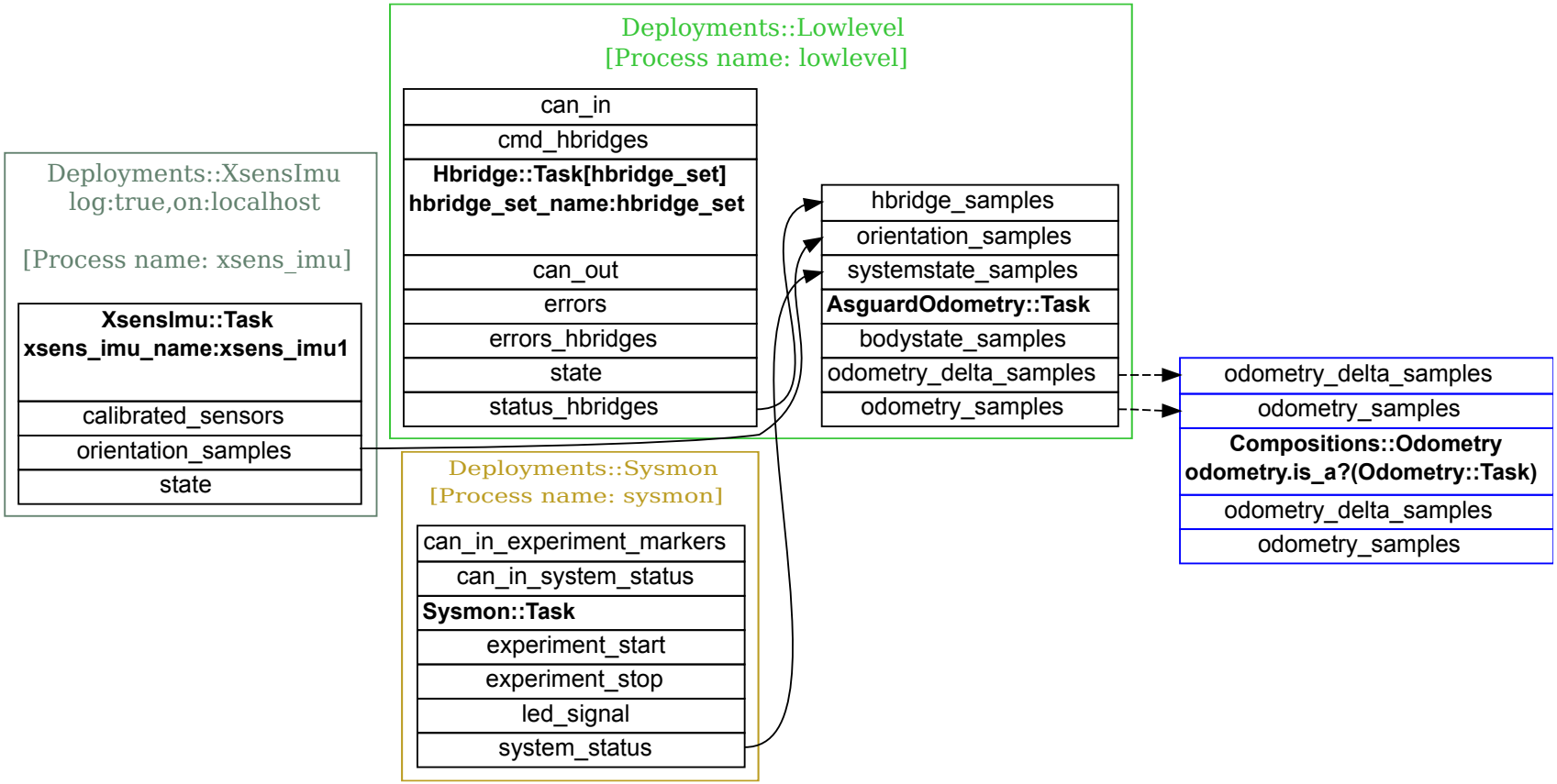
Disambiguate by name

```
Robot.devices do
    device XsensImu, :as => 'xsens_imu1'
    device XsensImu, :as => 'xsens_imu2'
    device Hokuyo
end
add_mission(Compositions::Odometry).
    use 'xsens_imu1'
```

Important

The device name has to match the deployed task name

Disambiguate by name



So . . .

