Topology-Based Vehicle Systems Modelling

by
Edward Kar-Yun Yam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical Engineering

Waterloo, Ontario, Canada, 2012

Abstract

The simulation tools that are used to model vehicle systems have not been advancing as quickly as the growth of research and technology surrounding the advancements of vehicle technology itself. A topological vehicle systems modelling package would use Modelica to take advantage of the flexibility and modularity of the language, the inherent multi-domain workspace and analytical accuracy of model equations. This package is defined through the use of SuperBlocks, a generalized model that allows the user to select and parameterize the appropriate sub-system directly within the workspace. This palette of SuperBlocks would be implemented within MapleSim6 to create MapleCar. This provides a customized balance between speed and accuracy after taking advantage of advanced graph-theoretic solutions methods used in MapleSim.

MapleCar provides several advantages to a user over conventional tools. The SuperBlocks would ease the required steps to model a full vehicle system by providing clear, simple connections to quickly get a simulation assembled. Next, each SuperBlock is represented by a model that contains a *replaceable* model, a Modelica function which allows its internal model to be changed through a user-friendly parameter selection. The combination of sub-systems accessible directly through a parameter allows a variety of vehicle systems to be easily assembled, as well as provide a container for future models to be shared and published.

A short demonstration of connecting these vehicle SuperBlocks from the MapleCar package is provided using MapleSim6. The generalized vehicle component palette provides a straight-forward, customizable drag-and-drop interface to assist in generating vehicle models for simulation. Conclusions and recommendations are provided at the end.

Acknowledgements

I would like to give my thanks to my supervisor Dr. Amir Khajepour for taking me on as his graduate student. Dr. Khajepour along with visiting professor Avesta Goodarzi and post-doctoral fellow Alireza Kasaiezadeh have used their combined expertise in the field of Automotive Research to create the initial proposition of this work which targets some of the issues with current software. This grand vision, along with continuous meetings and updates throughout the terms, helped guide the development of this new Modelica-based MapleSim package.

I would also like to extend my thanks to Professor John McPhee and his research group, alongside Orang Vahid and Chad Schmitke from MapleSoft for providing technical assistance and vehicle model reference material within MapleSim. Within Dr. Khajepour's group, I was first to work with both MapleSim and Modelica to specifically analyze vehicle dynamics, and they provided many helpful insights along the way.

Finally, I would like to thank my friends and family for continuously providing spiritual and emotional support throughout the years. My friends have been with me since freshmen year in undergrad here at the University of Waterloo. I respect each and every one of them, and all have begun a successful transition into real life. Laphroaig and Bowmore were always great accompaniment during early mornings and stressful nights on campus. My family has strived for academic excellence and I hope to make them proud.

Table of Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

A fundamental component of modern engineering involves the modelling and simulation of complex, real-world problems. This holds true for automotive research since vehicle systems are very expensive to test, in both time and money. Automotive software tools are used to aid in the modelling and simulation stages of a project to predict a working solution before the next phase of development is committed. However, these tools vary in modelling approaches, equation flexibility, ease of use and connectivity to other tools. The growth and development of some of these tools is difficult due to the proprietary nature of the software to retain its users. A new automotive modelling system will be proposed to provide several key advantages in the current repertoire of tools.

In the choice of any software, there is a trade-off between the available flexibility the software gives to the user and how easy or difficult it is to develop and maintain new vehicle models. On one end of the extreme, all of the dynamic equations involved in vehicle dynamics can be derived and solved using a differential algebraic solver. This allows for a maximum amount of flexibility with an infinite potential for model fidelity, but it would be impossible to maintain. On the other hand, software can use a static mathematical model that comprises of parameters and look-up tables to transform a generic vehicle into the desired model. This is essentially a drop-in solution that requires table selection and parameterization by the user, but the amount of flexibility is limited to the developers of the software. A middle ground exists between these two approaches that include a fast, simple modelling package to provide turn-key modular vehicle sub-systems, within an open, flexible physical modelling workspace.

These complex projects are beginning to break the traditional paradigms of classical mechanical engineering. Designing a large integrated system comprised of mechanical, electrical and software components into a singular system requires a mechatronics perspective. However, this is not only taxing on the capabilities of an engineer but requires the available simulation tools to be extremely flexible. It is not uncommon for a large mechatronics project to require a chain of software tools, using the results and analysis in one program as a component of input into the next program. This is especially true for simulations involving accurate powertrain models, or contains multiple levels of control within a model. The inclusion of alternative energy, or ‘green’, research only complicates the model workspace when many branches of science and engineering are diligently working to popularize better alternatives to the internal combustion engine [1][2]. Many complications arise when a researcher must involve switching between several software tools in both usability and functionality.

