# Rock - from Components to Systems

Sylvain Joyeux

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

1. The Robot Construction Kit
2. A note about embedded DSLs
3. From Components to Systems
4. Conclusion

# The Robot Construction Kit

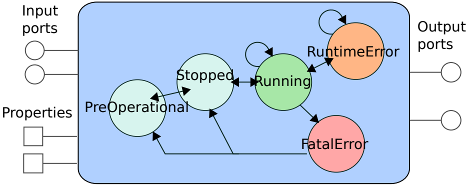
# Rock

Algorithms in Rock are developped in a framework-independent way

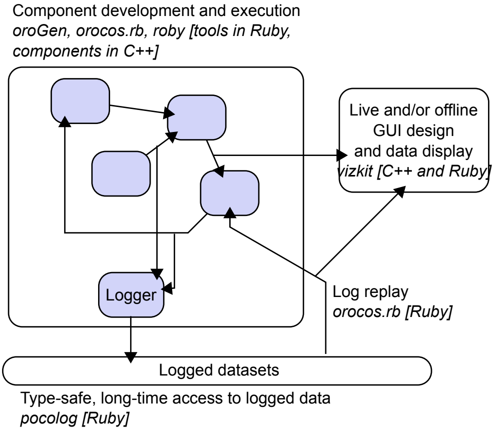
| Implementing | | Running | | Data Display and Analysis | |
|---|---|---|---|---|---|
| autoproj | *Build System* | orocos.rb | *Startup and Monitoring of components using Ruby scripts* | data_logger | *High-performance data logging component* |
| RTT | *C++ Component Implementation* | Roby | *Model-based deployment and system supervision* | orocos.rb | *Ruby library for replaying log data to components* |
| oroGen | *Fast Component Development* | | | pocolog | *Command-line tool and Ruby library to manipulate data log files* |
| typeGen | *Standalone Typekit Generation* | | | vizkit | *Robot control and Data Display UI, including offline* |

# RTT - Highlights

- hard-realtime compatible
- RTT core is “only” a component model and supporting infrastructure
  - ⇒ independent of any *communication layer*
- “main” communication layer is CORBA. Near-full support for data flow on ROS, YARP and POSIX message queues
- can talk to multiple communication layers at the same time



# Rock and RTT - Development chain




- main component functionality is C++
- "glueing" done in a dynamic language (ruby)
- Ruby is used from the low-level infrastructure to the models !

⇒ use embedded DSLs !!!

# System Management

## Design Principles

- use the right tool for the job
- coordination concerns must be left out of the components

⇒ how to easily define configurations ?

⇒ how to switch between these configurations ?

# A note about embedded DSLs

# A note about “embedded DSLs”

- reuse an existing programming language to provide a domain-specific language

⇒ DSL files are actual programs from the host language

## Turning an imperative syntax ...

```
project = Project.new
task = project.task_context 'BaseTask'
p = task.output_port('solution', '/gps/Solution')
p.doc="the GPS solution as reported by the hardware"
p = task.output_port('position_samples', '/base/samples/RigidBodyState')
p.doc="computed position in m"
task.error_states :IO_ERROR, :IO_TIMEOUT
p = task.property("utm_zone", "int", 32)
p.doc="UTM zone for conversion of WGS84 to UTM"
```

# A note about “embedded DSLs”

- reuse an existing programming language to provide a domain-specific language

⇒ DSL files are actual programs from the host language

## . . . into a declarative one

```
task_context 'BaseTask' do
  output_port('solution', '/gps/Solution').
    doc "the GPS solution as reported by the hardware"
  output_port('position_samples', '/base/samples/RigidBodyState').
    doc "computed position in m"
  error_states :IO_ERROR, :IO_TIMEOUT
  property("utm_zone", "int", 32).
    doc "UTM zone for conversion of WGS84 to UTM"
end
```

# eDSLs: advantages / disadvantages

**Advantages**

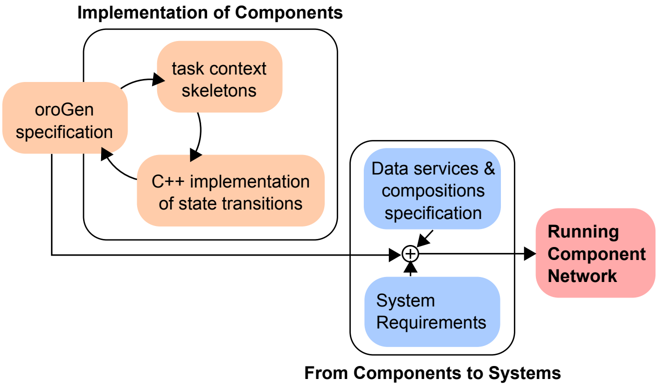
- no need for a specialized parser
- easy to extend for the tool developer and the tool “user”
- allows to mix model and programs
  - ⇒ results in highly reflexive systems

