The DLR Robots library – Using replaceable packages to simulate various serial robots

Tobias Bellmann[1] Andreas Seefried[1] Bernhard Thiele[1]

[1]Institute of System Dynamics and Control, German Aerospace Center (DLR), Germany,
{firstname.lastname}@dlr.de

Abstract

In order to simulate different kinds of serial robots, the implementation of functionalities such as the calculation of their direct and inverse kinematics, visualization, collision behavior, etc. is necessary. However, providing these functionalities in robot specific models leads to additional modeling overhead in cases where one would like to switch between several different robot models. The DLR Robots library demonstrates an implementation of all robot specific components as replaceable Modelica packages, allowing for an user-friendly way to exchange robot models without modifying the general structure of the overlying model.

Keywords: robots, replaceable packages, path-planning, inverse kinematics, LUA scripts

1 Introduction

The simulation of robotic systems is a great example for the multi-domain versatility of Modelica, combining multi-body mechanics with controllers, electric drives and algorithms, e.g. path-planning. A multitude of scientific works uses Modelica to simulate a specific robot, providing models for the mechanics and all other components exclusively for the simulated robot model (Kazi et al., 2002; Hirzinger et al., 2005; Dwiputra et al., 2014; Brossog et al., 2014). Other approaches use parameter sets to simulate more than one robot model within a given structure (Reiner, 2011). However, both approaches lack flexibility regarding switching between different robot types in a Modelica model, e.g. if the number of axes is changing. In this case, instead of changing a parameter value, the complete model structure has to be altered and adapted for the new robot model. In this paper, a new approach to model robots in Modelica is presented. By separating the robot functionalities (e.g. visualization, dynamics, path-planning, etc.) from the model-based description of the robot itself, and wrapping the latter in a replaceable package, it becomes possible to switch between entirely different robot models without the need for changing the structure of the main model. This approach is inspired by the Modelica Media library (Casella et al., 2006), where different media provide their own functions and models, also wrapped in a replaceable package, enabling the user to switch easily between different media in a model.

2 Structure of the library

The library is structured into the following sub-packages shown in Figure 1. The functional blocks e.g. for the calculation of direct or inverse kinematics, visualization functionalities, dynamic models, path-planning or collision detection can be found under `Robots.Blocks`. The sub-package `Robots.RobotModels` mainly contains the base class for the replaceable package `baseRobotModel` defining all common properties, functions and records. The implementation of two fictive robot models demonstrates the usage of this base class. A virtual robot controller is available in the `Robots.Controllers` sub-package, allowing to interpret robot programs and generate reference trajectories for the robot models. The `Utilities` and `Functions` sub-packages provide helper models used in the blocks and examples.

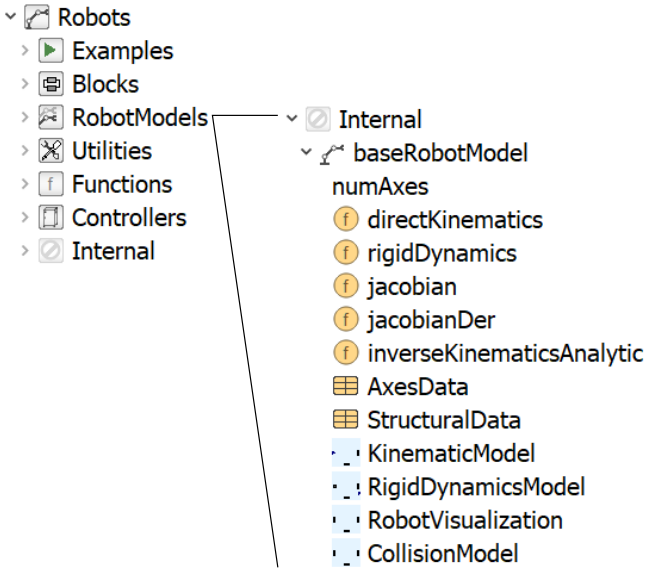

Figure 1. Overview over the DLR Robots library and the partial baseRobotModel package

2.1 Structure of a robot package

All specific robot models extend from the aforementioned base package `baseRobotModel`. This partial package defines the basic sub-models and their interfaces which are required for a robot to function within this library. Figure 1 shows the base package components to be overloaded

and defined by the user implemented specific robot model:

Constant `numAxes`: The integer constant `numAxes` defines the number of axes and is an important parameter for all other blocks, defining their input / output dimensions.

Overloaded functions: For certain functionalities in algorithms (e.g path-planning algorithms), a robot must provide several Modelica functions: the `directKinematics` function calculates the Cartesian pose consisting of `r[3]` (position) and `T[3,3]` (orientation matrix) of the robots end-effector from its joint angles `q[numAxes]`:

```
(r,T) = directKinematics(q)
```

The `rigidDynamics` function returns the torques `tau[numAxes]`, the mass inertia matrix `M[numAxes,numAxes]` and the additional torques generated by the Coriolis and gravitational forces `tau_aux[numAxes]`:

```
(tau,M,tau_aux) =
    rigidDynamics(q,qdot,qddot)
```

In case a linearization of the direct kinematics is necessary, the Jacobian and its derivative have to be provided as well:

```
J = jacobian(q)
J_der = jacobianDer(q,q_dot)
```

In some cases, the inverse kinematics of a serial robot can be calculated analytically. The function `inverseKinematicsAnalytic` can be used to define it:

```
q = inverseKinematicsAnalytic(r,T)
```

Parameter records: Every robot model has to provide two records to define its basic mechanical and dynamic parameters. The record `AxesData` contains all axes related data, e.g. minimum and maximum joint angles/extensions, maximum joint velocities and accelerations, as well as torque limits or gear transmission ratios. The `StructuralData` record should be used to define the mechanical dimensions of the robot, e.g. the distances between the robot joints as well as the position of the centers of gravity of the single robot components. The `StructuralData` record provides no strict naming convention to be overloaded, but is intended to be used freely by the robot model designer to hold all structural information of the robot.

