Towards Robot-independent Manipulation Behavior Description

Malte Wirkus
DFKI-RIC Bremen, Germany
Malte.Wirkus@dfki.de

Outline

- Introduction
- Robot Manipulation Behavior Generation
- Control system Specification
- Discussion

Introduction

- Robotic software frameworks
  - Define common component interface
    - → Increase resuability of software
  - Tools for software development
    - → Increase in developer's productivity

→ Access to large pool of software components

→ Robot programming: Increasingly software integration and configuration task

[http://www.ros.org]

[http://wiki.icub.org/yarpdoc/]

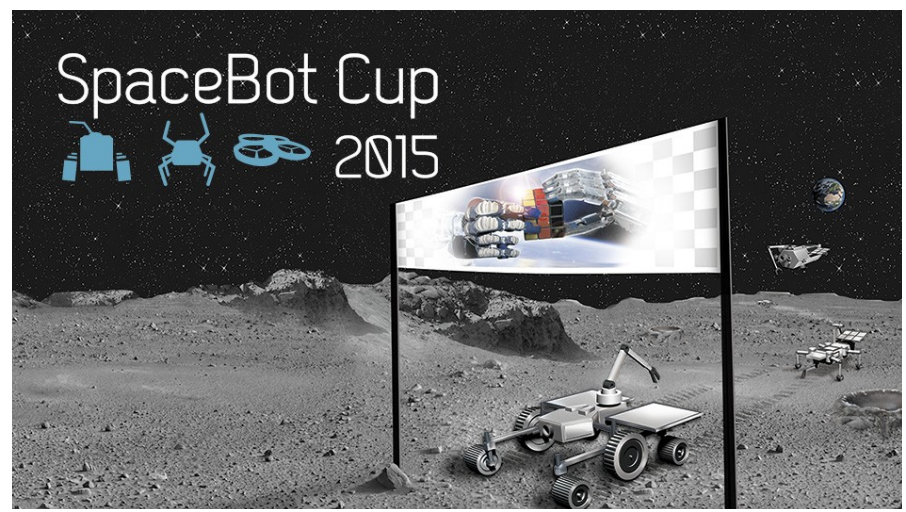
[http://rock-robotics.org]

- Expectations on robots increase
  - More complex tasks
  - Complex missions
    - Different modes of operation / behaviors

“Robot programming increasingly becomes a software integration and configuration task”

→ .. but it's still complex

[DARPA Robotics Grand Challenge]



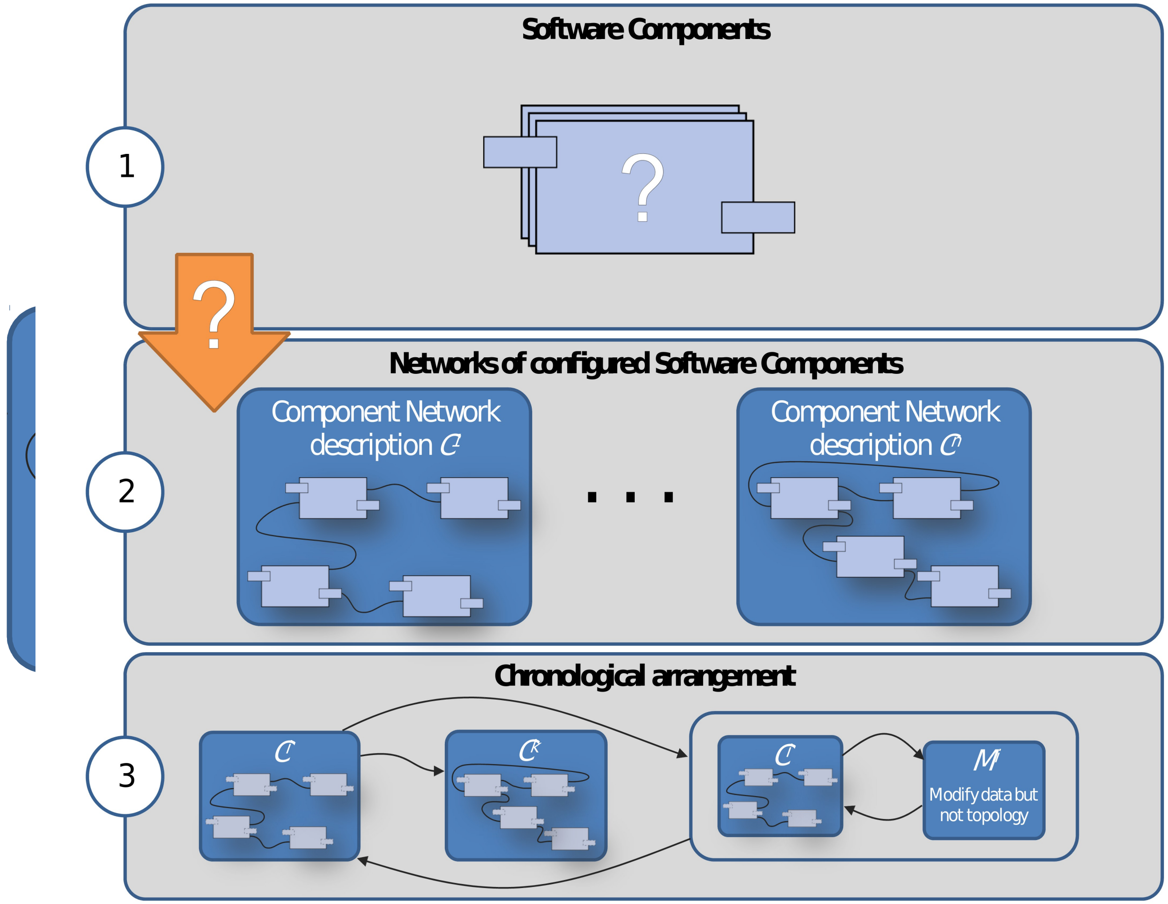
[DLR SpaceBot Cup]

Motivation and Goal

- Support creation of complex robot behavior
- Allow transfer between robots of different morphology

[all Images: http://robotik.dfki-bremen.de]