**Disadvantages**

- can only go as far as the host language’s syntax stretches
  - ⇒ result does not look as streamlined as with a custom parser

# From Components to Systems

# System Integration Process



# System Integration Process

- the deployment engine is integrated into a plan management framework
- straightforward integration into plan-based reasoning
- the functional layer is part of the system's situation

# Plan Management

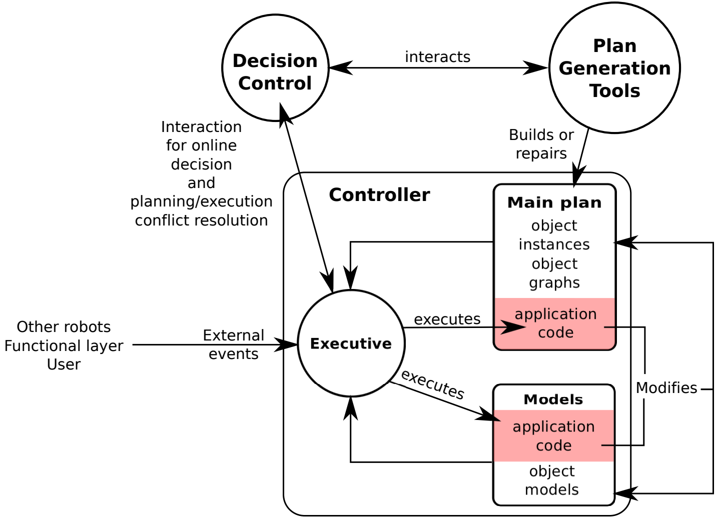
*“There’s more to life than making plans”*

- plan libraries
- plan transformation
- execution monitoring
- high-level goal management
- …

# Plan Management

- usually done by having multiple tools talk to each other
  - ⇒ segregates information, missing the “big picture”
- sometimes mitigated using one common planning model (IDEA, T-REX)
  - ⇒ constrained to what can be represented in these models
- even sometimes targetting an “only planning” system
  - implementations usually assumes that the functional layer “mostly works fine”
  - problem with stability when revising plans

# Architecture



# Plan Management

## Roby

A software framework which . . .

- has a model designed to represent the system's situation and its evolution
- can integrate planning-generated plans there
  - ⇒ but also other plan generators
- uses the fact that, during execution, "expensive" algorithms are fine
  - ⇒ you don't have to run them often

# Reuses a software engineering approach

Need to manipulate group of components as components themselves

$\Rightarrow$ compositions

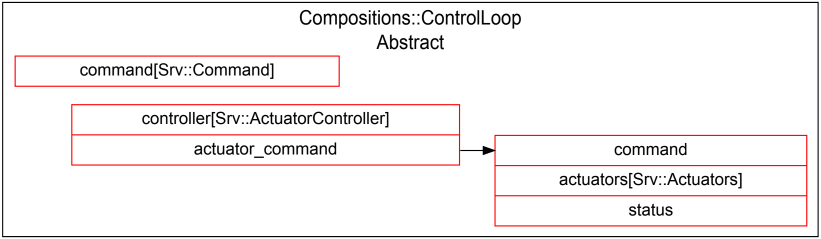
**Contributions**

- does it *with sharing*
- little system requirements are needed to deploy a full system
- adds dynamic reconfiguration
- adds flexible means of adapting the specification to the component's needs to promote reusability

# Compositions

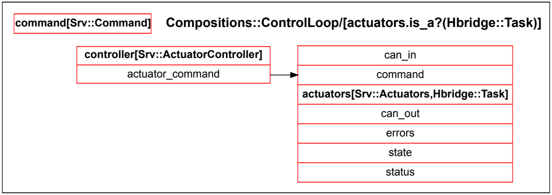
Represent a *functional* block. Provide a context

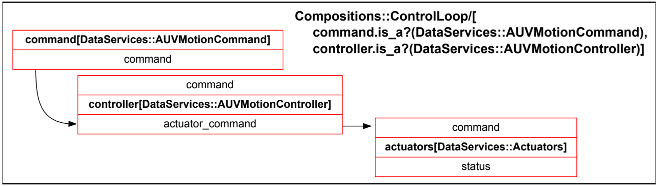
- for connections
- for dependencies/constraints between tasks



