Enhanced Motion Control of a Self-Driving Vehicle Using Modelica, FMI and ROS

Nikolas Schröder[1] Oliver Lenord[2] Ralph Lange[2]

[1]Institute of Flight Mechanics and Control, University of Stuttgart, Germany,
`lrt86824@stud.uni-stuttgart.de`
[2]Robert Bosch GmbH, Germany, `{oliver.lenord, ralph.lange}@de.bosch.com`

Abstract

This paper presents a new planar wheel model with bore friction, a control strategy to avoid locking conditions of floor vehicles with caster wheels, and the new FMI-Adapter software package, which connects the Functional Mock-up Interface (FMI) standard with the Robot Operating System (ROS). It is demonstrated how this technology enables a convenient model-based control design workflow. The approach is applied to the ActiveShuttle, a self-driving vehicle (SDV) for industrial logistics. After modeling the wheel friction characteristics of the ActiveShuttle, a feed forward controller to avoid high friction torques at the caster wheels in critical operation scenarios is designed and validated by model-in-the-loop simulations. The control function is exported as Functional Mock-up Unit (FMU) for co-simulation. With help of the FMI-Adapter package, the FMU is integrated as ROS node into the service-oriented robot control architecture, enhancing the existing motion controller. The functionality and performance is tested and successfully verified on the ActiveShuttle Dev Kit prototype.

Keywords: Modelica, FMI, ROS, Autonomous Systems, Robotics, Model-based Control, SDV, Caster Wheels

1 Introduction

A relevant application area of autonomous robotics is the industrial logistics. In the last years, a number of elaborate algorithms for task scheduling, coordination and path planning for fleets of self-driving vehicles (SDVs) in such applications have been proposed (Imlauer et al., 2016; Pecora et al., 2018). Prerequisite to apply these strategies is a reliable vehicle motion control. Trajectories commanded by the planner need to be properly executed by the drive platform to ensure that the goals of the mission are met in time and space. Safety and security margins have to be met, undesired interference due to deviations from the planned track need to be avoided, and at all times the vehicle must remain maneuverable.

Model-based control design is a well established approach to design and apply motion control strategies. Model-in-the-loop (MiL) simulations allow to validate and test the control design early on. Optimized controllers can be designed that take the physical properties and system dynamics into account (Thümmel et al., 2005).

In this work a common problem of motion platforms with differential drive and caster wheels is elaborated. By applying a model-based control design approach the reliability of the motion controller is significantly improved by the so called *Path Filter* introduced in Section 3.

The path filter module is realized as *ROS node* (see section 1.2) to allow the seamless integration into the service oriented software architecture ROS that is used on the target application *ActiveShuttle DevKit*. The widely used development environment for model-based control, Matlab Simulink, does provide a dedicated toolbox for ROS (The MathWorks). In this paper an alternative approach is applied aiming to leverage the benefits of the physical modeling language Modelica and the rich Modelica libraries such as the Modelica Standard Library (MSL) for the design, verification and validation of the path filter. For this purpose the free `PlanarMechanics` library (PML) (Zimmer, 2014) is extended and used to build up a plant model of the Active Shuttle DevKit (see section 2).

In order to enable a generic and efficient control design process, the integration of the path filter control function into ROS is facilitated through the *Functional Mock-up Interface* (FMI) (Modelica Association Project "FMI", 2014). Related approaches for such integration aim at simulation use cases only: The *Modelica-ROS Bridge* (Swaminathan) allows to integrate the Modelica language and corresponding tools with ROS by a TCP/IP-based bridge, implemented by the ROS_Bridge package for Modelica and a relay node from the ROS modelica_bridge package. A similar mechanism based on Unix IPC sockets has been proposed to integrate Modelica with the Gazebo simulator, which is used heavily by the ROS community to simulate robots in 3D environments (Bardaro et al., 2017). The *gazebo-fmi* project (Traversaro et al.) provides a plug-in to import FMUs in Gazebo.

With the new *FMI-Adapter for ROS* introduced in Section 4, a very generic mechanism is provided to integrate control functions into ROS. An export of the path filter block as FMU (Functional Mock-up Unit) allowed the straightforward integration into the ROS architecture and finally its application and test, as described in Section 5, on the Active Shuttle DevKit.

1.1 Use Case: ActiveShuttle

The *ActiveShuttle* (AS) is a self driving vehicle (SDV) for logistic services on the shop floor. A prototypical small batch series named *ActiveShuttle DevKit* operates at several Bosch plants in Germany. The missions carried out by the AS are bring and pick-up services of stacked container boxes on moving dollies.

The routes between the pick-up and drop-off locations are planned based on a map continuously updated by a simultaneous localization and mapping (SLAM) algorithm (Durrant-Whyte and Bailey, 2006). In order to make the motion of the SDVs predictable for the workers, the routes are restricted to be composed of straight segments only. Turns are restricted to be performed as turns on the spot after standstill only. Thus, the basic motion patterns are:

- Follow a straight line segment.
- Stop and turn on the spot.
- Stop and move in reverse direction.

The AS DevKit is actuated by a differential drive. Two caster wheels are positioned in the front and two in the rear. The chassis design ensures that all six wheels remain in contact to the ground while crossing sills or other uneven grounds.

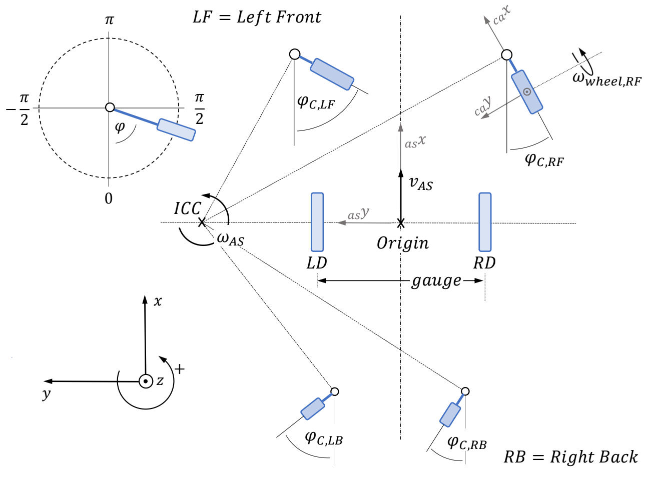


Figure 1. Active Shuttle coordinate system definition, sign convention and nomenclature.