Proposed in this thesis is a unified vehicle simulation package that assists a user in generating vehicle models, as well as accurately simulate the models using MapleSoft’s Maple and MapleSim software packages. The implementation of this concept is completed through the use of Modelica within MapleSim, which provides a graphical user interface on top of an object-oriented programming structure that packages algebraic model equations as blocks in a workspace. The modelling environment revolves around creating complex physical systems comprised of basic physical equation blocks. These systems can then be saved and parameterized to resemble a new library of vehicle components. By utilizing advanced functionalities of

Modelica in conjunction with the advanced solver methods of MapleSim, a new modelling framework has been designed to allow a user to quickly create vehicle models through a programmatic, topological block diagram. This package, currently named MapleCar, provides the necessary tools to quickly assemble and deploy a vehicle model in an open and efficient modelling environment, which natively includes support to simulate alternative domains within a unified workspace. MapleCar consists of multiple sub-packages, each containing a working library of components for a particular subsystem of the vehicle. These components are then automatically accessible through parameters within SuperBlocks, a generalized, high-level block that easily drops into the workspace and connects to neighbouring SuperBlocks. With all these features in place, a new hybrid workspace of system-level modelling and component-level modelling is formed with clear, organized component libraries to promote the sharing and reusability of vehicle models.

Chapter 2 contains an overview on currently available simulation tools comparing the typical functionalities of the software, common applications and shortcomings of each. The concept of a topology-based vehicle systems modelling is introduced in Chapter 3 as a package that provides a combined workplace for flexible manipulation of model equations while maintaining a user-friendly graphical interface. Several conceptual features of modelling using topological framework are explained. In Chapter 4, technical details of the package, and the structure of the basic Vehicle pallet within Modelica are explained. A working prototype of the new design tool is demonstrated in Chapter 5. Finally, a short discussion regarding the future of this work as well as concluding statements is provided in Chapter 6.

Chapter 2

2. Literature Review

With any new Engineering project, knowing which tools are available, and knowing which tools to use is as important as defining the project itself. In addition to this, features that assist users in generating models for simulation must be balanced by the software's openness to freely edit and manipulate the model equations. Obtaining an appropriate vehicle model is the first step in any vehicle project, but the speed and efficiency of work put into the project is ultimately limited by the user's own understanding of the software.

2.1 Matlab and Simulink

Matlab began as an educational tool designed to teach mathematical concepts of linear algebra without the need for students to directly learn FORTRAN, an early programming language that handled numerical computation at the time. Researchers first adopted its use for control theory engineering, but it quickly spread to other areas of research.

Simulink is an additional tool that operates above the main Matlab environment. Instead of the console and text-based environment in Matlab, a visual modelling environment is presented with several block libraries. The visualization of mathematical functions is primarily used for signal processing and control theory, where various layers of filters and functions are easily combined and compiled. Since there is such tight integration with Matlab and C-code, connectivity to external libraries and real-time hardware is popular for testing hardware and software in the loop experiments.

2.1.1 How it Works

Matlab is analogous to a programming environment with many mathematical matrix tools and functions, appropriately named “Matrix Laboratory.” Much onus relies on the user to effectively use the provided tools for relevant analysis. As a large numerical computational engine, the value of all the calculations is only understood by the user; little effort is made by the software to extract meaning from the computed values. Typical analysis performed in the Matlab environment relies on an algorithmic procedure to pass numbers between various functions and blocks until a user’s solution is reached. The core of the software is based in the C programming language, with an extensive library of prepared functions available to the user. Matlab also contains its own scripting language in which users can program and execute large sequences of functions or instructions creating a very open and flexible calculation environment.

2.1.2 How it's Used

Matlab is not a model generation environment; its purpose is to perform many calculations as defined by the user. Simulink, with its extensive list of additional toolboxes, contains a SimMechanics package that provides basic tools to assemble multibody models, but it is not a popular tool. When applied to vehicle systems dynamics, model equations are either manually entered, or another tool is used to generate a model for simulation within Matlab. This is a basic example of a software tool chain; a vehicle model is typically constructed in an external modelling environment, for example from CarSim or MapleSim, and is compiled as an external block inside the Simulink graphical block environment. Matlab does not assist in generating and defining vehicle models.