Robot Manipulation Behavior Generation



Contribution

Workflow for robot manipulation behavior design

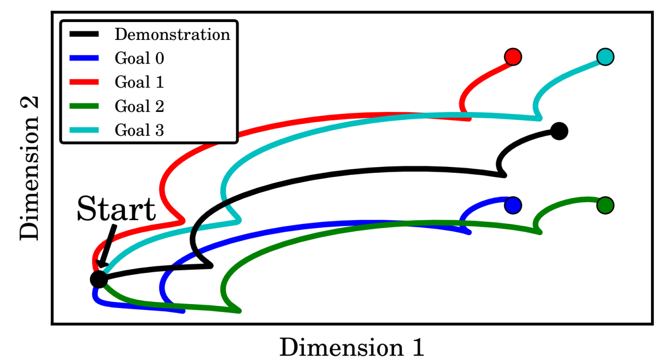
- easy to work with
- supports transfer of behavior

- Utilization of specific algorithms
- Data processing for control
- eDSL to support development (next section)

Utilization of specific algorithms

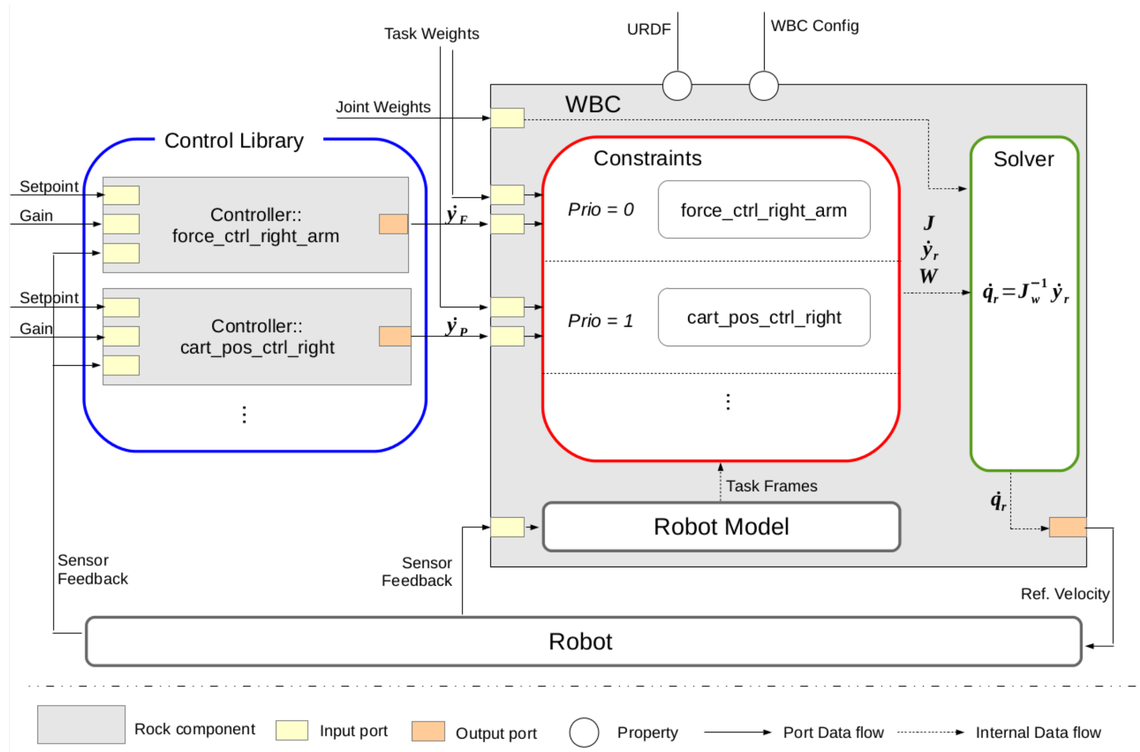
Use of parametric motion description

- Represent different motion by exchanging motion parameters
- Adaptive to current situation
- Tools for creating motion parameters
  - e.g. Imitation Learning



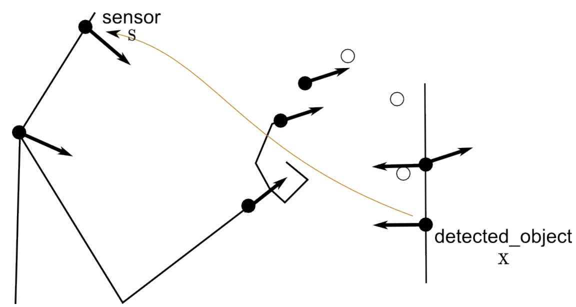
Use whole-body control algorithm

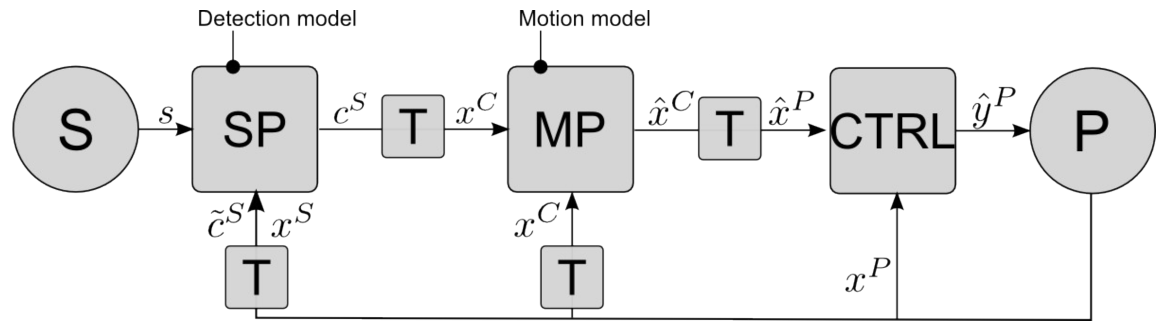
- Impose constraints on parts of the robot
- Allow parallel execution of controllers using same joints