The reference coordinate system of the SDV is fixed to the middle frame and located in between the driven wheel axes when standing on flat ground. Given that the vehicle is operated in the plane only, the motion state of the AS can be described by equation 1 with the longitudinal velocity v_{AS} and the angular velocity ω perpendicular to the moving plane.

$$\boldsymbol{x}_{AS} = \begin{pmatrix} v_{AS} \\ \omega_{AS} \end{pmatrix} = \begin{pmatrix} {}_{AS}v_{x,AS} \\ {}_{AS}\omega_{z,AS} \end{pmatrix} \tag{1}$$

During operation it has to be ensured that the above described motion patterns can be executed in an arbitrary sequence. A critical condition occurs in the transitions from moving straight to turning and vice versa. In these transitions the desired motion state of the AS is inconsistent with the actual motion state of the caster wheels given by equation 2, with the $\omega_{C,i}$ describing the angular velocity of the i-th caster wheel w.r.t. its spinning axis and $\varphi_{C,i}$ describing the orientation relative to the vehicle w.r.t. the vertical axis. The caster wheels' rotational axes are not aligned with the instantaneous center of curvature (ICC) of the SDV, located at the center of the reference frame:

$$\boldsymbol{x}_{C,i} = \begin{pmatrix} \omega_{C,i} \\ \varphi_{C,i} \end{pmatrix} = \begin{pmatrix} {}_{CA}\omega_{y,C,i} \\ {}_{AS}\varphi_{C,i} \end{pmatrix} \tag{2}$$

Due to the fact that the vehicle is at standstill when the turn is initiated, the maximum bore friction torque has to be overcome in addition to the inertial forces. Projecting the friction torque at the i-th caster wheel onto the point of contact of the driven wheels with the radii from the ICC to the driven wheel r_{DW} and caster wheel r_{CW}, reveals that due to $r_{DW} = gauge/2 < r_{CW,i}$ a significant share of the available traction force at the driven wheels is assigned to the bore torque of the caster wheels:

$$F_{DW} = \frac{r_{CW,i}}{r_{DW} \cdot r_{trail}} \cdot T_{bore} \tag{3}$$

If under full load the required driving torque exceeds the maximum motor torque, the SDV is not able to follow the commanded trajectory. Hence, avoiding the risk of this critical state by reducing the impact of the bore friction has been identified as significant contribution to make the operation of SDVs with differential drives more reliable.

1.2 Robot Operating System

The Robot Operating System (ROS) (Quigley et al., 2009) has been used for the development of the AS DevKit. ROS can be considered as a framework and middleware for robotic systems. It also provides a rich set of development tools and basic functional capabilities for perception, control, planning and manipulation. In the last ten years, a huge open-source community has grown around this project, which provides numerous software packages for all aspects of robotics (`www.ros.org/browse/`).

ROS uses a service-oriented architecture with publish-subscribe and request-response communication methods. It even allows to integrate new components dynamically at run-time. ROS supports most prevalent programming languages, particularly including C++, Python, Java, C#, JavaScript, and Ruby. These features facilitate the integration of new technologies with ROS.

2 Physical Model of the SDV

2.1 Model Requirements and Architecture

Based on the use case described in Section 1.1, the following physical effects have been identified that need to be represented by a physical model of the SDV:

- Planar motions of the SDV relative to a fixed ground and related inertial forces.

- Normal forces at the driven and caster wheels considering the chassis kinematics and mass.
- Limited traction of the driven wheels considering the maximum wheel slip.
- Bore friction torque at the caster wheels with stiction.
- Limited drive torque defined by the motor characteristics.

Due to the usage as plant model for MiL simulations with slow accelerations and limited maneuvers, the following simplifying assumptions have been applied:

- No real time requirements.
- No inertial torque along the longitudinal axis.
- Neglect the inertial torque along the lateral axis.

Aiming to keep the physical model as simple and generic as possible the `PlanarMechanics` Library (PML) (Zimmer, 2014) has been chosen as basis for the mechanical model. The library provides all elements required to describe a planar multibody system and provides a set of basic tire models referred to as `WheelJoint` that have been adopted as described in Subsection 2.2. To enable a simple reuse of these basic tire models for the described class of differential drive vehicles, additional components have been developed combining the wheel joints with components from the Modelica Standard Library (MSL).

In a separate `SelfDrivingVehicles` library the extended PLM has bee utilized to build up packages for specific applications such as the AS DevKit and others. The corresponding `System` package contains models that describe the connections between the driven wheels and caster wheels as well as a block to calculate the normal forces of the wheels dependent on the actual orientation.

The `Controller` package contains models of common control concepts applicable to any differential drive vehicle such as motion profiles described in vehicle coordinates and their mapping to command values for the left and right drive as well as implementations of the *Path Filter* introduced in Section 3.

This architecture allows to separate the application specific properties from common concepts of differential drive vehicles. Models of new applications can be created with little effort.

2.2 Wheel Models for Differential Drive

Driven Wheel. The wheel itself is modeled with a `SlipBasedWheelJoint` (PML) and a 1D-rotational `Inertia` component (MSL). The `frame_a` connector of the `SlipBasedJoint` interfaces the driven wheel subsystem with the main structure of the vehicle model. The `Inertia` component is actuated by a `Torque` signal.

Caster Wheel. Figure 2 shows a schematic illustration of a swivel caster wheel. The fork contains a bearing which allows the wheel to swivel relative to the structure it is mounted on. The bearing is modeled with a `Revolute` joint (PML). Its `frame_a` connector interfaces the

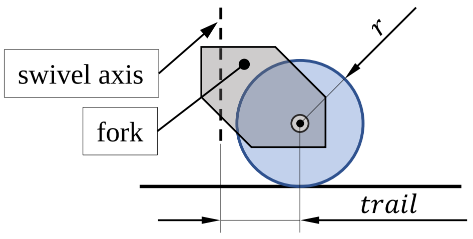


Figure 2. Nomenclature of a caster wheel.

subsystem with the main structure of the vehicle model. A `RelAngleSensor` (MSL) measures the current caster wheel orientation and provides it as `RealOutput`. The fork is modeled with a `FixedTranslation` component (PML) with the length $l = trail$ that connects the `Revolute` with the wheel joint. As specified in Section 2.1, the wheel joint of the caster wheel is required to consider bore friction. Bore friction is a friction torque that counteracts a wheel's rotational movement around its vertical axis. As depicted in figure 3, bore torque is denoted with T_{bore} and is opposed to the acting torque T_z and the angular velocity ω_z. In the following, we first describe a bore friction characteristic that was proposed by (Zimmer and Otter, 2010). Thereafter, we propose an alternative approach that better suits our model requirements and explain how the `IdealWheelJoint` is extended to allow its correct implementation.

