



# Flexible Modular Robotic Simulation Environment for Research and Education

Dennis Krupke

Dennis.Krupke@informatik.uni-hamburg.de



University of Hamburg  
Faculty of Mathematics, Informatics and Natural Sciences  
Department of Informatics  
**Technical Aspects of Multimodal Systems**

May 15, 2012



# Table of Content

## Introduction

State-of-the-Art

Requirements of a Flexible and Easy-to-use System

## System Description

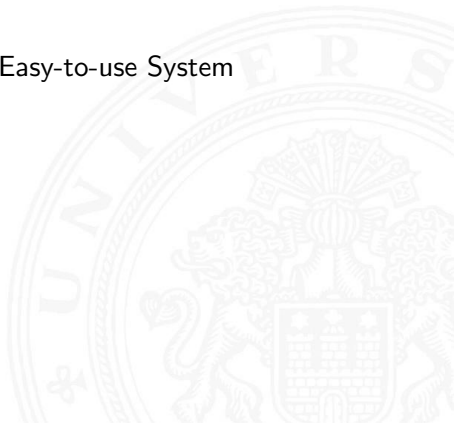
Architecture

Features

## Graphical User Interface

Configuration Interface

Control Interface





# Modular Robots



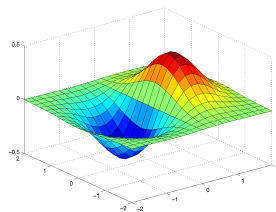


# Simulation Tools

Common systems for simulation of control algorithms

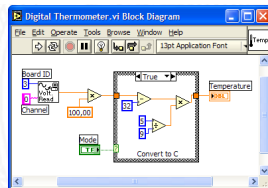
## Focus on mathematics

- ▶ Matlab
- ▶ Octave
- ▶ Scilab



## System flow centered

- ▶ LabVIEW



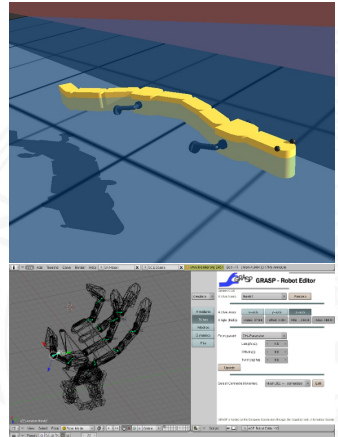


# Simulation Tools Cont.

Systems for simulation and control of robots

## Interactive and integrated systems

- ▶ Player-Project
- ▶ Webots
- ▶ ROS
- ▶ OpenRAVE
  - ▶ OpenGRASP
  - ▶ GRASPiT!



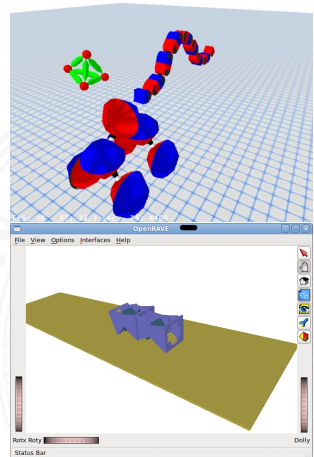


# Simulation Tools Cont.

Systems for simulation and control of modular robots

## Simulating Modular Robots

- ▶ Unified Simulator for Self-Reconfigurable Robots (USSR)
- ▶ OpenMR



# Demands

What is needed for efficient application?

- ▶ **easy-to-use**
- ▶ flexibility
- ▶ useful for beginners and experts
- ▶ reasonable results
- ▶ extendability



# Demands

What is needed for efficient application?

- ▶ easy-to-use
- ▶ flexibility
- ▶ useful for beginners and experts
- ▶ reasonable results
- ▶ extendability





# Demands

What is needed for efficient application?

- ▶ easy-to-use
- ▶ flexibility
- ▶ useful for beginners and experts
- ▶ reasonable results
- ▶ extendability





# Demands

What is needed for efficient application?

- ▶ easy-to-use
- ▶ flexibility
- ▶ useful for beginners and experts
- ▶ reasonable results
- ▶ extendability



# Demands

What is needed for efficient application?

