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://rock-robotics.org]
[http://www.ros.org]
[http://wiki.icub.org/yarpdoc/]
• 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
Motivation and Goal

- Support creation of complex robot behavior
- Allow transfer between robots of different morphology
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
Data processing for Cartesian control

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

2. Combine

Describe robot behavior:
- Build compounds of items from Model step
- Extend processing chains

3. Concatenate

Arrange behaviors chronologically:
- Extract patterns from data in component network
- Chronological arrangement of behaviors
Control Network Specification
- **Component model**
  - Orocos RTT
  - Configuration interface
  - Data flow interface
  - Life-cycle
  - Single-purpose

- **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

[http://rock-robotics.org]
User code

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

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

Create configurations for base components

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

Inject information in Instance requirements of compositions

Component Network
Control system specification

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

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

Load specific config for Driver. Give additional information.
Provide kinematic description. Relate devices to robot's body by names.
control_collection "12" 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
Discussion
Different view on development process

Behavior Design

1. Model
   - Build a pool of reusable items:
     - Create model parameters for motion models and object detection algorithms
     - Describe robot kinematics, and devices
     - Create virtual control interfaces

2. Combine
   - Describe robot behavior:
     - Build compounds of items from Model step
     - Extend processing chains

3. Concatenate
   - Arrange behaviors chronologically:
     - Extract patterns from data in component network
     - Chronological arrangement of behaviors

Mission Design

Model
- Task
- Robot
- Model
- Combine
- Refinements
- Subsystems
  - Kinematic model
  - Drivers
- Concatenate
  - Abstraction
- Concatenate
  - Sequencing
- Events
- Pattern Extraction
- Event mapping
- Chronological Arrangement

Motion Descriptions
- Object Descriptions
- Compound Network
- Components

BU: Bottom-Up
TD: Top-Down

Shared between different robots
Robot specific
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

Preprocessing/Synchronisation

Behavior Segmentation

Segmented Trajectories

Imitation Learning

Processed Trajectories

Movement Primitive (MP) Refinement

Simulator

Cost

Movement Primitive

Reward Function

Skill Template Learning

Skill Template Pool

Context/Task Parameter

Movement Primitive
Wahrnehmung

Objektstruktur

Objektseitig

Einmalig pro Objekt

Einmalig pro Interaktionsart und Objektteil

Einmalig pro Interaktionsart und Roboter

Bewegungsgenerierung

Roboterstruktur

Roboterseitig

Einmalig pro Roboter

Regelung

Automatisiert

Algorithms

Strukturen

Interaktion
Hardware Resources

Robot

Kinematic model

Virtual control systems

Detection models

Motion models

Control Network Models

Labeled software components

Base Components

Model

Concatenate

Events

Data Patterns

Relations

Not Discussed

used by
Advantages of eDSL

- **Extensible:** since statements in the eDSL are methods on the objects, extending an eDSL implemented 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 functionality

---

“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).”