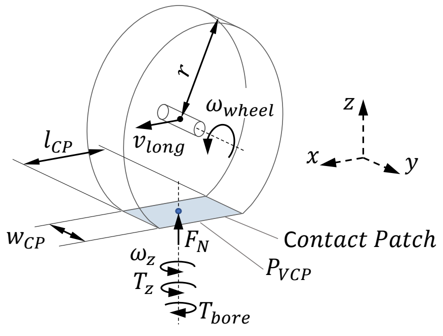


Figure 3. Tire road contact.

Equation 4 shows the proposal by (Zimmer and Otter, 2010).

$$|T_{bore}| = \begin{cases} |T_{bore,max}| \cdot \dfrac{|\lambda_{bore}|}{\lambda_{bore,lim}} \\ \quad \text{if } |\omega_z| \cdot s_{rep} < \lambda_{bore,lim} \cdot |\omega_{wheel}| \cdot r \\ |T_{bore,max}| \quad \text{else} \end{cases} \tag{4}$$

Similar to (Rill, 2007), they state that the friction torque T_{bore} is proportional to the bore slip λ_{bore}. Here, bore slip describes the ratio between the representative slip velocity $\omega_z \cdot s_{rep}$ of the tires contact patch and the wheels linear velocity $\omega_{wheel} \cdot r$.

$$|\lambda_{bore}| = \frac{|\omega_z| \cdot s_{rep}}{|\omega_{wheel}| \cdot r} \tag{5}$$

Within the contact patch, s_{rep} is the distance between the virtual contact point (P_{VCP}) and an infinitesimal element which is representative for the distances of all infinitesimal elements. It results when integrating over the contact patch.

$$s_{rep} = \frac{1}{\sqrt{12}}\sqrt{l_{cp}^2 + w_{cp}^2} \tag{6}$$

The bore torque is limited to

$$|T_{bore,max}| = F_N \cdot \mu_{bore} \cdot s_{rep}. \tag{7}$$

Here, μ_{bore} denotes the friction coefficient between tire and road, F_N the wheel contact force that acts on P_{VCP} (cf. figure 3). The limiting parameter in equation 4 is the bore slip limit $\lambda_{bore,lim} > 0$ that determines at which bore slip value the maximum bore torque is reached. Due to the fact that the actual bore slip λ_{bore} is not defined for $\omega_{wheel} = 0$ (cf. equation 5), the condition $\lambda_{bore} < \lambda_{bore,lim}$ in equation 4 is expressed such that division by zero is avoided.

Figure 4 shows a visualization of equation 4. When $|\omega_z| = |\omega_{wheel}| = 0$, the acting torque $|T_z|$ has to overcome the stiction torque $|T_{bore,stic}| = |T_{bore,max}|$. As long as $|T_z| < |T_{bore,stic}|$, the caster wheel is caught in a locking condition and cannot change its orientation around its vertical axis.

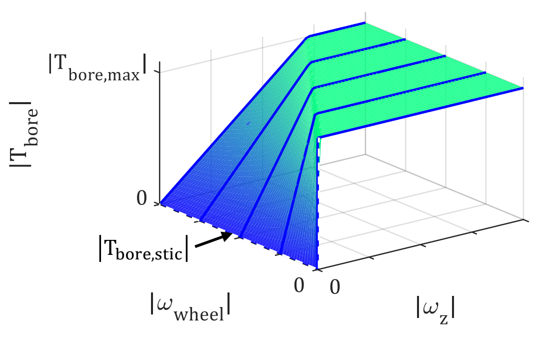


Figure 4. Bore friction characteristic (Zimmer and Otter, 2010).

In a first modeling approach equation 4 has been implemented by a regularized s-shaped characteristic as it is used in the `SlipBasedWheelJoint`. However, simulations revealed that a vehicle at standstill actuated by a constant driving torque that is considerably smaller than the expected break-off torque $|T_{bore,stic}|$ at the single caster wheels, would still start moving over the course of a few seconds. This undesired effect can be explained by using a continuous function to avoid the discontinuity which requires $|\omega_z| > 0$ rad/s when $|T_{bore}| > 0$. Even though the s-shaped characteristic can be tuned such that ω_z is comparably small when reaching $|T_{bore}| = T_{bore,max}$, still it is just a matter of time until the growing share of the acting forces pointing in the longitudinal direction of the caster wheels accelerate ω_{wheel} which leads to rapidly decreasing bore friction.

In order to properly capture the locking behavior and despite loosing real time capabilites by introducing events, the model equation 4 was implemented with help of the hybrid friction formulation used in the MSL `BearingFriction` model. However, it was found that this second approach is still not sufficient. Due to the fact that the friction characteristic (cf. figure 4) drops quickly from $T_{bore,max}$ to zero for $\omega_{wheel} \neq 0$, very small deviations of $\Delta \approx 10^{-3}$ rad/s allow the wheel to brake free. Hence, the behavior of the AS DevKit cannot be replicated with the bore friction characteristic introduced in equation 4.

In order to avoid the undesired effect described above, an alternative approach is presented in the following. In contrast to the previous model the new friction characteristic, shown in figure 5, has a linear slope w.r.t. ω_{wheel} near zero, such that small deviations do not take effect. This leads to the following formulation of the bore torque:

$$|T_{bore}| = \begin{cases} (|T_{bore,max}| - |T_{bore,stic}|) \cdot \dfrac{|\lambda_{bore}|}{\lambda_{bore,lim}} + |T_{bore,stic}| \\ \quad \text{if } |\omega_z| \cdot s_{rep} < \lambda_{bore,lim} \cdot |\omega_{wheel}| \cdot r \\ |T_{bore,max}| \quad \text{else} \end{cases} \tag{8}$$

with $T_{bore,stic}$ defined as

$$|T_{bore,stic}| = \max(0, |T_{bore,max}| - k_{stic} \cdot |\omega_{wheel}|). \tag{9}$$

For $k_{stic} \to \infty$, equation 8 tends to equation 4.

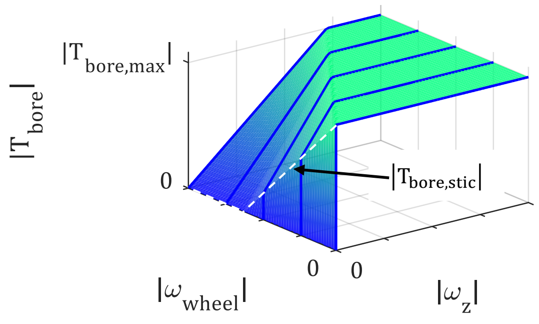


Figure 5. Alternative bore friction characteristic.