- ▶ easy-to-use
- ▶ flexibility
- ▶ useful for beginners and experts
- ▶ reasonable results
- ▶ extendability

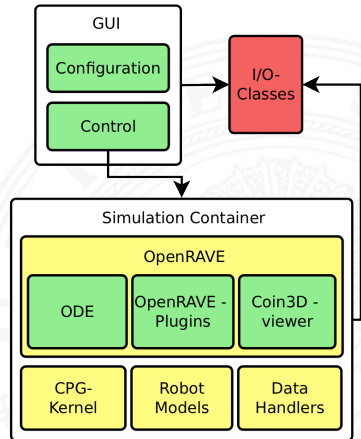




# System Architecture

## Component Based View

- ▶ graphical user interfaces
  - ▶ configuration wizard
  - ▶ expert configuration dialog
  - ▶ control window
- ▶ simulation-/control-core
- ▶ data I-/O
  - ▶ calculated values
  - ▶ configuration files

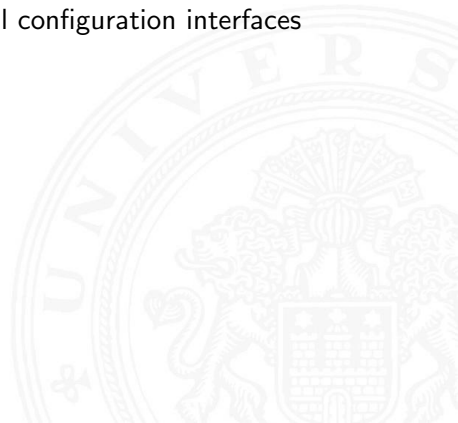




# Main Features

Most important features of the proposed system

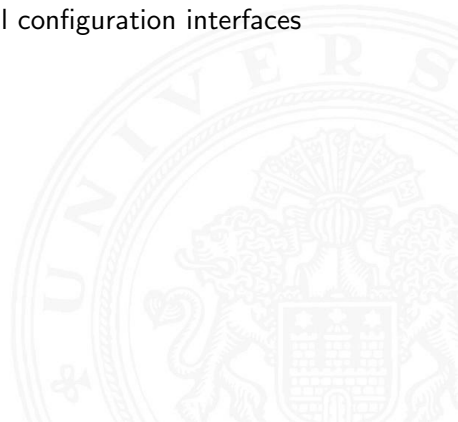
- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins