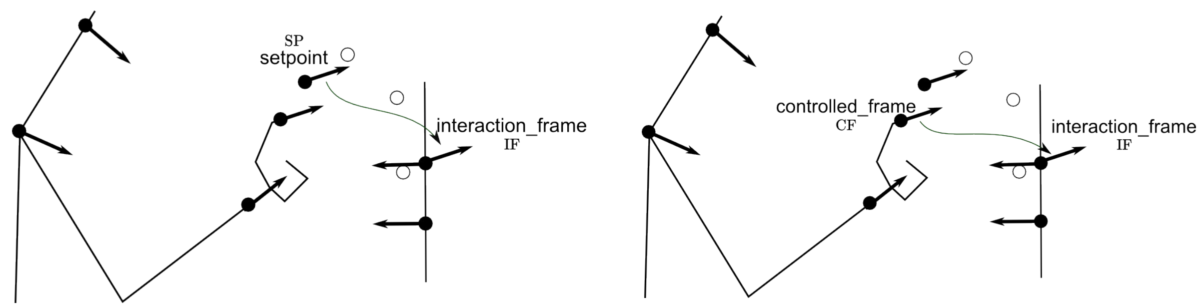
<u>**Data processing for Cartesian control**</u>

Perception



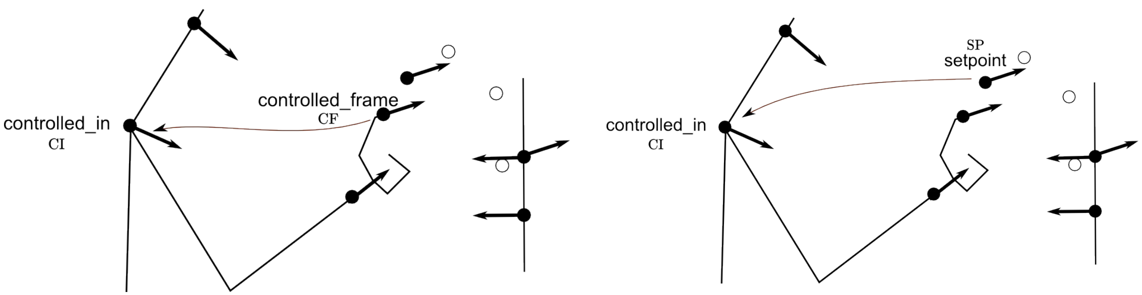


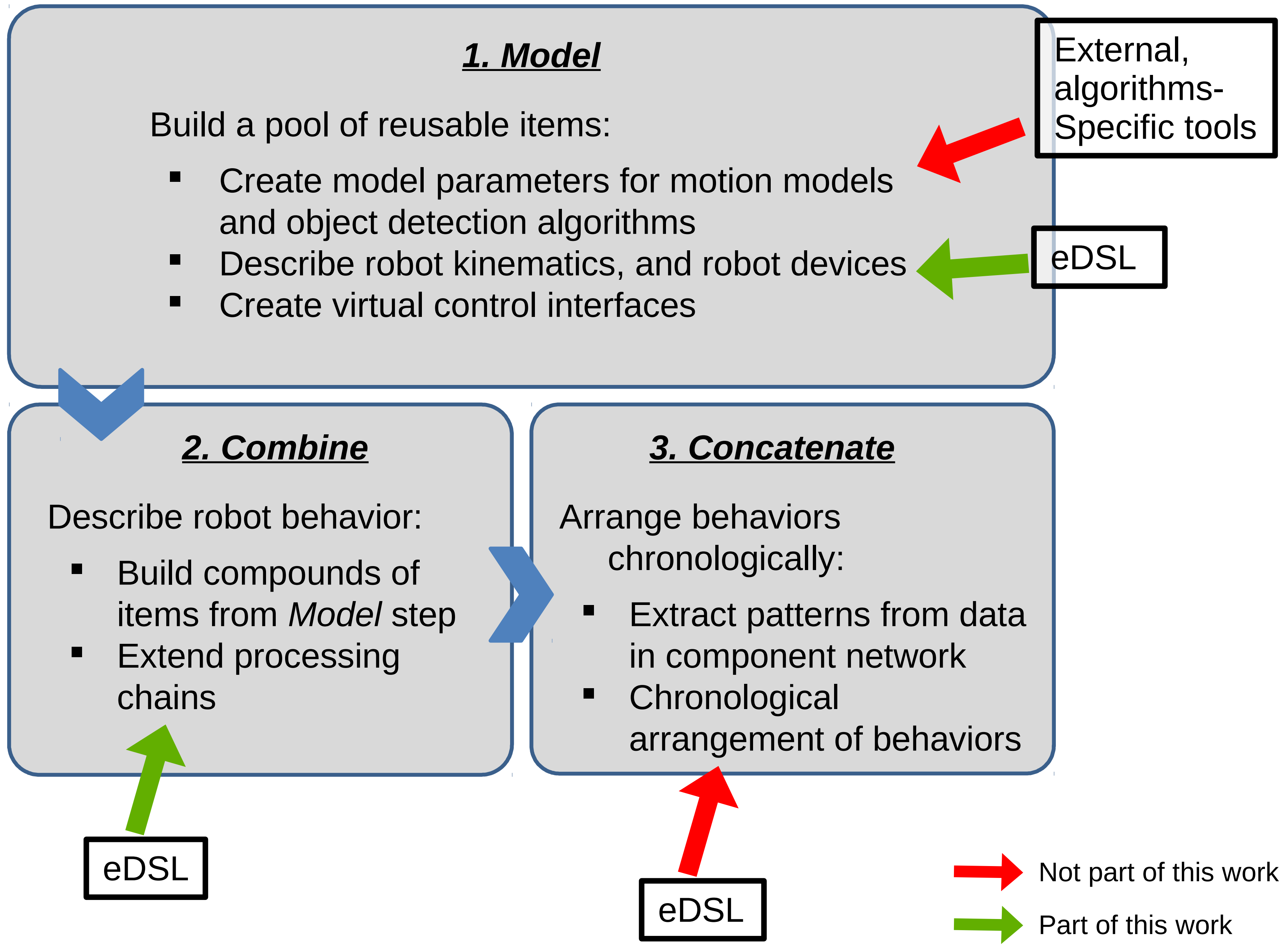
Trajectory Generation



- Decouples robot morphology from task motion and sensor processing
- Motion description can be applied to different context