In order to implement equation 8, the new model `IdealWheelJointBore` combines concepts of the `IdealWheelJoint` (PML) and the `BearingFriction` (MSL) model. This allows to represent the locking condition explicitly as discrete state to assure that $T_{bore} = -T_z$ as long as $|T_z| < |T_{bore,stic}|$ (while $\omega_z = 0$). The `BearingFriction` component is extended with a `RealInput` to provide the current wheel contact force F_N. As the `Bearing Friction` component requires the exact torque T_z that is applied to the wheel, the wheel joint is additionally extended with a rotational `SpringDamper` component (MSL). This allows to dynamically resolve the distribution of the overall driving torque to the caster wheels despite a statically over-determined system when multiple caster wheels are connected to the same frame.

Calculation of the wheel contact forces. This subsystem provides a mathematical model for the calculation of the wheel contact forces. It is a simplified static approach that takes the current caster wheel orientations into account. However, the lateral dynamics of the AS, which have an influence on the wheel contact forces when accelerating or decelerating, are neglected due to rather small accelerations compared to gravity.

2.3 Verification of the ActiveShuttle Model

The verification of the AS simulation model focusses on the plausible replication of the observed behavior of the caster wheels and motion of the vehicle body, especially w.r.t. to the critical operation scenarios (cf. section 2.1). This is considerd sufficient in order to prove the path filter concept in Section 3.2. Therefore the model is not validated against measurements of the real system.

Turn on the spot. Figure 6 shows the simulation results for a desired clock-wise (CW) turn on the spot with two different masses. With $m_{AS} = 150$ kg, the AS is capable to initiate the turn and overcome $|T_{bore,stic}| = |T_{bore,max}|$ at all four caster wheels. However, at maximum payload (total mass $m_{AS} = 250$ kg), it is caught in a locking condition. Hence, ω_{AS} remains zero. Since $|T_{bore,max}|$ is remarkably higher in the fully loaded case, the drives of the AS are not strong enough to overcome the stiction torques at the caster wheels.

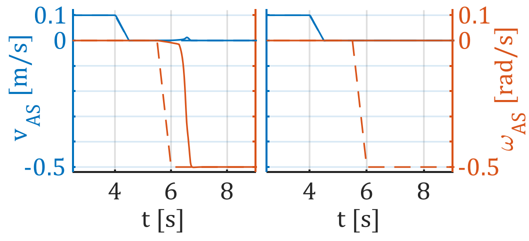


Figure 6. CW turn on the spot with $m_{AS} = 150$ kg (left) and $m_{AS} = 250$ kg (right). Dashed lines represent desired motion, solid lines the simulated motion. All caster wheels with $\mu_{bore} = 0.6$, $k_{stic} = 1$ and $\lambda_{bore,lim} = 1$.

Flipping caster wheels. Figure 7 shows the simulation results for a desired transition from driving straight forward to straight backwards. Due to the prior motion segment, all caster wheel orientations are 0 rad when $v_{AS,des}$ is reversed to -0.3 m/s. It can be noticed that the caster wheels are not changing their orientation by half a turn to π rad instantly. Instead the caster wheel's orientations start to flip randomly after about $t = 9$ s. This reflects the behavior to be observed at the real vehicle.

3 Design of the Path Filter

In this section a feed-forward controller is described that ensures that the AS continues its motion in the direction of the current caster wheel orientations before it is smoothly transferred in the direction of the desired orientations.

3.1 Control Architecture and Model

Figure 8 depicts the architecture of the path filter. Here, the index $(.)_C$ is representative for one of the four caster wheels. Moreover, ω_C is equivalent to $\omega_{C,wheel}$.

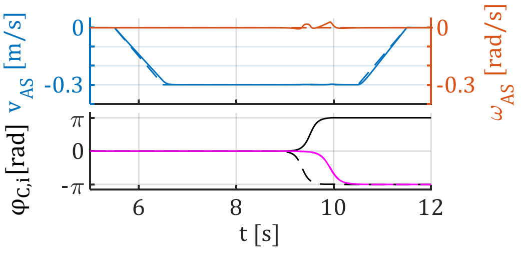
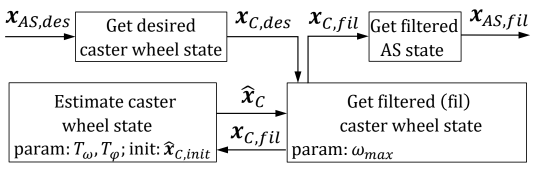


Figure 7. Random caster wheel flip with $m_{AS} = 250$ kg. Top: cf. figure 6. Bottom: Simulated caster wheel orientations. Black and magenta lines represent back and front caster wheels, resp. Dashed and solid lines represent right and left side, resp. All caster wheels with $\mu_{bore} = 0.6$, $k_{stic} = 1$ and $\lambda_{bore,lim} = 1$.



$x_{AS,des}$
Get desired caster wheel state
$x_{C,des}$
$x_{C,fil}$
Get filtered AS state
$x_{AS,fil}$
Estimate caster wheel state param: T_ω, T_φ; init: $\hat{x}_{C,init}$
$\hat{x}_C$
$x_{C,fil}$
Get filtered (fil) caster wheel state param: ω_{max}

Figure 8. Architecture of the path filter feed-forward controller

Get desired caster wheel state The path tracker of the motion control architecture commands the desired motion in the AS state representation. However, the path filter algorithm requires this information in the caster wheel state representation. Therefore, the geometric correlation between the two state representations is taken into account in order to provide a transformation. Hence, the subsystem requires the wheel radius and the position of the swivel axis of one representative caster wheel as parameters.

Estimate current caster wheel state We assume that all filtered states can be realized by the AS without encountering a locking condition or other external influences that keep it from reaching the commanded state. Consequently, it is taken for granted that the actual caster wheel state $\boldsymbol{x}_C$ is transferred into $\boldsymbol{x}_{C,fil}$ with a certain delay. Here, the delay is caused by the inertia of the AS. Hence, we calculate the estimate $\hat{\boldsymbol{x}}_C$ with two first order hold elements. The two time constants T_φ and T_ω can be tuned to match the AS dynamics.