# Main Features

## Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins

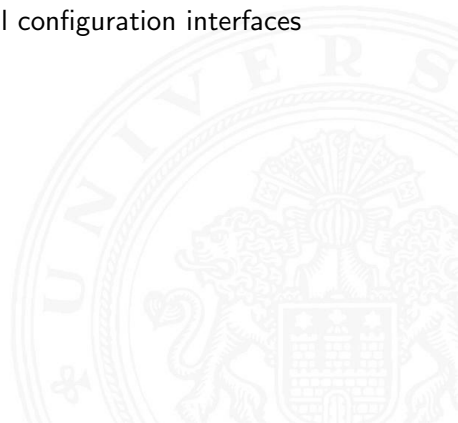




# Main Features

Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins

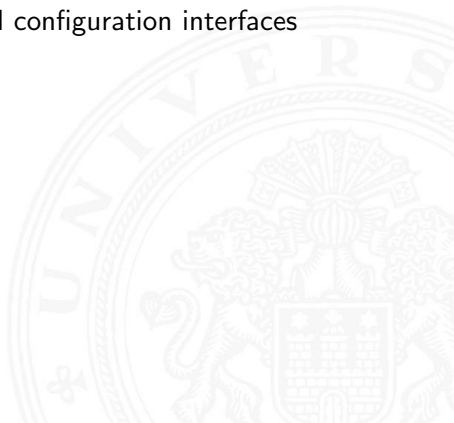




# Main Features

Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins



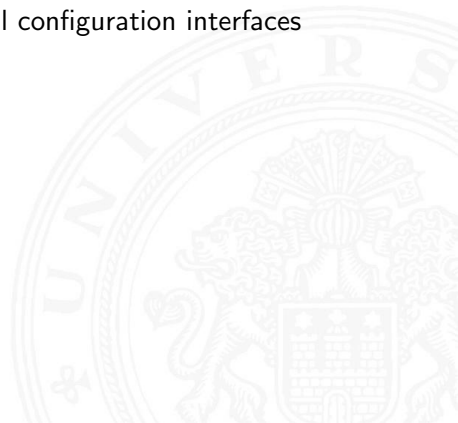




# Main Features

Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins

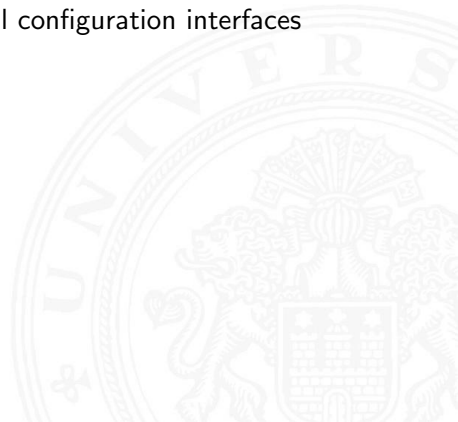




# Main Features

Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins

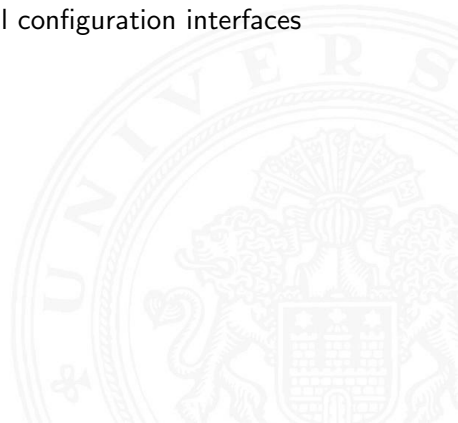




# Main Features

Most important features of the proposed system

- ▶ two different kinds of graphical configuration interfaces
- ▶ reusability
- ▶ extendability
- ▶ interactivity
- ▶ data recording
- ▶ control of real robots
- ▶ support of OpenRAVE plugins





# Easy Configuration of the Simulation

## Configurability

To enable the user to set up a simulation very fast, **configuration file writers** have been created that can be accessed by the GUI:

- ▶ robot
- ▶ sensors
- ▶ actuation
- ▶ environment
- ▶ global properties

▶ Structure of the file format.



# Extending the Library of Control Algorithms

## Extendability

- ▶ New control algorithms can be added by the user with the graphical configuration interface.
- ▶ Combination of **self-registering types** and **dynamic class loading** allows to extend the library of user-defined control algorithms during the runtime of the program.

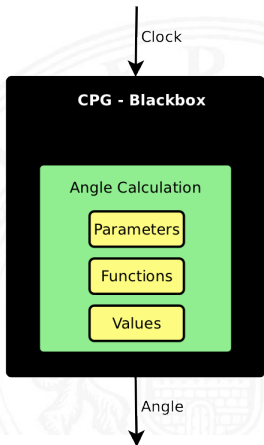


# CPG Blackbox

## User's view to the control algorithms

The user only needs to take care of how to calculate the next joint positions using:

- ▶ parameters
- ▶ functions
- ▶ interim results





# Implementing New Control Algorithms

How can this be done, easily?

Implementing a new control module needs just to add a small code snippet:

```

1  current_time = old_time + Stepsize;
2  for (int jointNr=0; jointNr<_numOfJoints; jointNr++)
3  {
4      current_angle(jointNr) = Amplitude
5          * sin(2*PI * Frequency * current_time
6              + jointNr * PhaseDifference);
7
8      SetAngle(jointNr, current_angle(jointNr));
9  }
```



# Data Handling

All data of interest can be stored to XML-formatted files.

- ▶ control algorithms
- ▶ sensor information
- ▶ robot information

## Exporting

Data series of single types can be exported with the GUI for later usage with GNU-Plot, Matlab or other tools.





# Configuration GUI

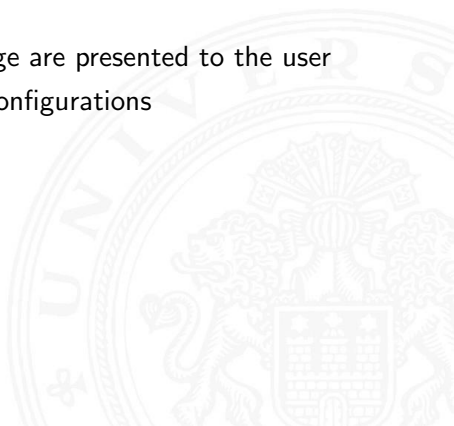
The configuration GUI allows to write down all information to a `configuration file` that is necessary to run a proper simulation.