Overloaded models: Every user defined robot model package extending the `baseRobotModel` package also includes a number of models to be used by the functional blocks. The `KinematicModel` model uses standard MultiBody components as joints, rotations and translations to build the kinematic chain between the robots base and its tool center point (TCP). The coordinate frames of every robot axis are provided as an array of MultiBody frames ()`frame_axes`).

The `RobotVisualization` model encapsules the visualization components of the DLR Visualization library used for the real-time visualization of robot systems. It does not contain a kinematic structure, but attaches a visualizer block to every frame of every axis, provided by the `frame_axes` input.

The `RigidDynamicsModel` model uses standard Modelica MultiBody components to model a rigid dynamics model of the robot, including masses and inertias of the robot arm. Similar to the `KinematicModel`, MultiBody joints are used to build the kinematic, but this model also describes the robot components' masses (See Figure 2) and therefore allows the calculation of the dynamic forces acting them.

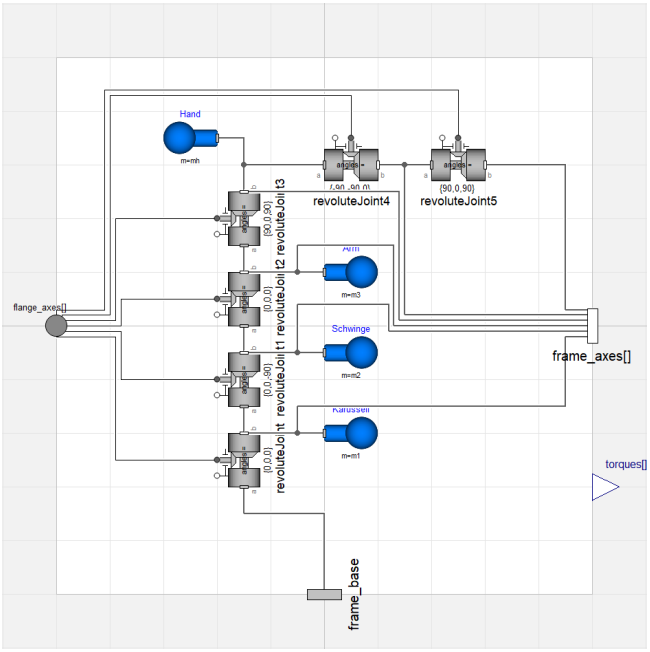

Figure 2. Example for a simple rigid dynamics model for a six axes industrial robot

The `CollisionModel` provides an interface to the DLR ContactDetection library, used to calculate collisions between the robot, itself and other components in its reach. Like the `RobotVisualization` block, the `CollisionObject` blocks modeling the shape of the robot are attached via the connector array `frame_axes`.

2.2 The robotChoice partial model

In order to exchange the robot package in a functional model, the base class `Internal.robotChoice` is provided as an interface definition to parameterize the robot model:

```
partial model robotChoice
"A base class providing the dialog option
    to choose the robot model."
replaceable package robotModel =
    Robots.RobotModels.BelloBot
    constrainedby
Robots.RobotModels.Internal.baseRobotModel
    "Robot model to be simulated"
annotation (choicesAllMatching=true);
...
end robotChoice;
```


DOI 10.3384/ecp20174153

Within the functional block extending this partial interface definition, the robot package can now be chosen and used with the parameter `robotModel` (see Figure 3).

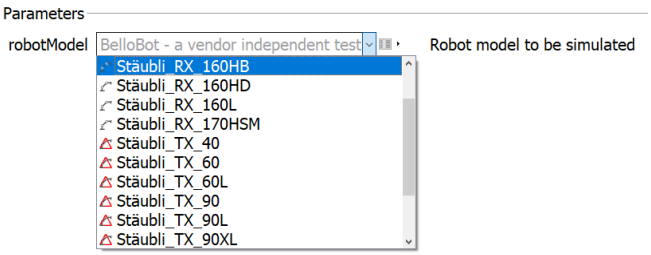

Figure 3. Dymola parameter dialog showing the selection of the robot model

For example, if the user requires access to the maximum axis acceleration, this could be achieved by the following code:

```
extends Robots.Internal.robotChoice;
Real q_ddot[robotModel.numAxes];
...
q_ddot = robotModel.AxesData.q_ddot;
```

3 Utilizing functional blocks to build models with serial robots

The DLR Robots library provides a multitude of robot model independent functionalities. These functional models from the subpackage `Robots.Blocks` utilize the aforementioned `robotChoice` parameter to select the robot model to be used and therefore allow the user to design robot model independent simulation models. Figure 4 shows some of the available blocks of this sub-package.