Get filtered caster wheel state The actual filter algorithm calculates first the deviation between the desired and estimated caster wheel orientation. Equation 10 makes use of the modulo function in order to assure that $\Delta\varphi_C \in [-\pi;\pi]$ rad. This ensures that the AS takes the shortest path to its desired state.

$$\Delta\varphi_C = mod(\varphi_{C,des} - \hat{\varphi}_C + \pi, 2\pi) - \pi \tag{10}$$

Equation 11 states the calculation scheme for the filtered caster wheel orientation. Here, the factor k determines which proportion of $\Delta\varphi_C$ can be overcome in one calculation cycle.

$$\varphi_{C,fil} = \hat{\varphi}_C + k \cdot \Delta\varphi_C \tag{11}$$

According to Equation 12, k is defined to grow proportionally with the estimated caster wheel speed $\hat{\omega}_C$. Its definition is based on the idea that the faster a wheel is rolling, the easier it can be turned (cf. equation 4 and 5).

$$k = min(1, |\frac{\hat{\omega}_C}{\omega_{max}}|) \tag{12}$$

When the caster wheels are rolling very slowly ($\hat{\omega}_C \approx 0$ rad/s), the AS is forced to start moving in the direction of its current orientation $\hat{\varphi}_C$. Once the caster wheels have picked up some speed, their orientations change with $|\omega_{C,z}| > 0$. In other words, the path filter describes a trajectory in the bore friction characteristics of the caster wheels that avoids the peak friction torques (cf. figure 5). ω_{max} represents the slope of k. Small values reduce the time Δt that is necessary to transfer between estimated and desired state, but at the same time describe a trajectory in the bore friction characteristic that encounters higher bore torques. Hence, the choice of ω_{max} can be seen as a trade-off between transfer time and effort.

Get filtered AS state This subsystem represents the inverse of the first subsystem. It provides a transformation from the filtered caster wheel state to the filtered AS state. Here, it is important to provide the parameters of the caster wheel that was used with the first subsystem.

3.2 Path Filter Verification

In this section, we examine the path filter behavior with the help of two test setups. In both cases the path filter is set up with respect to the right front caster wheel (C,RF) and fed with a desired motion profile that includes the typical operation scenarios (cf. section 1.1). The "standalone" test case examines the input output behavior of the path filter, while the second "MIL" simulation includes the AS model as system plant and aims to show its impact on the physical behavior of the AS.

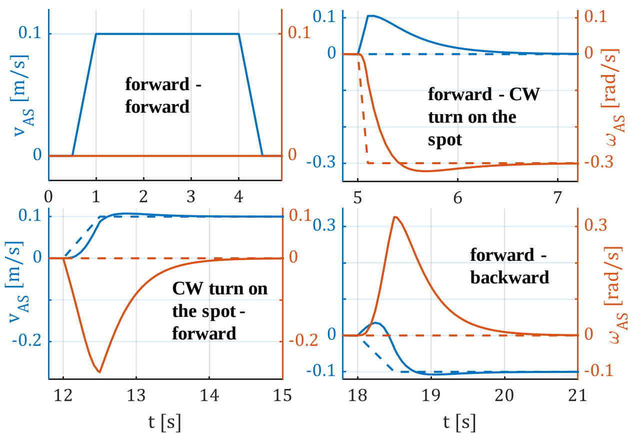


Figure 9. Standalone simulation of the path filter with ω_{max} = 100 rad/s and $T_\varphi = T_\omega = 0.01$ s. Dashed lines represent the desired motion, solid lines the filtered motion. For better scaling, the single motion scenarios are plotted separately.

The plots of the standalone simulation depicted in figure 9 show the path filter input ($\boldsymbol{x}_{AS,des}$) and output ($\boldsymbol{x}_{AS,fil}$). Here, the four motion scenarios are simulated in one sequence. Between each scenario, the AS is fully stopped. The plot on the top left reveals that $\boldsymbol{x}_{AS,des} = \boldsymbol{x}_{AS,fil}$ for the first motion segment. Here, the desired caster wheel orientation for driving straight forward is equal to the orientation that the estimation subsystem was initialized with (cf. figure 8).

$$\varphi_{C,RF,des} = \hat{\varphi}_{C,RF,init} = 0 \tag{13}$$

Consequently, $\Delta\varphi_{C,RF}$ remains zero during the first motion scenario and the filtered state is set to the desired state (cf. equation 11 and 10). For all other motion segments, the behavior of the path filter can be nicely seen. The commanded state $x_{AS,fil}$ makes the AS continue its prior motion before it smoothly passes into the desired state. When changing from driving straight forward to backward (bottom right plot), an angular velocity component is consciously induced by the path filter and avoids the random caster wheel flip.

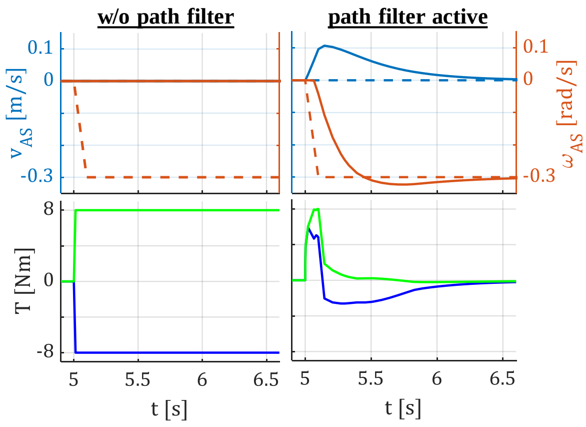


Figure 10. MIL simulation for a CW turn on the spot. The AS was driving straight forward prior to that. Top row: dashed lines represent the desired motion, solid lines the simulated motion of the AS model. Bottom row: green lines represent the left drive, blue lines the right drive. Path filter parameters: cf. figure 9. AS model parameters: $m_{AS} = 250$ kg, $T_{LD/RD,max} = \pm 8$ Nm, caster wheels as in figure 6.

Figure 10 shows the MIL simulation results for a CW turn on the spot. As motivated in Section 3, the path filter is able to avoid the locking condition that occurs in the setup without path filter.

4 FMI in ROS Control Architecture

In this section, we first describe relevant mechanisms and features of ROS, before we provide details on the FMI-Adapter package. Thereafter, we explain the integration of the Path Filter FMU with the ROS-based navigation architecture of the AS DevKit.