- ▶ robot
  - ▶ modules
  - ▶ topology of joints
  - ▶ sensor positions
- ▶ sensors
- ▶ control algorithms and their assignment to the joints
- ▶ environment
- ▶ simulation parameters



## Beginner's Configuration Wizard

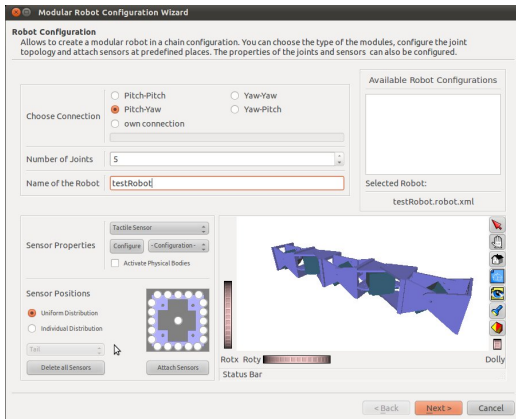
- ▶ every necessary adjustment will be done until the wizard is finished successfully
- ▶ explanations of the current page are presented to the user
- ▶ mandatory fields assert valid configurations



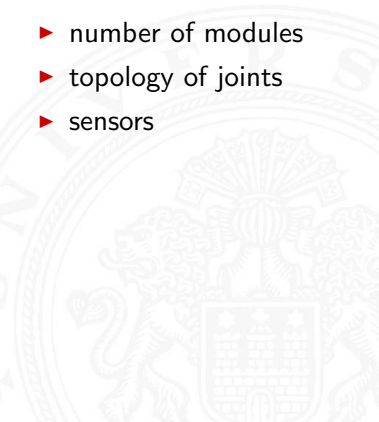


# Beginner's Configuration Wizard Cont.

## Robot configuration



- ▶ number of modules
- ▶ topology of joints
- ▶ sensors





# Beginner's Configuration Wizard Cont.

## Actuation generator definition

**Modular Robot Configuration Wizard**

**Actuation Generator Definition**  
A new control mechanism for a group of joints of a modular robot in chain-like configuration can be defined.

Fields:  
- Name: SmootherSinusoidalGenerator  
- Use Sensors:   
- PhaseDifference: 0,400000 (with dropdown arrow) [Add Parameter]  
- SmoothedAngle [Add Result Type]  
- Prototype: void SmoothAngle(double unsmoothedAngle) [Add Function]

**Summary**

**NAME OF THE NEW ACTUATION MODULE:**  
SmootherSinusoidalGenerator

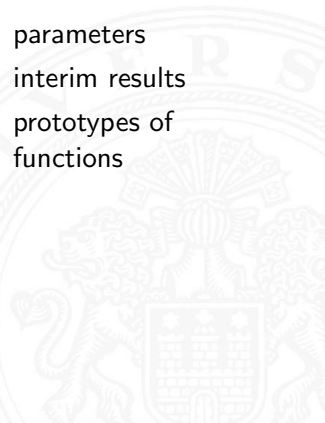
**PARAMETERS:**  
Amplitude  
Frequency  
PhaseDifference

**VALUE TYPES:**  
SmoothedAngle

**PROTOTYPES:**  
void SmoothAngle(double unsmoothedAngle)

Buttons: < Back, Next >, Cancel

- ▶ parameters
- ▶ interim results
- ▶ prototypes of functions





# Beginner's Configuration Wizard Cont.

## Actuation generator implementation

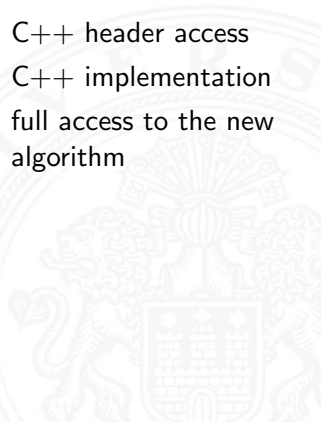
Modular Robot Configuration Wizard

**Actuation Generator Implementation**  
The implementation of the previously declared actuation generator can be done with this dialog. Types and Parameters can be accessed by their names while implementing the function-prototypes especially the ComputeAngles() Function