Visualizer blocks: The `Visualizer` blocks allow the user to visualize the simulated robot with the DLR Visualization library.The DLR Visualization library is used to provide real-time visualization of the robot and its surroundings.

Direct kinematics blocks: Blocks from the package `DirectKinematics` enable access to the axes limits (`AxesLimits` model) or provide models wrapping the robots direct kinematics, Jacobian and its derivative. The user can define additional TCP transformations in order to account for different tool center points.

Inverse kinematics blocks: The `InverseKinematics` package holds several algorithms to calculate the robots inverse kinematics. If an analytical inverse kinematics is provided by the selected robot model package, the `AnalyticalInverseKinematic` model can be used, which simply wraps the function `robotModel.inverseKinematicsAnalytic`. The `DampedLeastSquares` block calculates the inverse kinematics numerically via the well-known damped least squares algorithm (Wampler, 1986). It uses the

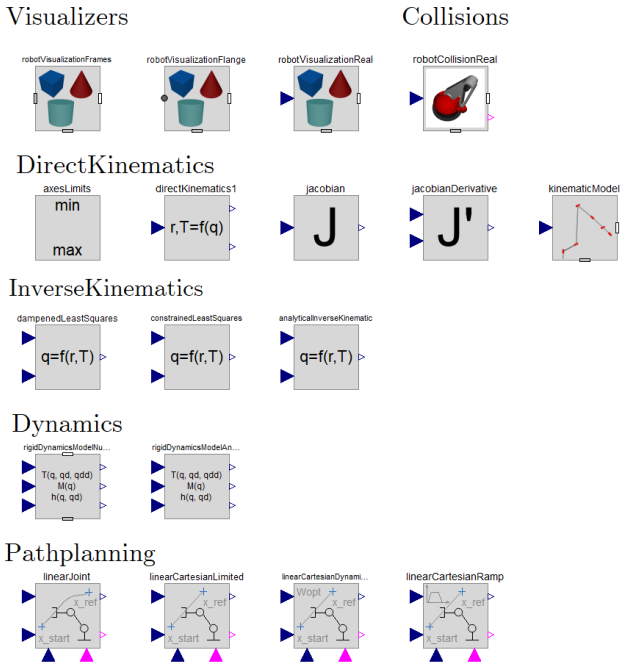

Figure 4. A selection of functional block, to be parameterized with the desired robot package

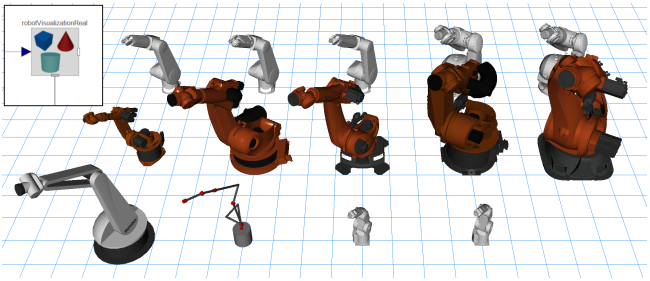

Figure 5. The same visualization block is used with different robot packages selected, visualizing Stäubli, KUKA, Mitsubishi and fictional robots

`directKinematics` and `jacobian` functions from the selected robot package to calculate the joint angles to a given end-effector position via an iteration loop. The `ConstrainedInverseKinematics` model implements the algorithm from (Bellmann et al., 2011b) to generate the joint trajectory resulting in a Cartesian movement of the end-effector, where the reference pose is followed as closely as possible. Here, joint limitations such as minimum and maximum joint angles, velocities, accelerations and torques are considered. In every time step of the calculation, the optimal change of the joint angles Δq is calculated via a local optimization step. The optimization criterion demands a minimization of the pose error, but can also contain additional sub-criteria, for example the desired configuration of redundant robot kinematics with more than six axes.

Robot dynamics blocks: The `Dynamics` subpackage contains models to calculate the robot joint torques/forces, mass inertia matrices and forces act-

ing on the robot structure, either by encapsuling the `robotModel.rigidDynamics` function or utilizing the `RigidDynamicsModel` to calculate them numerically.

Path-planning blocks: The DLR Robots library provides several path-planning algorithms intended to generate e.g. point-to-point (PTP) trajectories between two poses. The `LinearJoint` and the `LinearJointAxes` blocks generate time-optimal PTP trajectories between two poses by interpolating between them in joint space. Maximum joint velocities and accelerations from `robotModel.AxisData` are also considered. The `LinearCartesianLimited` and `LinearCartesianDynamicProgramming` blocks also generate PTP trajectories between two poses, but always on a straight line, interpolated in Cartesian space. The first one is a very fast two-pass algorithm considering the velocity and acceleration limits, but the resulting trajectory can exceed them in the vicinity of numerical singularities ($||J|| < eps$). The second one implements a Dynamic Programming approach to calculate the joint trajectories in a time optimal way, always considering the axes' dynamical limits.

The `RealtimeMovement` block is basically an inverse kinematics block to be used for sensor guided reference trajectories, where a reference pose trajectory is given and the joint angles have to be calculated accordingly and checked for feasibility.