4.1 ROS Concepts

ROS uses a service-oriented architecture based on two common middleware mechanisms: publish-subscribe and request-response. Next, we briefly describe relevant ROS concepts:

Nodes. A software component is named node in ROS. Each node runs as a separate (Linux) process. Yet, a node may be instantiated multiple times, e.g., to run the same motor driver node twice for the two motors of a differential drive. To be able to distinguish two running node instances of the same executable, ROS provides a hierarchical naming scheme.

Topics. A topic is a typed and named n-to-m communication channel. Any node may open a *publisher* on a topic and *publish* (i.e. send) a *message* on it. This message is delivered to all nodes that have *subscribed* to that topic. The type of messages on a topic is defined by the first publisher. The `std_msgs` package of ROS provides message types for all primitive data types (i.e., bool, char, int, float, etc.). The packages `sensor_msgs` and `geometry_msgs` provide message types for common sensor data (e.g., laser scans, camera images, inertial measurements) and geometric primitives (e.g., points, poses, transformations), respectively. An interface definition language (IDL) allows to define application-specific messages types.

Services and actions. Using the same IDL and naming concept, services implement a typed request-response mechanism. Actions are a mechanism for long-running services, where the client may *preempt* the request.

Topics, services and actions are implemented with TCP/IP or UDP/IP. Therefore, the nodes can be distributed easily to different machines.

ROS master. The master is a dedicated process that provides a registration and lookup for nodes, topics, services and actions.

Parameter server. The parameter server provides a shared dictionary of typed key-value pairs, following the node naming scheme.

Launch files. A launch file is an XML-based specification to start a whole (sub-)system consisting of multiple nodes with corresponding parameterization. The specification language also allows to rename topics and services to connect nodes that have been developed independently.

Time and clock. ROS represents time by two 32 bit values (seconds and nanoseconds) since the epoch. In normal operation, the computer's clock is used as time source. Yet, ROS also allows a simulated clock with varying rate.

Packages. They are used to logically organize the software in ROS. A package may contain one or more nodes, a third-party library, a set of message types, launch files, etc. A package may specify dependencies to other packages, which are used for the build and installation process.

Callbacks and spin thread. Inside a node, each subscription, service server, and action server is associated with a callback function. Incoming messages are pushed to a first-in-first-out queue, which is processed sequentially by the spin thread by calling the corresponding callback function with the message data. ROS also allows to multiple callback queues and spin threads inside a node.

Timers. In addition to these communication-related events, a node may define timers to invoke certain functions periodically through the spin-thread mechanisms.

4.2 FMI-Adapter for ROS

The new fmi_adapter is a ROS package implemented in C++ for wrapping co-simulation FMUs according to the FMI standard 2.0 into ROS nodes. The package documentation is provided at `wiki.ros.org/fmi_adapter` and the source code can be downloaded at `github.com/boschresearch/fmi_adapter/`. An early version for ROS 2 can be found at `github.com/boschresearch/fmi_adapter_ros2/`.

The fmi_adapter package aims at providing the most important functions of the FMI 2.0 Co-Simulation interface (Modelica Association Project "FMI", 2014) mapped to ROS concepts/types as depicted in the following table:

FMI	ROS
input variable	subscription
output variable	publisher
state variable	*no explicit counterpart*
parameter initialization	parameter server
simulation time	ROS clock - offset
communication step-size	timer

It is intended neither to implement the whole FMI 2.0 interface nor to provide the rich set of introspection functions as for example FMI Library (JModelica.org, 2012) or FMI4cpp (SFI Offshore Mechatronics Research Centre, 2018). For advanced use-cases, ROS developers are referred to such libraries. Internally, fmi_adapter is based on the FMI Library, but the specific types are hidden from the developer.

The fmi_adapter package can be used, both, as a standalone ROS node and as a library.

Node use. The fmi_adapter package provides a ROS node, which takes the file path of an FMU as parameter `fmu_path` and creates subscribers and publishers with message type `std_msgs::Float64` for the input and output variables of the FMU, respectively. Next, it queries the ROS parameter server with the names of all variables and parameters of the FMU. For each name being found, the corresponding variable or rather parameter is initialized with the value being retrieved from the parameter server. Finally, the initialization mode of the FMU is exited and the node runs/simulates the FMU with a user-definable update period according to the ROS clock – until the node is shutdown. The following line gives an advanced example for invoking this node:

```
rosrun fmi_adapter node \
  _fmu_path:=./TransportDelay.fmu \
    _step_size:=0.001 _d:=0.5 \
    __name:=nodeB \
    /nodeB/u:=/nodeA/angle
```

In this example, the FMU implements a simple transport delay with real-valued input u, output y and delay parameter d.[1] In this invocation, the step-size is set to 1 ms, the delay is set to 0.5 s, the node's name is set to `nodeB` and the input x is connected to the topic `/nodeA/angle`. Hence, the values of `/nodeA/angle` will be published on `/nodeB/y` with a delay of 0.5 s, sampled at 1 kHz.

Library use. The fmi_adapter package also provides a shared library which gives much more control about the integration of an FMU in a ROS node. Most important, it allows to decompose complex ROS message types and to map the individual fields to the primitive-typed input variables of an FMU. Also, it enables the use of multiple FMUs inside a ROS node. Finally, it provides some basic functions to introspect a given FMU, e.g., to query the variable names depending on their causality.

For this purpose, the fmi_adapter library provides a C++ class `fmi_adapter::FMIAdapter`, which wraps a single FMU whose file path is passed as constructor argument (cf. Figure 11). On such an instance the default experiment step-size can be queried (by `getDefault-ExperimentStep`), the names of the input variables, output variables, and parameters can be retrieved (by `get-InputVariableNames`, etc.), and initial values can be set (by `setInitialValue` and `initializeFromROS-Parameters`).

The end of the initialization phase is marked using `exitInitializationMode`. Now, inputs can be set programmatically per variable using `setInputValue` and the FMU simulation can be advanced with two functions `doStep` and `doStepsUntil`. Output values can be retrieved with `getOutputValue`. For input values, the `FMIAdapter` class allows to pass timestamped values and thus even to specify a trajectory, where the user can decide whether the input values are interpolated linearly or considered as a step function. This feature facilitates to translate between different sampling/sensor rates.

Implementation details. Several subtle details had to be considered in the mapping between FMI and ROS concepts. The most important is the representation of time. ROS represents time in seconds and nanoseconds since the epoch whereas FMI uses a floating-point-based representation. The latter loses precision for large values. Also, an FMU may specify a specific start time, typically zero. Therefore, `exitInitializationMode` expects a ROS timestamp. This timestamp is used as offset between the ROS time and FMU time in all future function calls.