- the engine will not leave any ambiguity
- if it is ambiguous, it will generate an error
- if there's something missing, it will generate an error
- in all the other cases, it will deploy automatically

Multi-host deployments

- two-machine setup, avalon-front and avalon-back
- supervision runs on avalon-back (localhost)
- tell the supervision what should run where

```
Roby.app.use_deployments_from 'avalon_back', :on => 'localhost'
Roby.app.orocos_process_server 'front', 'avalon-front'
Roby.app.use_deployments_from 'avalon_front', :on => 'front'
```

Multi-host deployments

- the system will deploy by picking tasks on each available machine
- it will try to reduce network load (i.e. pick tasks that are close to each other)
- the orocos process server must be manually started on avalon-front with

  `orocos_process_server`
- the `avalon_front` oroGen project does not need to be built on `avalon_back`, but needs to be on `avalon_front`

Up to you now !

- create deployment files to play around
- run
  `scripts/orocos/instantiate -r robot_name -o svg --no-policies --no-loggers deployment_name`
  - ⇒ deploys the specification from config/deployments/deployment_name.rb
- alternatively, to get the network before deployment,
  `scripts/orocos/instantiate -r robot_name -o svg --no-policies --no-loggers --no-deployments deployment_name`
  - ⇒ same, but does not apply the generated network to deployed tasks

5. Dataflow Configuration

The principle

- connection policies are not portable across systems
  - $\Rightarrow$ depends on device rates, ...

$\Rightarrow$ compute them !

Basic requirement

Basics: input ports can be declared as . . .

needs_data_connection this is the default. You'll only get the last sample written

needs_reliable_connection if turned on, the connections will be set up so that all samples should reach the task

```
input_port("can_in", "canbus/Message").
    needs_reliable_connection.
    doc("the HBridge-related messages coming from the CAN bus").
```

Needed information

- a minimum device period

```
device(XsensImu, :as => 'xsens_imu1').
    period(0.01)
```

- a burst size, and burst rate (if there is one)
- worstcase trigger latency
- maximum processing times

Needed information

- a minimum device period
- a burst size, and burst rate (if there is one)

```
output_port("can_out", "canbus/Message").
    burst(40, 0).
    doc("the HBridge-related messages coming to be written on th
```

- worstcase trigger latency
- maximum processing times

Needed information

- a minimum device period
- a burst size, and burst rate (if there is one)
- worstcase trigger latency
  - ⇒ default is 5ms for realtime tasks, 25ms for non-realtime tasks
- maximum processing times

Needed information

- a minimum device period
- a burst size, and burst rate (if there is one)
- worstcase trigger latency
- maximum processing times
  - ⇒ by default, 0. Can be set with worstcase_processing_time

```
class Icp::Task
    worstcase_processing_time 1
end
```

Other

Given all the information listed until now, you'll get the worstcase buffer size
To reduce it a bit, you can specify when output ports are written

⇒ see trigger methods on Orocos::Spec::OutputPort in orogen

Note

- if the policy is manually provided in connect (in composition spec), then automatic policy configuration is completely bypassed

```
composition 'Odometry' do
  ...
  connect imu.orientation_samples => odometry.orientation_samples,
    :type => :buffer, :size => 20
end
```

- this information is used to configure logging too !!!
  - ⇒ if not available, logging falls back to a buffer size of Orocos::RobyPlugin::Engine.default_logging_buffer_size

Logging configuration

- by default, all ports are logged
- can be turned completely off in config/robot_name.rb with

  State.orocos.disable_logging

- on a per-type basis with

  State.orocos.exclude_from_log `"/canbus/Message"`

- for all tasks of a certain type with

  State.orocos.exclude_from_log XsensImu::Task

- look at the documentation of Orocos::RobyPlugin::Configuration in orocos.rb

6. Runtime

Configuring tasks

- exclusively done through a configure method on the task model

```
class TrajectoryFollower::Task
  def configure
    super
    # Write properties
    orogen_task.controllerType = 0
  end
end
```

- less than ideal. Configuration files coming soon !