The `TeachBox` block calculates simple and slow movements around the Cartesian axes, in order to implement the possibility to control the robot by a classical teaching interface.

All PTP path-planning blocks can be triggered via a Boolean input. Upon activating the trigger, the current reference pose `(r_ref, T_ref)` is then stored as the target pose, and the movement from the current robot configuration `q_robot` starts. After finishing the movement, the Boolean output `finished` becomes true, indicating the completion of the movement.

4 Using LUA scripts as robot programs

In the real world, a robot program defines a sequence of robot operations, for example different kind of movements, activation of tools, etc. Robot programs must be able to react to external signals, e.g. from a high-level programmable logic controller, or sensors. Nearly every manufacturer provides its own language for their robots, for example KUKA KRL or Mitsubishi MELFA Basic. In order to program a simulated robot in Modelica, different methods can be thought of, for example using Modelica state machines, Modelica functions generating reference positions over time or Modelica models to be parameterized with the reference trajectories and operation activities. However, all these pure Modelica implementations generate significant overhead and require a recompilation of the model if the program has to be changed. In the DLR Robots library, a different approach has been chosen. By utilizing the Modelica/C interface and the external object mechanism, a LUA[1] interpreter can be coupled with the Modelica robot model, using a LUA script as robot program. The flexibility of LUA allows the user to exploit all aspects of a high-level scripting language such as variables, mathematical operations or a feature-rich command-set, but also the usage of custom designed commands tailored to the needs of robot programming, e.g. for PTP movement commands.

4.1 Coupling the LUA interpreter with Modelica

The components of the LUA/Modelica conglomerate are shown in Figure 6. One of the advantages of LUA is the very sleek implementation of its interpreter in C, consisting only of a handful .c/.h C-files. The Robots library provides a small C++ library integrating the following components:

- the interface to the Modelica External Object (see Table 1),
- the LUA interpreter,
- a data core to hold the robot program states like reference position, current command, etc.,
- helper functions to load and execute the LUA script, and
- custom LUA function definitions to be used in the LUA script to control the robot (see Table 2)

The LUA interpreter runs in a separate thread and therefore does not block the execution of the simulation process in Modelica. The Modelica External Object interface to said C++ library comprises of several Modelica functions listed in Table 1.

To enable the user to program robot operations in LUA, a set of custom LUA functions must be provided. Table 2 shows the available commands, which can be used in a LUA script to control the robot in Modelica.

4.2 The robot controller in Modelica

The DLR Robots library provides a robot controller model (`Robots.Controllers.ControllerSixAxes`) suited for the control of six axes industrial robots. Figure 7 shows the interfaces of this model. The inputs and outputs can be utilized by the LUA robot program with the I/O commands from Table 2, allowing the controller to react to signals from the Modelica model. The robot joint sensor input `q_robot` is used by the controllers' path-planning in order to determine the start point for planned trajectories and whether the trajectory is finished. Said trajectory is provided via the robot reference joint angle output `q_ref` and can be subsequently used as an

[1] `https://www.lua.org/`

DOI 10.3384/ecp20174153

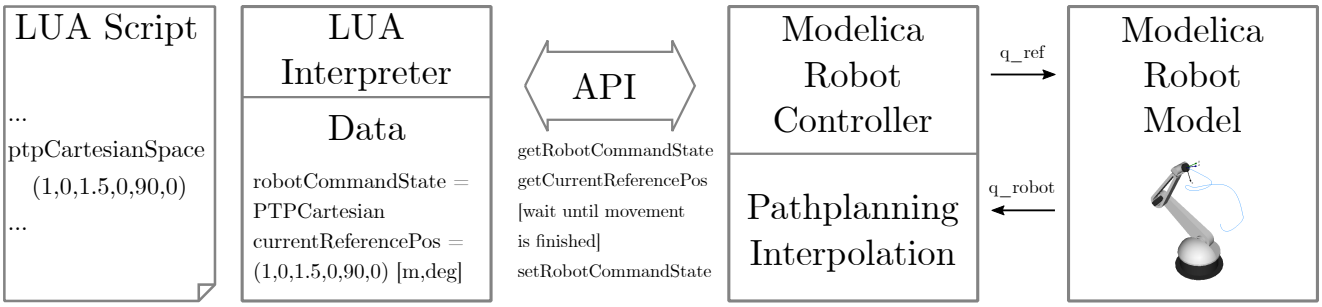

Figure 6. Overview of the LUA/Modelica integration. The LUA script is executed by the LUA interpreter. The Modelica model communicates via the external object API with the LUA Interpreter and handles the physics simulation of the LUA program commands.