Control



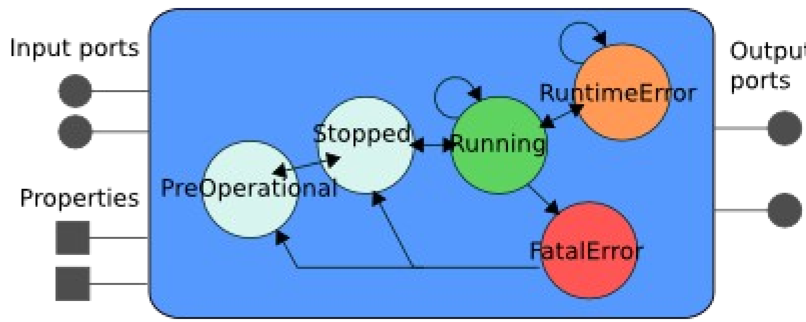

1. Model
Build a pool of reusable items:
Create model parameters for motion models and object detection algorithms
Describe robot kinematics, and robot devices
Create virtual control interfaces
External, algorithms-Specific tools
eDSL
2. Combine
Describe robot behavior:
Build compounds of items from Model step
Extend processing chains
eDSL
3. Concatenate
Arrange behaviors chronologically:
Extract patterns from data in component network
Chronological arrangement of behaviors
eDSL
Not part of this work
Part of this work

Control Network Specification

Rock
the Robot Construction Kit

[http://rock-robotics.org]

- **Component model**
  - Orocos RTT
  - Configuration interface
  - Data flow interface
  - Life-cycle
  - Single-purpose



[http://rock-robotics.org]

- **System modeling**
  - Data Service: Semantic labels → abstract data flow interface
  - Compositions: Functional subnetworks of actual components, Data Services, already modeled subnetworks
  - Instantiation requirements: Selection of actual components for Data Services. Choosing of configurations for component.
- **Plan management**
  - Represent and execute plans
  - Component network models can be used as tasks
    - Component network instantiation
    - Supervision

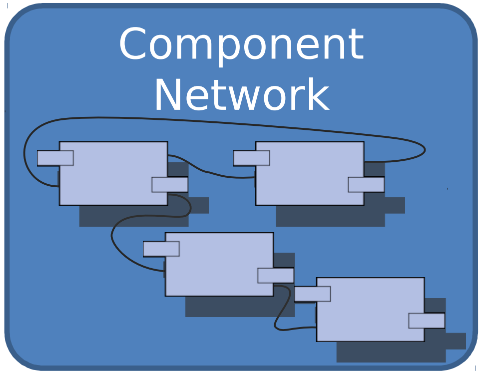
User code

```
edsl_context do
    #block of ruby code
    #and context-specific
    #commands
end
```

```
class MetaModel
   def context_command(arg)
      #configure MetaModel
   end
end
```

Inject information in Instance requirements of compositions

Create configurations for base components



Base Components

- Kinematics
- Split/merge data streams
- Multi-purpose controllers
- Transformer
- Whole-Body Control

[http://robotik.dfki-bremen.de]

Control system specification

- Aggregation of different hardware parts
  - Represented by their driver and
- Create multi-stage control network

Hardware Resources

Declare new device type

```
module Devices
      joints_device_type "MyJointsPositionDriver" do
            position_controlled
      end
      joints_device_type "MyJointsVelocityDriver" do
            velocity_controlled
      end
end

MyJointDriver::Task.driver_for
     Devices::MyJointsPositionDriver, :as =>
     'position_controlled'
MyJointDriver::Task.driver_for
     Devices::MyJointsVelocityDriver, :as =>
     'velocity_controlled'
```

Register Rock-Component that implements driver for the new hardware

Robot

```
robot do
      kinematic_description
           "/path/to/my/kinematic_description.urdf"
      device(Devices::JointsPositionDriver, :as =>
           'armr').joint_names('ar', 'br',
           'cr').with_conf('armr')
      device(Devices::JointsPositionDriver, :as =>
           'arml').joint_names('al', 'bl',
           'cl').with_conf('arml')
      device(Devices::JointsPositionDriver, :as =>
           'hr').joint_names('wr','gr').with_conf('hr')
      device(Devices::JointsPositionDriver, :as =>
           'hl').joint_names('wl','gl').with_conf('hl')
      device(Devices::JointsVelocityDriver, :as =>
           'head').joint_names('p', 't').
           with_conf('head')
end
```

Provide kinematic description.
Relate devices to robot's body by names.

Compose robot of device models

Load specific config for Driver.
Give additional information.