Configuring device drivers

A bit better

- one generic configuration method called device_id

```
device(Hokuyo).
  period(0.025).
  device_id("/dev/ttyS1")
```

- device definition can be retrieved in the configure method

```
class Hokuyo::Task
  def configure
    super
    device = robot_device
    orogen_task.device = device.device_id
  end
end
```

Configuring device drivers

can be extended on a per device-type basis

```
class Hokuyo::Task
  device_t = driver_for "Hokuyo"
  device_t.extend_device_configuration do
    def enable_remission_values; @enable_remission_values = true end
    def remission_values?; @enable_remission_values end
  end
end
```

in config/asguardv3.rb

```
device(Hokuyo).
  period(0.025).
  device_id("/dev/ttyS1").
  enable_remission_values
```

in tasks/orogen/hokuyo.rb

```
class Hokuyo::Task
  def configure
    super
    device = robot_device
    if device.remission_values?
      # enable on orogen_task
    end
```

Running your deployments

```
scripts/orocos/run -r robot_name deployment_name
```

But also

```
scripts/orocos/run -r robot_name deployment_name
device_name
```

And

```
scripts/orocos/run -r robot_name - device_name
```

Predefining deployments

use define(name, model) instead of add(model)

```
define('trajectory_following', ControlLoop).
  use TrajectoryFollower::Task, Skid4Control::SimpleController
define("drive_simple", ControlLoop).
  use Controldev::Joystick, Skid4Control::SimpleController
define("debug_piv", ControlLoop).
  use Controldev::Sliderbox, Control::PIVController
```

$\Rightarrow$ can be used as a string in “add”

```
add_mission('debug_piv')
```

Defining modalities

- a modality is a way to do something
- the supervision's goal is to allow to switch between modalities online
- if you select a modality, other running modalities of the same category are stopped
- only things defined with `define` can be used

```
model.data_service_type "NavigationMode"
Compositions::ControlLoop.provides NavigationMode
Compositions::CorridorServoing.provides NavigationMode
modality_selection NavigationMode, "trajectory_following",
    "corridor_servoing", "drive_simple", "drive_piv"
```

Selecting modalities online - programmatically

```
navigation_mode = nil
Roby.every(0.1, :on_error => :disable) do
  if State.lowlevel_state?
    if State.lowlevel_state != 3
      if navigation_mode
        navigation_mode.stop!
        navigation_mode = nil
      end
    elsif State.lowlevel_state == 3
      if !State.navigation_mode?
        Robot.warn "switched to mode 3, but no navigation mode is selecte
      elsif !navigation_mode
        Robot.info "starting navigation mode #{State.navigation_mode}"
        navigation_mode, _ = Robot.send("#{State.navigation_mode}!")
        navigation_mode = navigation_mode.as_service
      end
    end
  end
```

Selecting modalities online - shell

```
scripts/orocos/shell [--host hostname]
> trajectory_following!
```

(`actions` lists all the commands that are available)

Getting a predefined startup

`scripts/run robot_name [robot_type]`

- loads
- starts the Roby engine
- loads controllers/robot_name.rb
  - ⇒ this is where you put your main system's configuration

Asguard

controllers/asguardv3.rb contains the modality switching code between mode 2 (teleoperation) and mode 3 (autonomous)