Table 1. Modelica functions utilized by the External Object (con-/destructor omitted)

Function	Description
`open`	Opens a LUA script
`run`	Runs the loaded LUA script in a parallel thread
`getRobot↩ CommandState`	Returns the current command state of the robot program
`setRobot↩ CommandState`	Sets the current command state of the robot program
`setNumAxes`	Defines the number of axes in programs
`getReference↩ CartesianPosition`	Returns the current TCP reference position in Cartesian space
`getAnalogOutputs`	Returns the value of the analog outputs to Modelica
`setAnalogInputs`	Sets the value of the analog inputs in the LUA interpreter
`getDigitalOutputs`	Returns the value of the digital outputs to Modelica
`setDigitalInputs`	Sets the value of the digital inputs in the LUA interpreter
`getOverrides`	Retruns the override values to set a speed scale factor for movements

input for the axes controllers. Internally, the controller consists of three sub-components (see Figure 8). The interpreter encapsules the API from Table 1 and provides the sample clock for the robot controller. The path-planning block contains the trajectory generator models from `Robots.Blocks.Pathplanning.TrajectoryGenerators`, and switches between the different path-planning algorithms depending on the robot command state (e.g. idle, PTP in joint space or Cartesian space, etc.). The interpolator block takes the sampled reference trajectory from the path-planning block and

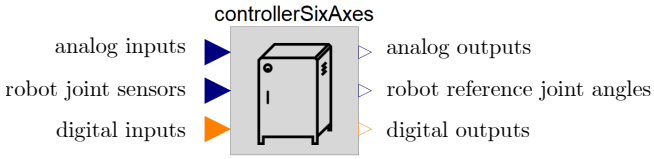

Figure 7. Interfaces of the `ControllerSixAxes` model

interpolates it with various filters (e.g. moving average) to create a smooth reference joint trajectory output.

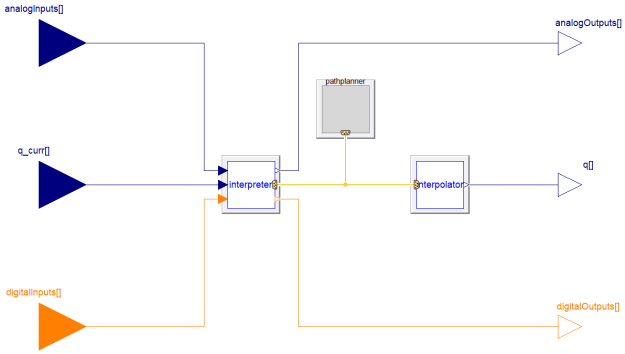

Figure 8. Sub-components of the `ControllerSixAxes` model

4.3 The robot command state

The main arbiter between the LUA interpreted program calls and the Modelica model is the Variable `RobotCommandState`. It determines the current state of the robot and can be set to one of the path-planning command states (*PTPJOINT_CARTESIAN, PTPJOINT_JOINT, PTPCARTESIAN, TRAJECTORY, TEACH*) by the LUA interpreter. Additionally, it can be reset by the Modelica model to *IDLE* by the path-planning block at the end of a movement. Figure 9 shows this alternating interaction between the two executed threads: the LUA interpreter and the Modelica simulation.

Table 2. Custom LUA functions to be used in robot programs

Function	Description
`ptpCartesianSpace` ($\underline{r}_{ref}, \underline{\varphi}_{ref}$)	Commands a linear PTP movement in Cartesian space
`ptpJointSpace`↩ `Angles`($q_1,..,q_n$)	Commands a linear PTP movement in Joint space (reference position in joint angles)
`ptpJointSpace`↩ `Position`($\underline{r}_{ref}, \underline{\varphi}_{ref}$)	Commands a linear PTP movement in Joint space (reference position in cartesian coordinates)
`setDigitalOutput` (*index, value*)	Sets the digital output *index* to *value*. This value can then be used in Modelica to control parts of the model.
value = `getDigitalInput` (*index*)	Returns the *value* of the digital input from Modelica to be used in the LUA script
`setAnalogOutput` (*index, value*)	Sets the analog output *index* to *value*. This value can then be used in Modelica to control parts of the model.
value = `getAnalogInput` (*index*)	Returns the *value* of the digital input from Modelica to be used in the LUA script
`wait`(*time*)	Pauses the robot program execution for *time* seconds
`print`(`string`)	Outputs *string* to the console window.

4.4 Program example - positioning during a welding process

In this example, a LUA robot program is used to position a robot in predefined welding positions and to activate the welding gun. The following listing shows a part of the LUA robot program:

```
print("Welding Program 1")
setDigitalOutput(1,0); //deactivate
    welding gun
wait(1);
ptpJointSpaceAngles(0,-90,45,0,45,0)
ptpCartesianSpace
 (0.37,-1.78,1.21,-42.03,117.47,-35.89)
ptpCartesianSpace
 (0.37,-1.78,1.18,-42.03,117.47,-35.89)
setDigitalOutput(1,1);  //activate welding
    gun
wait(1);  // wait for completion of
    welding process
setDigitalOutput(1,0);  //deactivate
    welding gun
ptpCartesianSpace
 (0.375,-1.78,1.21,-42.03,117.47,-35.89)
 ...
```