Another subtle difference is that FMI supports various characters in the variable names that are not allowed in parameter or topic names in ROS. We introduced a function `rosifyName` to replace these characters by underscores.

[1]ROS does not support physical unit specifications but assumes that all values are defined in the International System of Units (SI).

Finally, the FMU has to include a binary for Linux. The export of such FMUs is supported by various commercial and open-source modeling tools.

4.3 Integration of the Path Filter FMU

Drive commands are represented as `TwistStamped` messages (from the `geometry_msgs` package) in the ROS-based navigation architecture of the AS DevKit. The `twist.linear.x` field represents the lateral velocity and the `twist.angular.z` field the rotational velocity. This is a very common representation in ROS for velocity commands for differential wheeled robots.

In the navigation stack of the AS DevKit these messages are sent from the path tracker node to the engine driver node on a topic named `/velDes`. The former node implements a controller to follow the given path from the global path planner; the latter node translates the lateral and rotational velocity to motor commands for the left drive motor and right drive motor.

We integrated the Path Filter FMU in a new node named `PathFilter` between those two nodes using the fmi_adapter library. This new node consists of one function `main` only, with just 25 lines of code. On receiving a `TwistStamped` message on the topic `/velDes`, it feeds the Path Filter FMU with the values of `twist.linear.x` and `twist.angular.z` using `setInputValue` and runs the FMU up to the current time. Then, it reads the resulting output values from the FMU, creates a new `TwistStamped` message and publishes it on a new topic `/velDesFil`.

To integrate the `PathFilter` node in the existing architecture, only two lines in the corresponding ROS launch file had to be changed: A new line for the `Path-Filter` node had to be added and the input topic for the engine driver node had to be changed to `/velDesFil`.

5 Application and Test

In this section, we describe the application of the ROS architecture with the newly integrated `PathFilter` node to an AS DevKit. In a series of field tests it was the motivation to gather data that supports the results of the MIL simulations in Section 3.2 and further verifies that the path filter reaches its objectives (cf. section 3).

5.1 Test Setup

In contrast to the adaption of the launch file that was described in Section 4.3, we operated the Active Shuttle DevKit in manual mode. Here, the `PathTracker` node is deactivated and the desired motion is published to the `/velDes` topic by a node that is interfaced with a joystick. The desired motion profile included the following segments:

- straight forward $\rightarrow$ stop $\rightarrow$ CW turn on the spot $\rightarrow$ stop $\rightarrow$ straight forward $\rightarrow$ stop $\rightarrow$ straight backwards $\rightarrow$ stop $\rightarrow$ straight forward

The profile was driven several times with different path filter parameter variations including one reference case with-

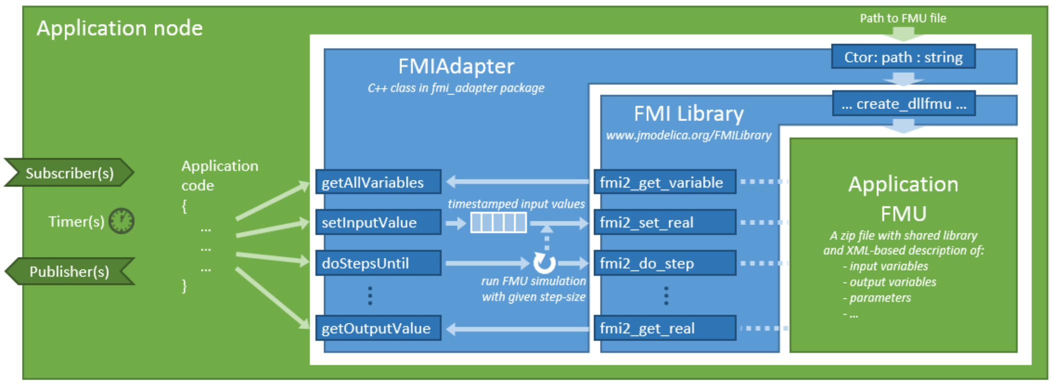


Figure 11. Architecture diagram from fmi_adapter package, illustrating library use

out path filter. The AS DevKit was loaded with 150 kg which sums up to an overall weight of $m_{AS} \approx 200$ kg. The relevant topics `/velDes` , `/velDesFil` and `/engine-data` were recorded with the `rosbag` tool. The latter topic holds the messages with the measured motor speeds and currents. The measured AS state $\boldsymbol{x}_{AS,m}$ results from $n_{LD,m}$, $n_{RD,m}$ and the AS kinematics. In order to verify that the path filter is solving the operational issue of randomly flipping caster wheels, $\boldsymbol{x}_{AS,m}$ is examined in combination with video recordings.

5.2 Test Results

Figure 12 shows a detailed view on the CW turn on the spot. Here, the measured motor currents of a reference case with no path filter are compared with a filtered case.

The time span where the motor currents of the reference case have reached a peak value of $\pm \approx 18$ A can be interpreted as the moment when the caster wheels are abruptly changing into their desired states $\boldsymbol{x}_{C,i,des}$. Here, the counteracting bore torque is abruptly vanishing and a significant drop in the motor currents can be observed. This drop results in an oscillation of the motor currents which lasts for a couple of seconds until it is damped.

The intended effect of the path filter can be nicely seen between $t = 15.4$ s and $t = 16$ s. The motor currents necessary to induce the turn on the spot are considerably reduced compared to the reference case. The path filter forces the AS to continue driving in the direction of the current caster wheel orientations first. Hence, the motor currents are first rising with a positive slope before the right motor current is dropping below zero. This accelerates the caster wheel speeds before the turn is initiated and reduces the bore friction. Moreover, we observe that the drop in the motor currents and the resulting oscillation are significantly mitigated.