# Specialization

Mean of adapting the base composition model to specific components / services: new constraints, new connections



# System Deployment

## Assumptions

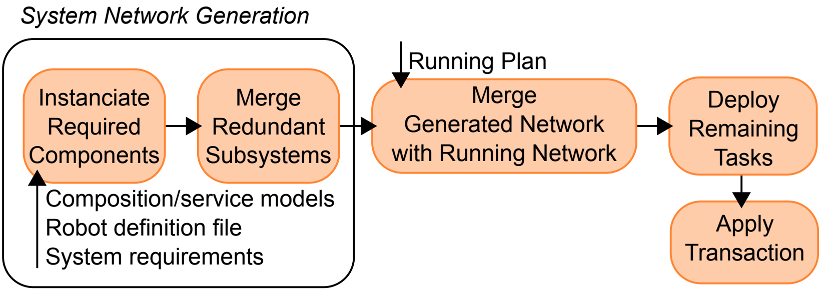
A component's behaviour only depends on

- its configuration
- its inputs

## Key idea

- inject dependencies and constraints *locally*
- …but remove redundancies afterwards (and check validity of the resulting network)
- devices can't be instantiated more than once (obviously)
- deployment can be verified offline

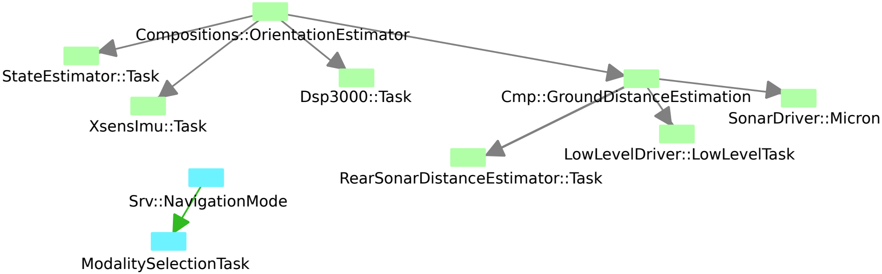
# System Deployment



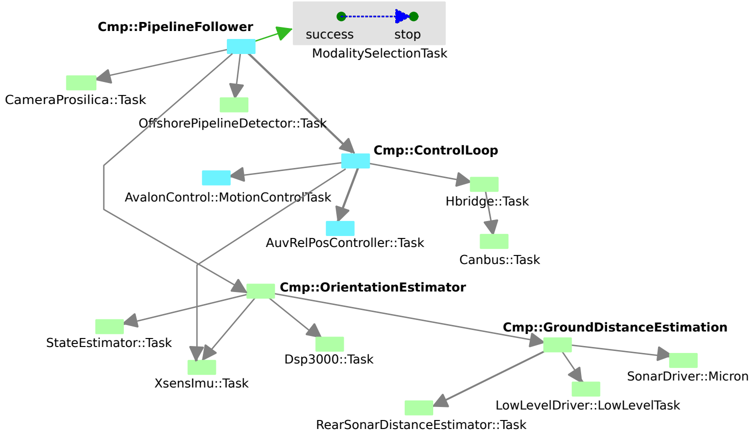
# Dynamic Management

- changing system: representation of requirements in the plan itself (as tasks)
- reconfiguration: components are scheduled for reconfiguration in step 3 if needed.
- try to minimize component restart cycles (do not restart unless really needed)

# Modality Selection



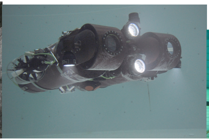
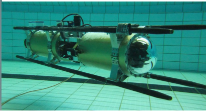
# Modality Selection



# Conclusion

# Conclusion



The approach we presented runs on 4 very different systems:

- an autonomous navigation platform
- a reconfigurable rover system
- two very different AUVs (with different missions / sensor sets / requirements)
- ran at Sauc-E 2010, will run on Sauc-E 2011

# Summary

**Presented Rock**

A complete, flexible robot development chain that offers framework-independent algorithms, integrated components, post-mortem analysis tools and advanced system deployment tools.

**Presented Rock's system deployment approach**

- promotes reusability of models and monitoring code
- integrated into plan management: can execute, monitor and adapt running systems

Thank you for your attention

`http://rock-robotics.org`
Prerelease: end of June 2011
Release: September 2011