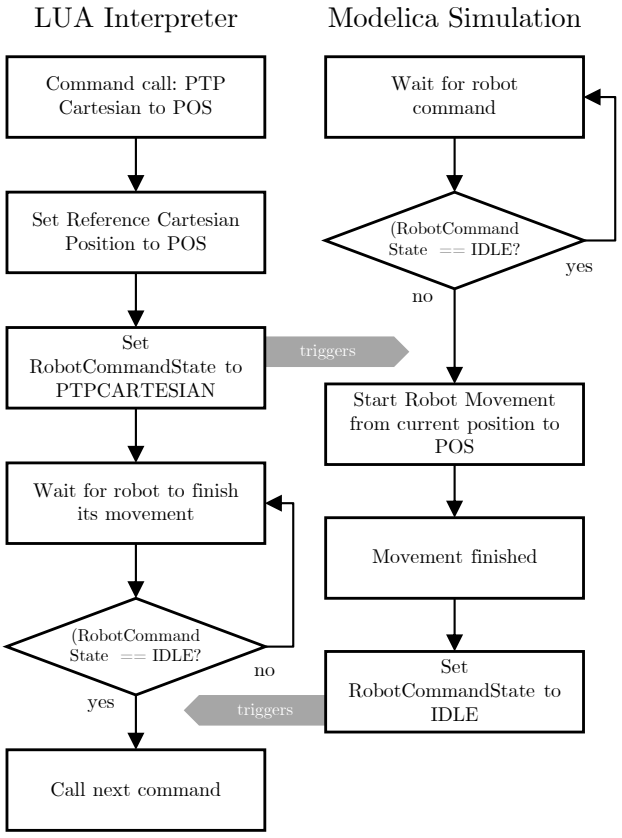

Figure 9. Exemplary interaction between the LUA interpreter thread and the Modelica simulation thread performing a Cartesian PTP movement to the position POS

Figure 10 shows the Modelica model with the aforementioned `ControllerSixAxes`, the robot drive train, kinematics and the robot visualization. The selected robot model is a KUKA Quantec robot, equipped with a welding gun operating on a car frame. The first digital output of the robot controller is connected with the welding gun to control their actuators and to activate the welding process. The robot uses the PTP path-planning blocks of the library to reach the predefined welding positions in the program. After a position is reached, the digital output is activated triggering the closing of the welding gun and ultimately the start of the welding. After waiting for the welding process to finish, the robot moves to the next welding position.

5 Applications

As a base library for robotic research, the DLR Robots library can be used in various simulation and control applications. Some projects utilize it to analyze robotic systems, while others are making use of the real-time capable path-planning algorithms to control real-world robots. All following examples are basically using the same functional blocks introduced in Section 3, but with different robot model packages.

DOI 10.3384/ecp20174153

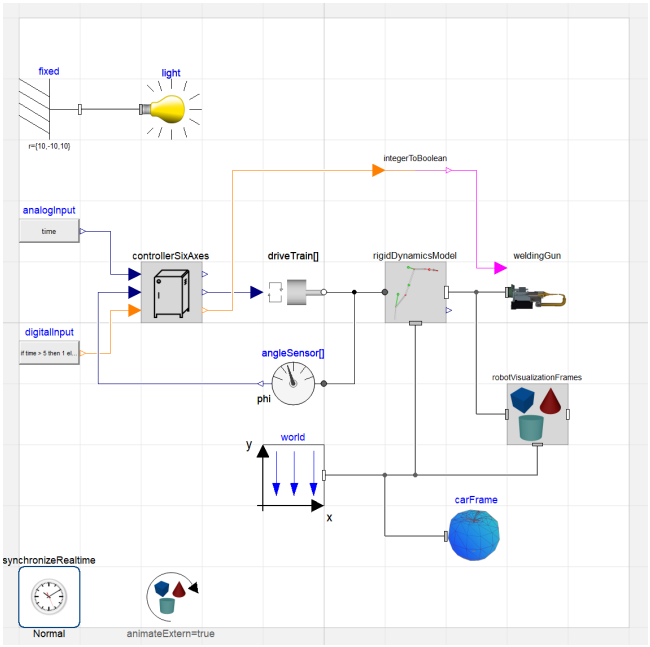

Figure 10. Modelica model utilizing the DLR Robots library to simulate the robot movements of a welding process

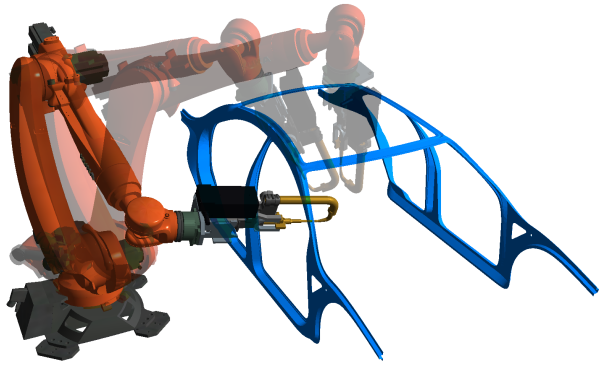

Figure 11. Positioning for multiple welding processes during execution of the LUA robot program

5.1 DLR Robotic Motion Simulator

The DLR Robotic Motion Simulator is a modified KUKA KR500TÜV industrial robot carrying a multi-purpose simulator cockpit (see Figure 12). The system can be used to perform interactive driving and flight simulations (Bellmann et al., 2011a). During the simulation run, the driver/pilot controls the virtual vehicle and the robot moves accordingly in real-time to simulate the movements of the vehicle. In this use-case, the DLR Robots library is used to provide the real-time path-planning of the robot joint trajectories while considering the hardware limits of the system.