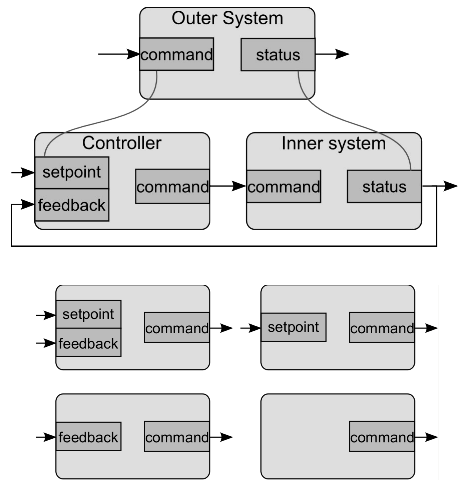
Control Networks

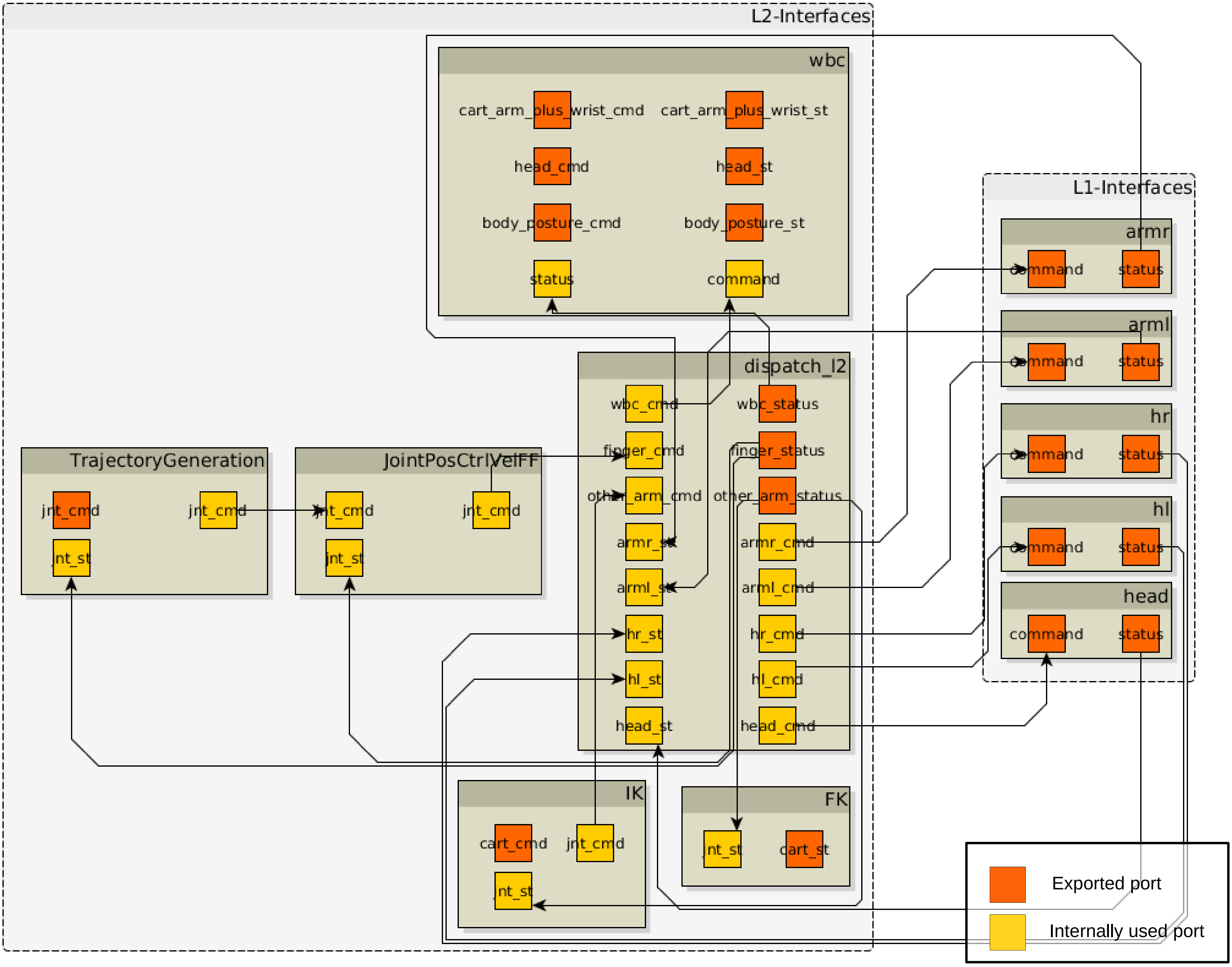
One stage in multi-stage control network

Different control interface types

```
control_collection "l2" do
      used_joints = [’ar’,’br’,’cr’,’wr’,’p’,’t’]
      wbc_interface used_joints, :as => "wbc",
           :initial_joint_weights => [1]*used_joints.size
           do
            cartesian_control_interface [’O’,’WR’],
                  :as => "cart_arm_plus_wrist",
                  :joint_names => [’ar’,’br’,’cr’,’wr’],
                  :priority => 1, :weights => [1,1,1,0.5]

            control_interface [’p’,’t’], :as => "head",
                 :priority => 2

            control_interface [’ar’,’br’,’cr’,’wr’],
                  :as => "body_posture", :priority => 3
      end

      control_interface [’gr’],
            :control_mode => :position,
            :as => ’finger’
      cartesian_control_interface ’O’, ’WL’,
            :joint_names => [’al’,’bl’,’cl’,’wl’],
            :control_mode => :velocity,
            :as => ’other_arm’
      end

      cascade_control finger_interface do
            push TrajectoryGeneration::Task
                        .with_conf(’arm_with_hand’)
      end
end
```