Figure 13 shows a detailed view of the change from driving backward to forward. As it was elaborated in Section 3 the path filter intends to avoid the random caster wheel flip. We observe that the path filter forces the AS to keep driving backwards for a split of a second before it consciously induces an angular velocity component which triggers the orientation change of the caster wheels. Even

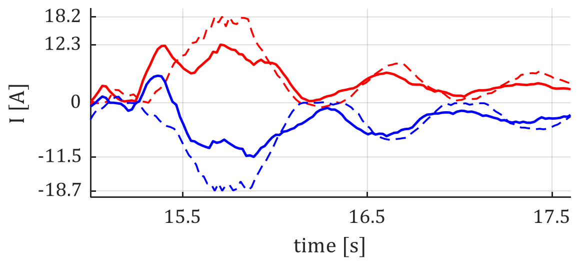


Figure 12. Measured motor currents for initiating a CW turn on the spot. The AS ($m_{AS} \approx 200$ kg) was driving forward prior to that. Dashed lines represent the results w/o path filter, solid lines the results with $\omega_{max} = 100$ rad/s and $T_\varphi = T_\omega = 0.01$ s. Red and blue lines represent the left and right drive, resp.

though the actual caster wheel orientations can not be measured we assume that the strategy works out. $\boldsymbol{x}_{AS,m}$ is following $\boldsymbol{x}_{AS,fil}$ and $\boldsymbol{x}_{AS,m}$ is significantly different from zero between $t = 46$ s and $t = 49$ s. Our assumption was verified through video recordings which clearly show that the caster wheels are turning instantaneously after the motion is started.

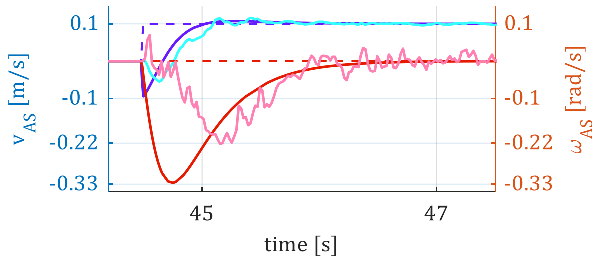


Figure 13. Desired, filtered and measured AS state for changing the driving direction. The AS was driving backwards prior to that. Dashed lines represent the desired motion, solid lines the filtered and measured motion. Path filter parameters and AS mass as in figure 12.

6 Conclusions and Outlook

In this paper a new model for wheels with bore friction has been presented. This model is suitable to describe the impact of bore stiction and allows to replicate critical locking conditions of differential drive vehicles with caster wheels. Based on the developed wheel model a vehicle model of the AS DevKit has been developed and used to design and validate the so-called *Path Filter*.

Using the new *FMI-Adapter* the PathFilter has been successfully integrated into a ROS control architecture and deployed to the AS DevKit. This way it has been demonstrated that FMI is a viable and attractive approach for an integrated end-to-end workflow from model-based control design to software integration for robotic applications and service oriented architectures. The open source *fmi_adapter* package enables users to wrap an FMU into a ROS node without deeper understanding of the FMI internals. The development process is drastically reduced with respect to time, effort and complexity.

The *Path Filter* has a significant positive impact on the handling qualities of the AS. It significantly reduces the effort necessary to perform movements that require the caster wheels to be turned on the spot. Moreover, it damps oscillations in the motor currents caused by the abrupt release of the counteracting bore torque. The path filter reduces the jerk which increases the durability of hardware components, eases the handling of fragile goods and improves the stability of shaky loads. Most important, the risk of getting stuck in a lock condition is drastically reduced. The Path Filter has been designed as self-contained function that can be retrofitted into existing control architecture between motion planer and motion controller.

The demonstrated application uses a micro processor as target and the deployed software was not subject to certified development processes. In the future work it needs to be ensured that the code within an FMU satisfies requirements of safety critical software and is optimized for real time applications. The current efforts within the publicly funded European project EMPHYSIS (ITEA3, 2017), developing the FMI for embedded systems (eFMI) standard, is addressing these challenges.

The Path Filter could be improved with respect to calibration effort and ressource demand by reducing the number of tuning parameters and the number of states. The estimation of the caster wheel angle using two first order holds could be replaced by an estimator based on the measured velocities of the driven wheels.

The plant model of the SDV could be enhanced to consider the impact of the currently neglected inertial forces.

Further studies shall reveal which constraints could be incorprated by the motion planner to determine trajectories that are compliant with the orientation of the caster wheels, which would allow to dispense the Path Filter.

References

G. Bardaro, L. Bascetta, F. Casella, and M. Matteucci. Using Modelica for advanced Multi-Body modelling in 3D graphical robotic simulators. In *Proc. of the 12th Int'l Modelica Conference*, Prague, Czech Republic, May 2017.

H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): Part I/II. *IEEE Robotics Automation Magazine*, 13(2/3):99–110/108–117, Jun/Aug 2006.

S. Imlauer, C. Mühlbacher, G. Steinbauer, S. Gspandl, and M. Reip. Hierarchical Planning with Traffic Zones for a Team of Industrial Transport Robots. In *Proc. of 4th Workshop on Distributed and Multi-Agent Planning (DMAP)*, pages 57–65, London, UK, Jun 2016.

ITEA3. EMPHYSIS – Embedded systems with physical models in the production code software, 2017. Retrieved 13 Nov 2018 from `itea3.org/project/emphysis.html`.

JModelica.org. FMI Library, 2012. Retrieved 3 Jul 2018 from `jmodelica.org`.

Modelica Association Project "FMI". Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0, Jul 2014.

F. Pecora, H. Andreasson, M. Mansouri, and V. Petkov. A Loosely-Coupled Approach for Multi-Robot Coordination, Motion Planning and Control. In *Proc. of 28th ICAPS*, Delft, The Netherlands, Jun 2018.

M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Proc. of ICRA Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

G. Rill. *Simulation von Kraftfahrzeugen*. Vieweg Verlag, Regensburg, Germany, 2007.

SFI Offshore Mechatronics Research Centre. FMI4cpp, 2018. Retrieved 25 Oct 2018 from `github.com/SFI-Mechatronics/FMI4cpp/`.

S. Swaminathan. Modelica-ROS Bridge. Retrieved 14 Jan 2019 from `github.com/ModROS`.

The MathWorks. Robot Operating System (ROS) Support from Robotics System Toolbox. Retrieved 23 Oct 2018 from `www.mathworks.com`.

M. Thümmel, G. Looye, M. Kurze, M. Otter, and J. Bals. Nonlinear Inverse Models for Control. In G. Schmitz, editor, *Proc. of the 4th Int'l Modelica Conference*, pages 267–279, Hamburg, Germany, March 2005.

S. Traversaro, P. Ramadoss, and L. Tricerri. gazebo-fmi. Retrieved 14 Jan 2019 from `github.com/robotology/gazebo-fmi`.

D. Zimmer. A free Modelica library for planar mechanical multibody systems, 2014. Retrieved 23 Oct 2018 from `github.com/dzimmer/PlanarMechanics`.

D. Zimmer and M. Otter. *Real-time models for wheels and tyres in an object-oriented modelling framework.* Vehicle System Dynamics, 2010.