5.2 DLR Terramechanics Robotic Locomotion Lab

The DLR Terramechanics Robotic Locomotion Lab (TROLL) is a robotic test bed for automated wheel/soil

Figure 12. The Robotic Motion Simulator (RMS) is a flexible, industrial robot based flight/driving simulator. The path-planning algorithms and kinematic functions of the DLR Robots library are used to control the Robotic Motion Simulator in real-time.

contact tests (Buse et al., 2018). An industrial robot is pressing a space rover wheel into a soil surface, measuring the resulting forces, slip of the wheel, deformation of the soil or even the movement of soil particles via camera based particle image velocimetry. The DLR Robots library has been used to simulate the complete system as part of a feasibility study (see Figure 13), and is also used to control the robot during operations.

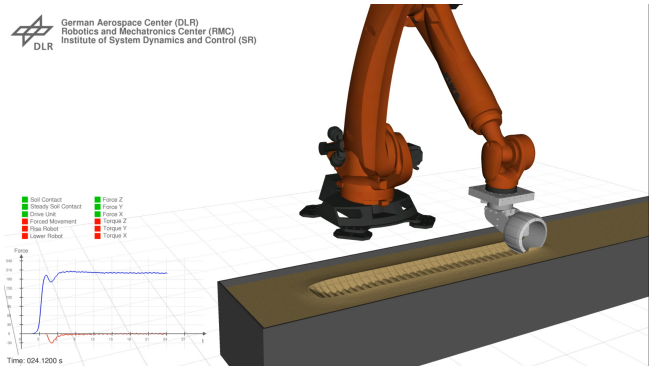

Figure 13. Feasibility study of the DLR Terramechanics Robotic Locomotion Lab - A KUKA KR3100 Quantec presses a rover wheel into a simulated soil surface, while following the rotating wheel with constant speed over the ground

5.3 Active Space debris removal with a robot arm

The increasing density of large debris objects in Low Earth Orbit poses a growing problem as the probability for collisions increases. In order to de-orbit large objects, several approaches are actively researched, such as Active Debris Removal (ADR) utilizing a robot arm. As part of an ESA project, this approach has been simulated utilizing the DLR Robots library to model the robots' arm behavior in a combined control GNC simulation (Reiner, 2018). The chaser satellite first synchronizes with the tumbling

movement of the target satellite (here: Envisat). Next, the robot arm with three axes grabs the docking ring and enables a physical connection between the chaser satellite and the target. Thereafter, the chaser satellite actively steadies the tumbling target and subsequently initiate the controlled de-orbiting. In this use-case, the DLR Robots library was used to simulate the kinematics and dynamics of the robot arm, whereas the drive-trains have been simulated with additional detailed models.The GNC simulation tool based on the object-oriented DLR SpaceSystems (Reiner and Bals, 2014) library and DLR Environments library (Briese et al., 2017) is used to design and simulate the control algorithms, satellite dynamics including flexible elements such as the solar panel, kinematics as well as the robot arm control.

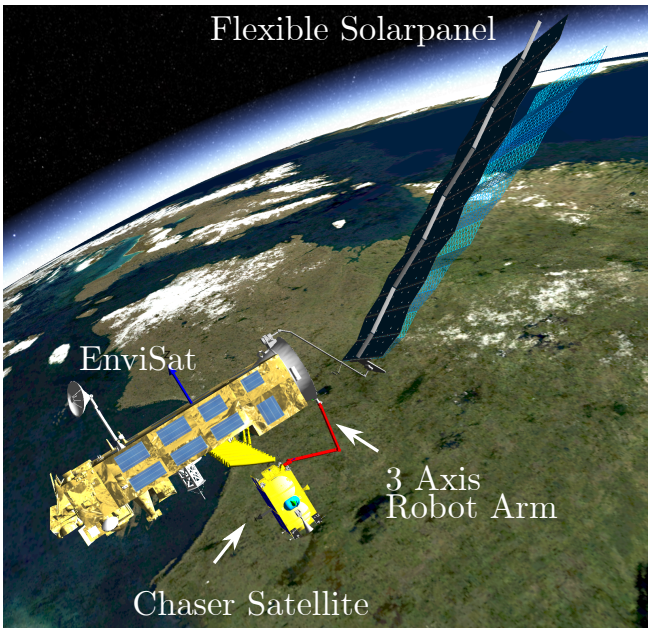

Figure 14. De-orbit scenario simulation of the nonfunctional EnviSat. The capture satellite grasp the tumbling target with a three axes robot arm.

5.4 Analyzing grasping and placement processes during rover missions

For the research project ROBEX (Robotic Exploration of EXtreme Environments) an analog (on earth) mission scenario has been designed, where a rover performs autonomous pick-up and placement of sensor packages (Hellerer et al., 2016; Wedler et al., 2015). In order to analyze the forces and torques acting on both the arm and the rover, the DLR Robots library has been used. Additionally the complete mission scenario has been simulated including robotic operations (see Figure 15).