Conversion between control modes if needed

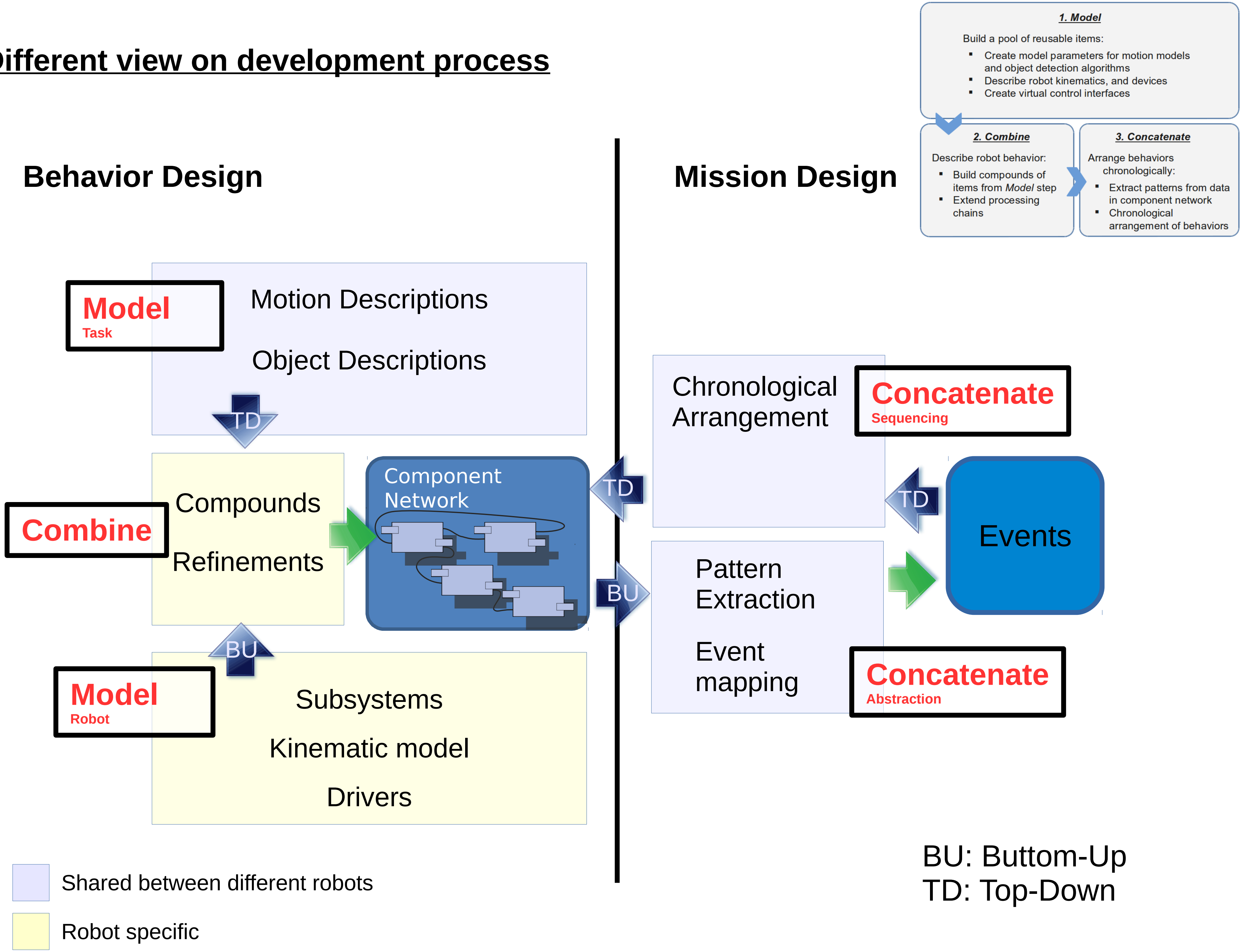




L2-Interfaces
wbc
cart_arm_plus_wrist_cmd
cart_arm_plus_wrist_st
head_cmd
head_st
body_posture_cmd
body_posture_st
status
command
dispatch_l2
wbc_cmd
wbc_status
finger_cmd
finger_status
other_arm_cmd
other_arm_status
armr_st
armr_cmd
arml_st
arml_cmd
hr_st
hr_cmd
hl_st
hl_cmd
head_st
head_cmd
TrajectoryGeneration
jnt_cmd
jnt_cmd
jnt_st
JointPosCtrlVelFF
jnt_cmd
jnt_cmd
jnt_st
IK
cart_cmd
jnt_cmd
jnt_st
FK
jnt_st
cart_st
L1-Interfaces
armr
command
status
arml
command
status
hr
command
status
hl
command
status
head
command
status
Exported port
Internally used port