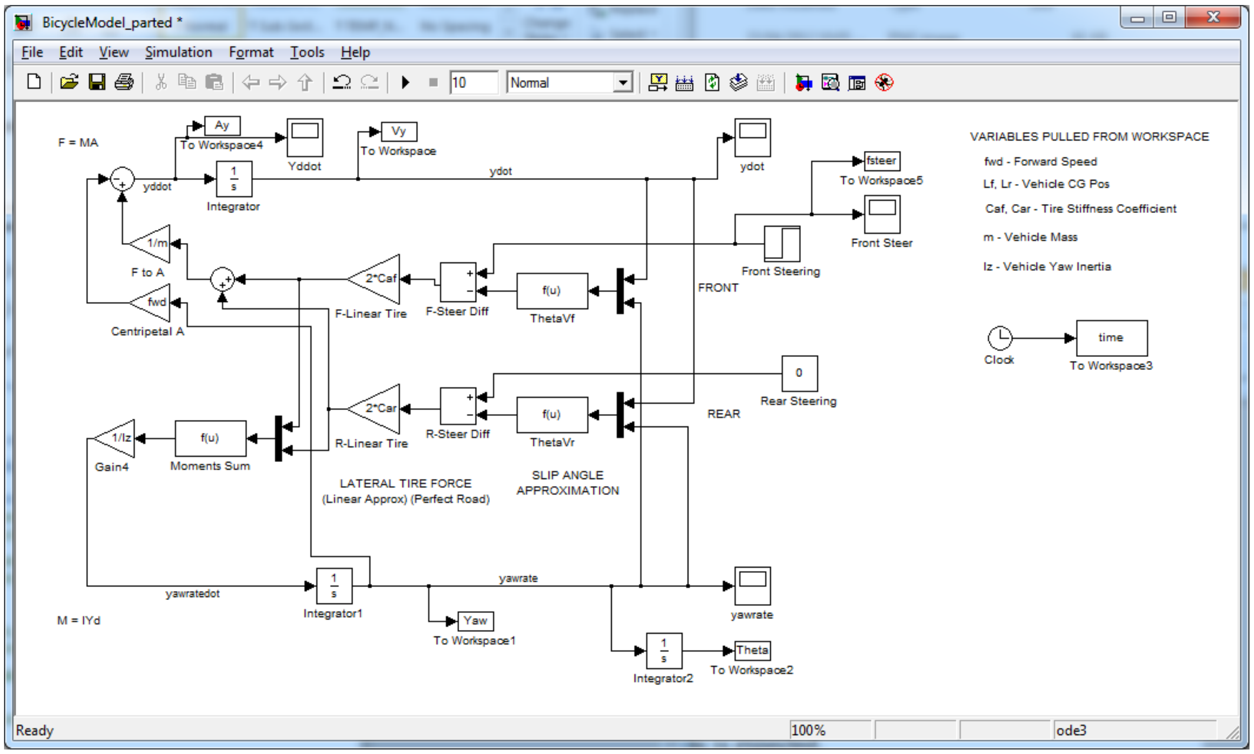


Figure 1 - Simulink model of a Bicycle Model with hand-derived equations

The flexible nature of the C programming language and extensive compatibility with external hardware makes it a preferred environment for hardware in the loop (HIL) or software in the loop (SIL) simulations. The library of toolboxes available in Matlab contain prepared algorithms and functions for easy testing of many high level concepts; from control engineering filters and controllers, to machine learning algorithms and optimization. As for presenting simulation results, 2D and 3D graphing tools are provided, as well as a spatial simulation environment through a Virtual Reality Modeling Language (VRML) geometry space.

2.2 MSC.Adams

MSC Adams is the "*most widely used multibody dynamics and motion analysis software in the world*" [3]. Developed by MSC.Software Corporation, Adams features a modular software environment to separately model, simulate and analyze complex dynamical problems. This software is used in all branches of engineering, and is used by 90 of the top 100 manufacturers in the world [4]. Accurate simulations are accompanied by a graphical modelling environment that can connect with computer-aided design (CAD) models.