6 Conclusions

The replaceable package mechanism in Modelica provides the user with great flexibility, as it allows parameterizing models not only with parameters or functions but also

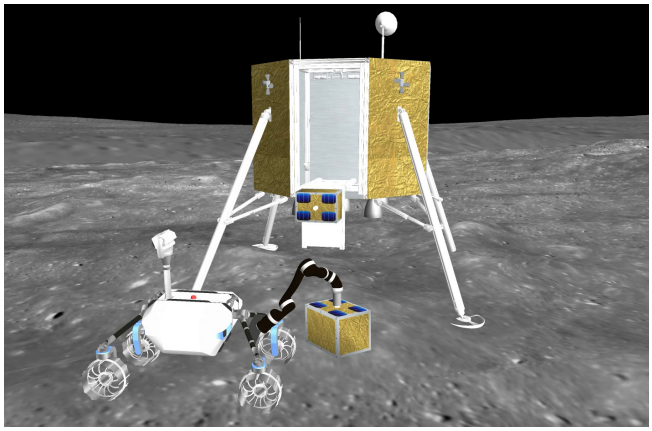

Figure 15. Rover equipped with Jaco Arm lifting a sensor package during the ROBEX mission scenario

complete sets of functionalities and even structural components. This is used in the DLR Robots library to ease the workload of system-modelers and to increase the reusability of Modelica models utilizing robotic systems. The combination of Modelica with the scripting language LUA is a promising method to provide flexible and higher-level control over simulation procedures, such as robot operations. In the future, a more generic LUA library will be developed in order to enable this potential in other domains beyond robotics.

7 Acknowledgments

The authors would like to thank Matthias Reiner and Fabian Buse (DLR) for providing additional example material for this paper. Additionaly we would like to thank Mehran Assanimoghaddam, Stefan Hartweg, Matthias Reiner and Robert Reiser (DLR) for their contributions to the library and valuable discussions.

References

Tobias Bellmann, Johann Heindl, Matthias Hellerer, Richard Kucher, Karan Sharma, and Gerd Hirzinger. The DLR Robot Motion Simulator Part I: Design and setup. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4694–4701. IEEE, Mai 2011a.

Tobias Bellmann, Martin Otter, and Gerd Hirzinger. The DLR Robot Motion Simulator Part II: Optimization based path-planning. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4702–4709. IEEE, Mai 2011b.

Lale Evrim Briese, Andreas Klöckner, and Matthias Reiner. The DLR Environment Library for Multi-Disciplinary Aerospace Applications. In *12th International Modelica Conference*, Mai 2017. URL `https://elib.dlr.de/112971/`.

Matthias Brossog, Johannes Kohl, Jochen Merhof, Simon Spreng, Jörg Franke, et al. Energy Consumption and Dynamic Behavior Analysis of a six-axis Industrial Robot in an Assembly System. *Procedia Cirp*, 23:131–136, 2014.

DOI
10.3384/ecp20174153

Fabian Buse, Tobias Bellmann, Roy Lichtenheldt, and Rainer Krenn. The DLR Terramechanics Robotics Locomotion Lab. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Juni 2018. URL https://elib.dlr.de/121796/.

Francesco Casella, Martin Otter, Katrin Proelss, Christoph Richter, and Hubertus Tummescheit. The Modelica Fluid and Media Library for Modeling of Incompressible and Compressible Thermo-fluid Pipe Networks. In *Proceedings of the 5th international modelica conference*, pages 631–640, 2006.

Rhama Dwiputra, Alexey Zakharov, Roustiam Chakirov, and Erwin Prassler. Modelica Model for the Youbot Manipulator. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 096, pages 1205–1212. Linköping University Electronic Press, 2014.

Matthias Hellerer, Martin J. Schuster, and Roy Lichtenheldt. Software-in-the-Loop Simulation of a Planetary Rover. In *The International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Juni 2016. URL https://elib.dlr.de/104934/.

Gerd Hirzinger, Johann Bals, Martin Otter, and Johannes Stelter. The DLR-KUKA Success Story: Robotics Research improves Industrial Robots. *IEEE Robotics & Automation Magazine*, 12(3):16–23, 2005.

Arif Kazi, Günther Merk, Martin Otter, and Hui Fan. Design Optimisation of Industrial Robots using the Modelica multi-physics Modeling Language. In *33rd International Symposium on Robotics*, pages 347–352, Oktober 2002. URL https://elib.dlr.de/11898/. LIDO-Berichtsjahr=2002,.

Matthias Reiner and Johann Bals. Nonlinear inverse models for the control of satellites with flexible structures. In *10th International Modelica Conference 2014*, Linköping Electronic Conference Proceedings, pages 577–587. LiU Electronic Press, 2014. URL https://elib.dlr.de/92164/.

Matthias J. Reiner. *Modellierung und Steuerung von strukturelastischen Robotern*. PhD thesis, Technische Universität München, 2011.

Matthias J. Reiner. Modelling And Combined Control Of A Satellite With A Robot Arm For Active Debris Removal. In *69th International Astronautical Congress*, 2018. URL https://elib.dlr.de/123349/.

Charles W Wampler. Manipulator Inverse Kinematic Solutions based on Vector Formulations and Damped Least-Squares Methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, 1986.

Armin Wedler, Mathias Hellerer, Bernhard Rebele, Heiner Gmeiner, Bernhard Vodermayer, Tobias Bellmann, Stefan Barthelmes, Roland Rosta, Caroline Lange, Lars Witte, Nicole Schmitz, Martin Knapmeyer, Alexandra Czeluschke, Laurenz Thomsen, Christoph Waldmann, Sascha Flögel, Martina Wilde, and Yuto Takei. ROBEX – Components and Methods for the Planetary Exploration Demonstration Mission. In *13th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, ASTRA. ESAWebsite, 2015. URL https://elib.dlr.de/98242/.