Header File	Implementation
<pre> SmootherSinusoidalGenerator(int);  // applies initial values for the used movement generator and res virtual void Reset();  // computes the angles for the number of joints given by the con virtual void ComputeAngles();  // returns the whole value frame after computing the angles for l QVector&lt;QVector&lt;double&gt;&gt; CalculateValues();  // returns a pointer to a copy of this object virtual ActuationModule* cloneSelf();  private: // these are your new parameters double* Amplitude; double* Frequency; double* PhaseDifference;  // returns a pointer to the current value of 'SmoothedAngle' of t double* current_SmoothedAngle(int jointNr); // returns a pointer to the last value of 'SmoothedAngle' of the j double* old_SmoothedAngle(int jointNr);  // these are your new functions void SmoothAngle(double unsmoothedAngle); }; #endif // SMOOTHERSINUSOIDALGENERATOR_H </pre>	<pre> QVector&lt;QVector&lt;double&gt;&gt; SmootherSinusoidalGenerator::C // calculates one step (are values for each joint are once calculat ComputeAngles(); return _internalData[1]; }  // sets the internal storage to 0.0 void SmootherSinusoidalGenerator::Reset() { for(int i=0; i&lt;_internalData.size(); i++) for(int m=0; m&lt;_internalData.at(0).size(); m++) _internalData[i][m].fill(0.0); }  ..... * You should leave the code above as it is. If you know what you ar * But in general you only need to implement the following functio .....  void SmootherSinusoidalGenerator::ComputeAngles() // copies new values to old _internalData[0] = _internalData[1]; // Here (after this comment) the calculation of the angles must be } </pre>

< Back   Next >   Cancel

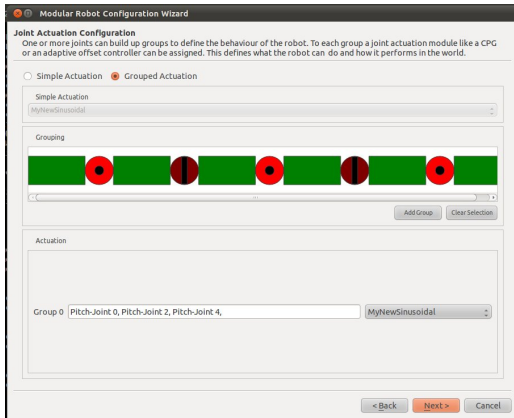
- ▶ C++ header access
- ▶ C++ implementation
- ▶ full access to the new algorithm





# Beginner's Configuration Wizard Cont.

## Joint actuation configuration

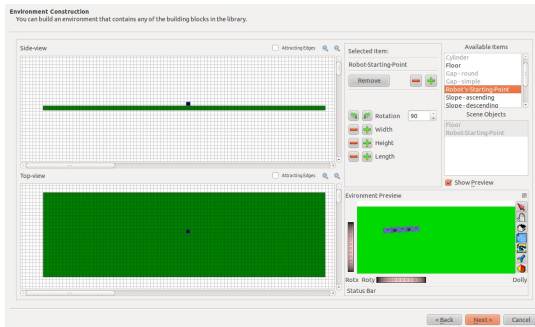


- ▶ building groups of joints
- ▶ assignment of algorithms to groups
- ▶ groups are allowed to overlap



# Beginner's Configuration Wizard Cont.

## Environment construction



- ▶ creation of several objects
- ▶ terrain can be created according to the needs
- ▶ previewing in a 3D viewer
- ▶ manipulation of the scene with two 2D projections



# Beginner's Configuration Wizard Cont.

## Simulation properties

**Modular Robot Configuration Wizard**

**Simulation Properties**  
 The first group shows where the definitions you made so far are stored. In the second group global simulation parameters must be defined to simulate properly. The most important parameter is the simulation mode, because it defines how the

Simulation Mode	Single Run Simulation
Genetic Algorithm -- Steps to go	150000
Sampling Rate	30
Simulation Step Width (OpenRAVE / ODE)	0,0010
Physics Engine	ODE
Gravity	9,81

Enable Storing of Computed Data  
 Show Viewer  
 Enable Experimental Parallel CPG-Computing

Robot File:   
 Actuation File:   
 Environment File:

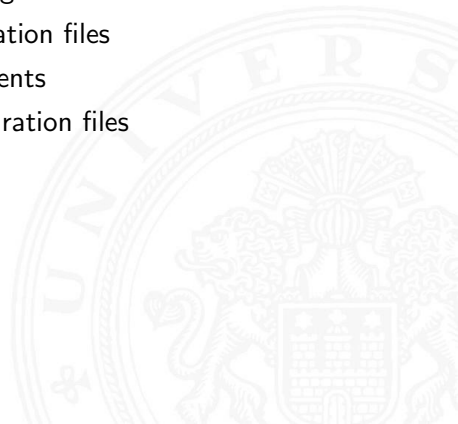
- ▶ simulation mode
- ▶ physics properties
- ▶ sampling and accuracy
- ▶ storing data
- ▶ summary of included configuration files





# Expert's Configuration Dialog

- ▶ users can decide what to configure
- ▶ separated creation of configuration files
- ▶ useful for adding new components
- ▶ time-saving *reusage* of configuration files
- ▶ recombinations are possible