2.2.1 How it Works

Introductory usage of MSC.Adams would involve three modules of the software suite; Adams/View, Adams/Solver and Adams/Postprocessor. Users interact with the Adams/View modeling environment to create multibody systems that emulate physical mechanisms. All bodies, masses, geometries and joints must be defined in their workspace before the model is simulated. Then, Adams/Solver uses classic analytical methods to generate the equations of motion for a multibody simulation. All the bodies, markers and joints contribute to parameters in a library equations based off the classical physical equations that describe the motion and force. Each body defined in the model automatically generates an equation for each degree of freedom in the body, while each joint or driver generates corresponding constraint equations. These are then simultaneously integrated while applying the joint constraints (depending on the solver selection) to generate the results. Optimization techniques to simplify the model equations occur after a large number of equations are created in the workspace, but large models can still result in an undesirable amount of computation time which complicates its usage in hardware and software in the loop simulations.

The basic Adams software does not provide too much assistance in generating models for specific applications; instead, separate modules can be purchased containing dedicated templates and toolsets for various industries. For complex machines and advanced kinematic and dynamic models, Adams/Machinery provides a library of components to use on top of the standard kinematic constraints, such as gears, belts and pulleys. Alternative domains of hydraulic, pneumatic and electrical analysis coupled with control theory engineering are available under Adams/Mechatronics and Adams/Controls packages to integrate multi-discipline, multi-domain components on top of the Mechanical domain presented in Adams/View. With respect to vehicle dynamics, MSC.Adams a rich collection of additional packages that target specific types of vehicle analysis. The Adams/Car package allows for the assisted generation of a full vehicle model or a specific subsystem in the Adams/View interface. Other products include prepared simulation environments for the chassis, driveline, suspension, tire, car ride and driver models to allow engineers to quickly and rigorously create and test their models.

2.2.2 How it's Used

The workflow of Adams.Car hints at a basic topological structure to simulate a vehicle. Separate models of subsystems must be loaded and connected together to form a car. This is achieved through creating an assembly of models that are connected through communicators. These communicators equate and transfer the force and displacement between connected models. In addition to the physical sub-systems, a library of tire models is included to handle the complex forces that occur between the tire and the ground. The result of the vehicle assembly becomes a high-fidelity, mechanical vehicle model that accounts for the kinematic and dynamic behaviours of the suspension and chassis.

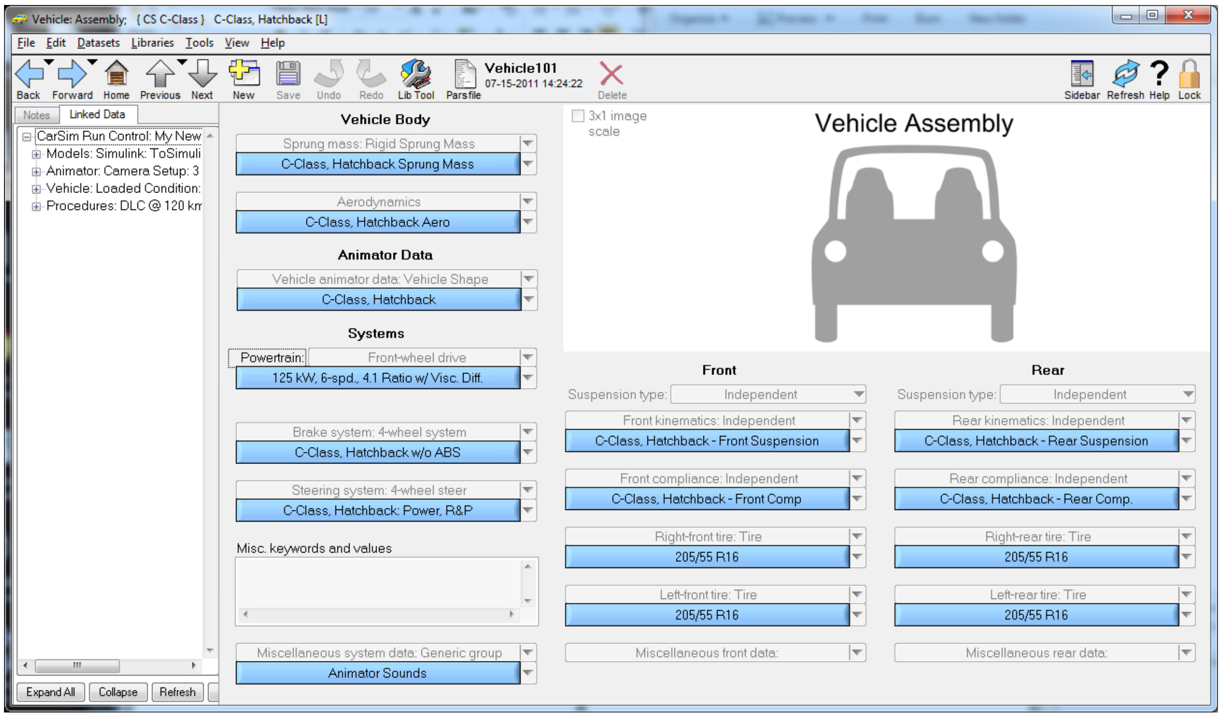
This is a great tool to examine mechanical sub-systems within a vehicle since the software is designed to accurately perform kinematic and dynamic analysis on multibody system, such as the suspension system. Common suspension types that share the same joints and linkages can be driven by Hard Points, a parameter set that contains all the physical measures of geometry and joint locations. This is useful for using Adams as a tool to generate the characteristic suspension curves of a suspension system and perform kinematic and compliance analysis. This information describes the subtle motions of translation and rotation when the suspension system undergoes a vertical input. This analysis is crucial when a reduced order model of a suspension is required [5]. The modularity of the system also provides a great research platform to test different suspension designs or create new suspension models [6].