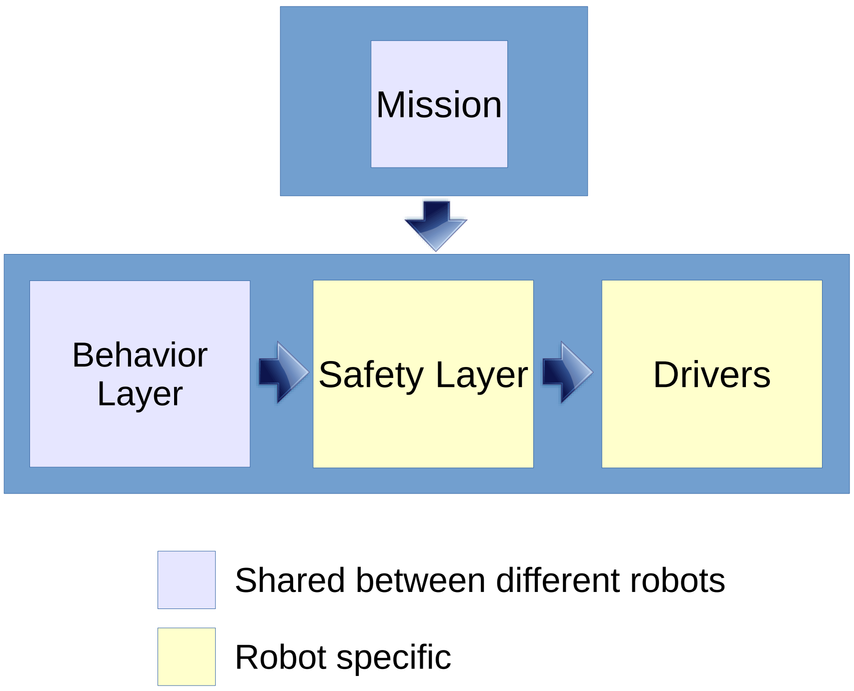
Discussion

Different view on development process

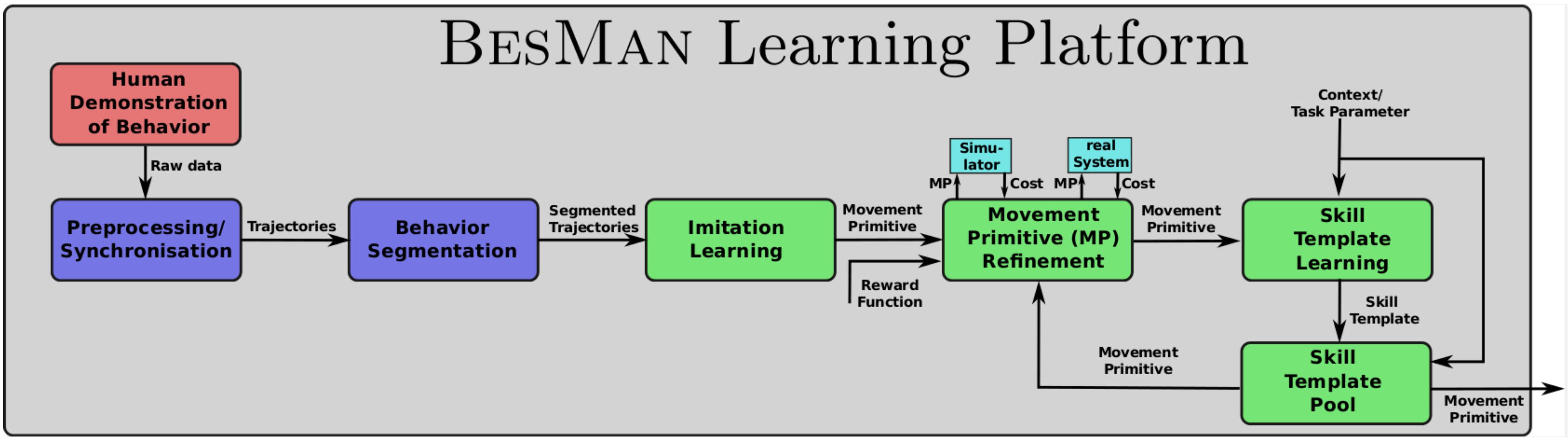


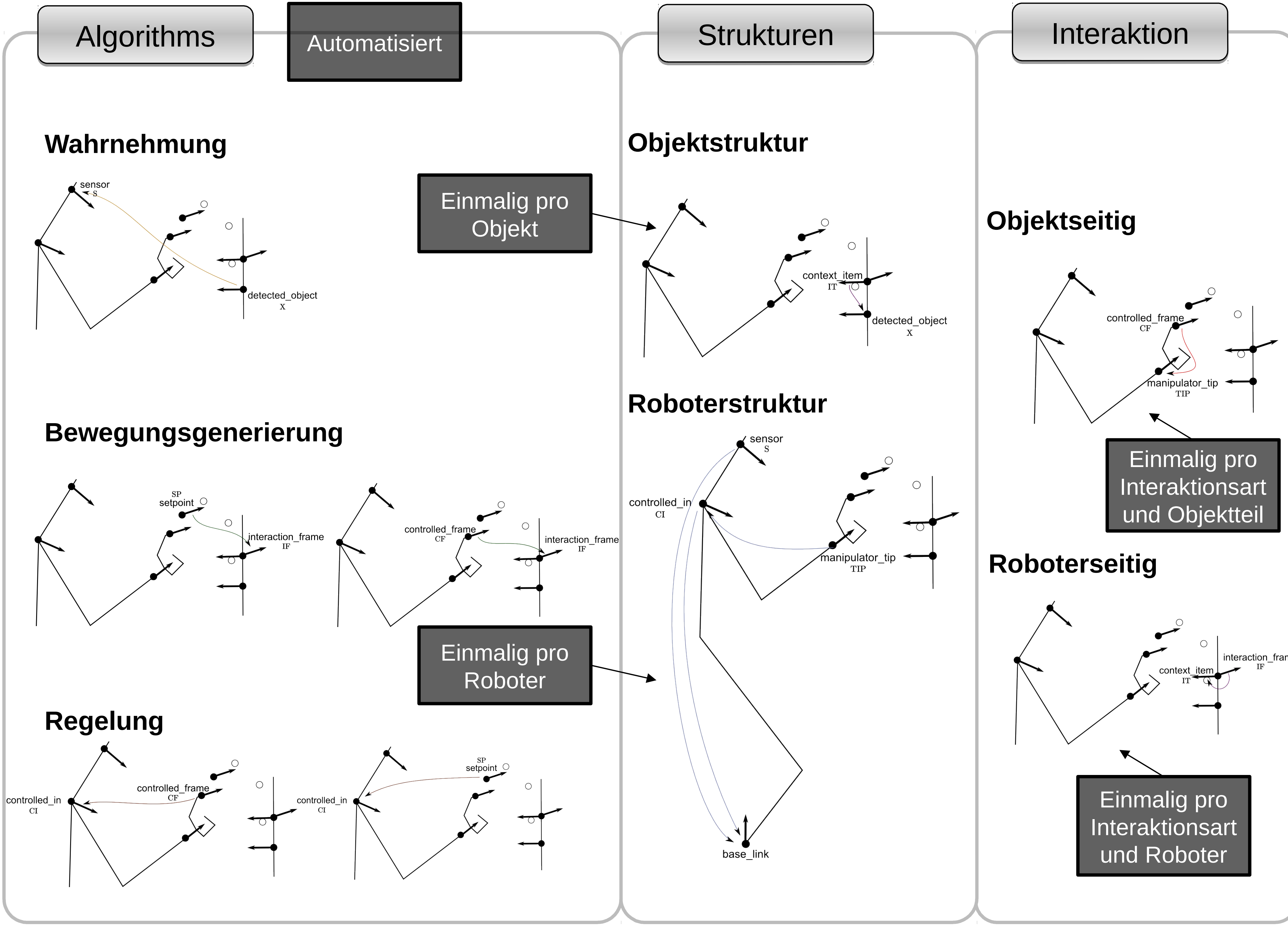
Next steps

- Further specify frame transformation handling
- Extend eDSL to support parametric motion description
- Support for pattern recognition in data of component network
- Evaluation