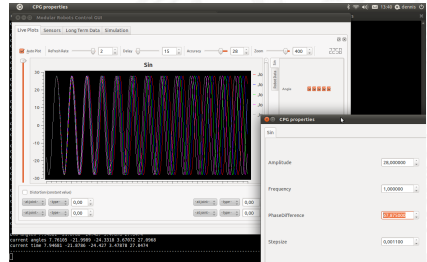




# Virtual Robot

## Controlling a Virtual Robot

- ▶ manipulation of the scene with the 3D viewer
- ▶ interactive modulation of the control algorithms
- ▶ supervision of control algorithms → live-plots

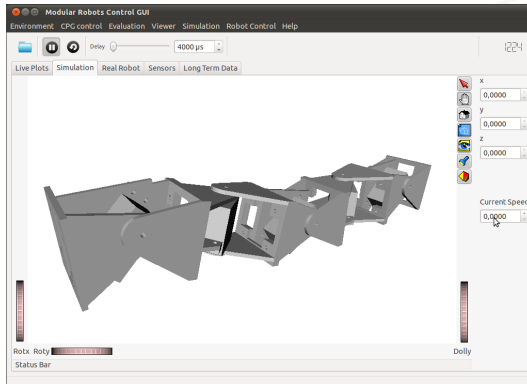




# Real Robot

## Controlling a Real Prototype

Same implementation of control algorithms can be used for real robots:

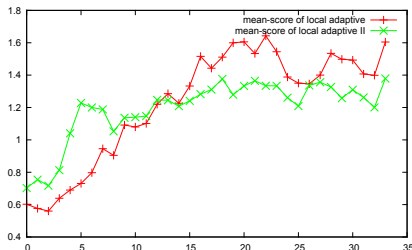




# Future Work

## Next Steps

- ▶ extending the configuration interface
  - ▶ integration of *module creation* into the configuration interface
  - ▶ adding a page to set simulation runs for automated optimization
- ▶ evaluation by people at different knowledge level





End

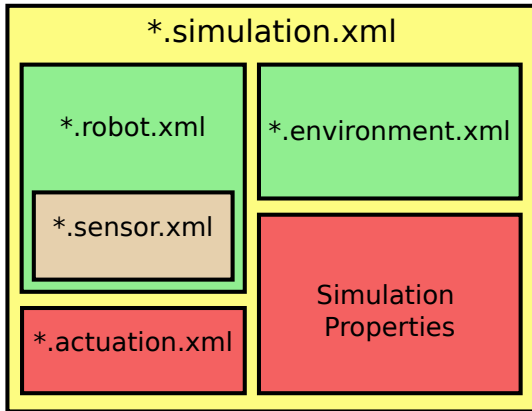
The End!

Thank you for your attention!





# Configuration File Format



- ▶ simulation
- ▶ robot
- ▶ sensor
- ▶ actuation
- ▶ environment

◀ Back to feature description.

◀ Back to configuration interface description.



# Self Registering Types

Each class gets a factory-proxy that registers the *name* of the current class and a *pointer to its maker-function*:

```

1  class FactoryProxy {
2  public:
3      FactoryProxy(){
4          // registers the maker-function
5          Factory["ReflexControl"] = maker;
6      }
7  };
  
```

- ▶ allows very flexible software
- ▶ users can extend the software at runtime



# Factory

## How to Use a Factory

### Construction

The right side of the assignment calls a *maker-function* which invokes the constructor of the current class and returns a pointer to the created object.

```
ActuationModule* controlModule = cpgFactory["MTRAN"](5);
```

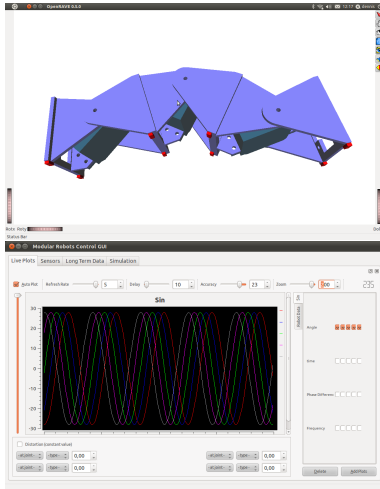
⇒ Objects can be created, even if the name of the specialized class is not known directly.

◀ Back to feature description.





# Visualization



- ▶ Coin3D / OpenRAVE-Viewer
- ▶ QWT – live-plots



## Other libraries

- ▶ Qt
- ▶ ODE
- ▶ GAlib
- ▶ Boost
- ▶ OpenMR