Communication busses

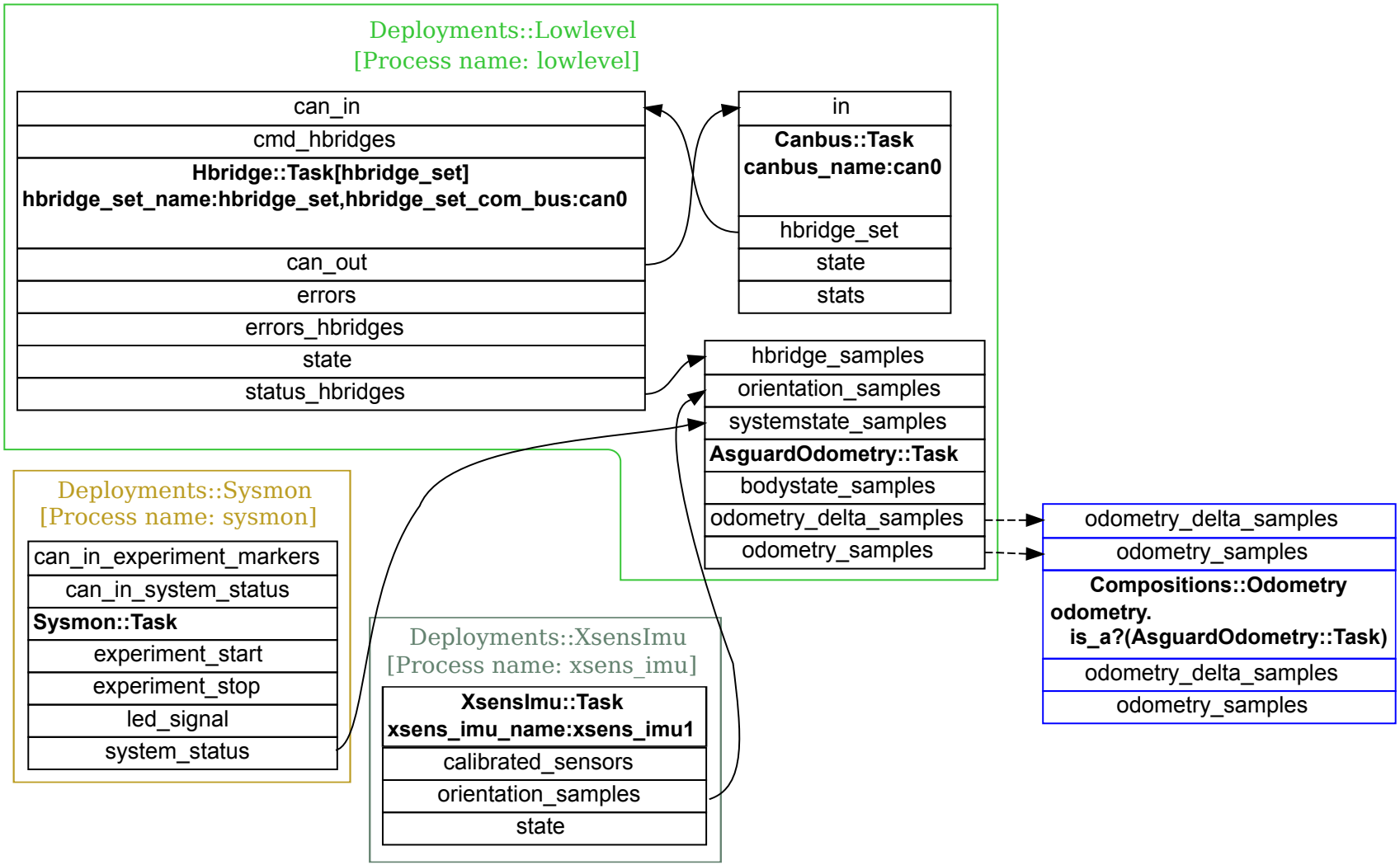
- they are dispatching data across other components
- you don't need to explicitely add them to your compositions
- only need to declare them

```
com_bus(Canbus, :as => 'can0').
  device_id '/dev/can0'

through 'can0' do
  hbridges = device(HbridgeSet)
  hbridges.slave(Hbridges).
      select_ids(-1, 2, 3, -4)
end
```

- Combus is a special device type defined by tasks/orogen/canbus.rb
- Canbus::Task configures itself based on who's connected to it
- at instanciation time, the supervision generates the necessary ports and connections

Communication busses



7. Conclusion

What did we not cover

- execution display and logs
- error representation and error recovery
- mission plans
- switching configuration (must be done at the composition level for now)
- the instanciation GUI (broken for now)

What you should be able to do

- get your system(s) running
- get to grips with modality switching
- get a lot of error reports
- get bugs fixed by me . . . :P