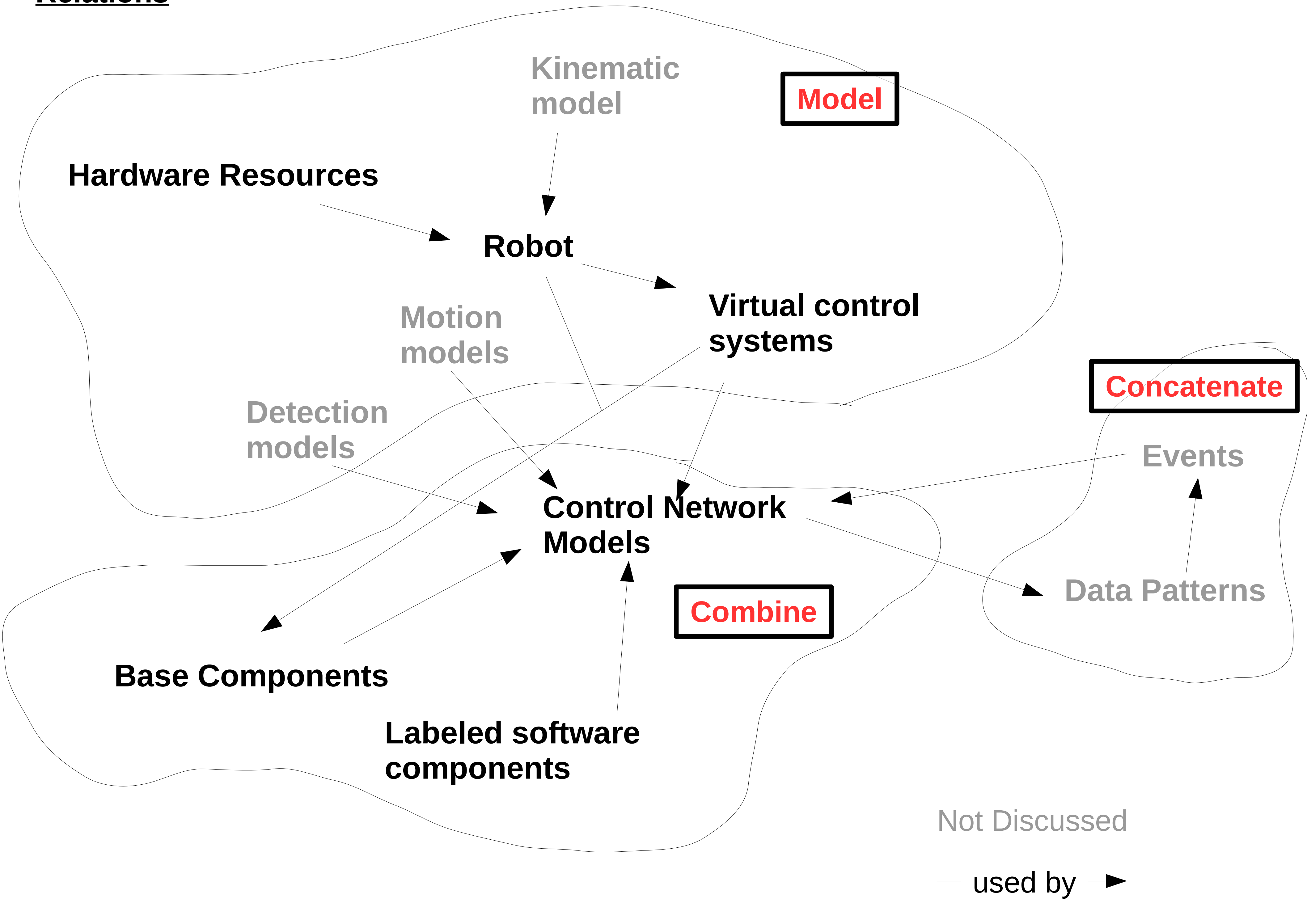


Thank you for your attention!


BesMan Learning Platform
Human Demonstration of Behavior
Raw data
Preprocessing/ Synchronisation
Trajectories
Behavior Segmentation
Segmented Trajectories
Imitation Learning
Movement Primitive
Reward Function
Simu-lator
real System
MP
Cost
MP
Cost
Movement Primitive (MP) Refinement
Movement Primitive
Context/ Task Parameter
Skill Template Learning
Skill Template
Movement Primitive
Skill Template Pool
Movement Primitive


Algorithms
Automatisiert
Strukturen
Interaktion
Wahrnehmung
sensor
S
detected_object
X
Einmalig pro Objekt
Objektstruktur
context_item
IT
detected_object
X
Objektseitig
controlled_frame
CF
manipulator_tip
TIP
Einmalig pro Interaktionsart und Objektteil
Bewegungsgenerierung
SP
setpoint
interaction_frame
IF
controlled_frame
CF
interaction_frame
IF
Roboterstruktur
sensor
S
controlled_in
CI
manipulator_tip
TIP
base_link
Einmalig pro Roboter
Roboterseitig
interaction_frame
IF
context_item
IT
Einmalig pro Interaktionsart und Roboter
Regelung
controlled_in
CI
controlled_frame
CF
controlled_in
CI
SP
setpoint

Relations


Kinematic model
Model
Hardware Resources
Robot
Virtual control systems
Motion models
Detection models
Concatenate
Events
Control Network Models
Data Patterns
Combine
Base Components
Labeled software components
Not Discussed
used by

Advantages of eDSL

- **Extensible:** since statements in the eDSL are methods on the objects, extending an eDSL im- plemented in Ruby (i.e. implementing plugins) is simply a matter of adding methods / attributes to existing classes – something that is allowed by the Ruby language
- **Reflexivity:** the one-to-one mapping between the description files and the API ensures that the API is constructive and descriptive enough to allow access to the models, as well as their online modificatio
- **Ability to bind programming and models:** eDSLs have the added advantage that one can easily link the model and the implementation
- **Reuse of the parser and type system of the host language:** one thing that everyone has to do when creating a new programming language is to implement a type system and a parser. Using eDSLs, one can reuse the type system of the host language, and focus on the func- tionality

Joyeux, S., & Albiez, J. (2011). Robot development: from components to systems. In 6th National Conference on Control Architectures of Robots (pp. 1–15).

eDSL: Programming errors and safety

“One natural concern about mixing a model-based approach with a programming approach is the one of safety: how to make sure that programming errors won’t leak into the general system management. These concerns can be addressed easily in a system like Roby. In Roby, errors are represented as objects that are part of a certain context. This context can be a task, a set of tasks or a specific event. When an error appears, various mechanisms allow to (i) handle it and let the system continue or (ii) kill the tasks that are affected to avoid long-term effects.

In this representation, any language exception (Ruby exception) originating from the code in the models (such as: event commands, polling block code, . . .) is caught and transformed in a normal Roby error. In other words, it is caught at the boundaries of the task and injected into the normal Roby exception handling. We believe that, this way, one reaches the same level of safety than with a system where code and models are separated. I.e. it is robust to “obvious” errors (errors that are detected by validation routines inside the code itself), but neither more or less robust to “silent” errors (errors that a diagnostic component could catch).”

Joyeux, S., & Albiez, J. (2011). Robot development: from components to systems. In 6th National Conference on Control Architectures of Robots. Retrieved from http://hal.inria.fr/inria-00599679/