2.3 CarSim

CarSim is a vehicle simulation package that "simulates the dynamic behavior of passenger cars, racecars, light trucks, and utility vehicles." [7]. It is accepted as standard software to use when any fast, drop-in vehicle model is needed. Within the CarSim interface, passenger vehicles can be simulated with a large library of driver models or standard maneuvers, parameter changes can be tested across any vehicle sub-system, and over 800 calculated variables can be plotted alongside automated visualizations. Parameters of the vehicle model are stored as large database entries as look-up tables or simple mathematical functions which allow very fast simulation times at the expense of enforcing a timestep and rounding errors. However, a large library of data sets for tire models, suspension sets, aerodynamic calculations, etc. collected and refined over years of use within industry provides a turn-key solution in quickly starting any vehicle simulation.

2.3.1 How it Works

The simulation philosophy behind this tool differs from most other software. The model predominately consists of look-up tables and simple mathematical curves to empirically emulate the various non-linear behaviours of a vehicle. Most notably, the characteristic suspension curves and tire models contain very complex, non-linear behaviour that is not trivial to simulate. There does not exist an all-encompassing tire model to accurately capture the dynamics of the tire undergoing various slip angles and loading conditions. Instead, the data obtained from thorough tire testing can be loaded as a look-up table such that the simulation can reflect this non-linear behaviour. The fine motion of a multi-body suspension model is similarly calculated. The response of any selected suspension model within CarSim is based on its kinematic characteristic curve. Since non-steerable suspension systems only contain one kinematic degree of freedom, every change in displacement and rotation is recorded offline and stored as a look-up table during simulation.



Vehicle: Assembly; { CS C-Class } C-Class, Hatchback [L]

Vehicle101
07-15-2011 14:24:22

CarSim Run Control: My New
Models: Simulink: ToSimuli
Animator: Camera Setup: 3
Vehicle: Loaded Condition:
Procedures: DLC @ 120 km

Vehicle Assembly

Vehicle Body

Sprung mass: Rigid Sprung Mass
C-Class, Hatchback Sprung Mass

Aerodynamics
C-Class, Hatchback Aero

Animator Data

Vehicle animator data: Vehicle Shape
C-Class, Hatchback

Systems

Powertrain: Front-wheel drive
125 kW, 6-spd., 4.1 Ratio w/ Visc. Diff.

Brake system: 4-wheel system
C-Class, Hatchback w/o ABS

Steering system: 4-wheel steer
C-Class, Hatchback: Power, R&P

Misc. keywords and values

Miscellaneous system data: Generic group
Animator Sounds

Front

Suspension type: Independent

Front kinematics: Independent
C-Class, Hatchback - Front Suspension

Front compliance: Independent
C-Class, Hatchback - Front Comp

Right-front tire: Tire
205/55 R16

Left-front tire: Tire
205/55 R16

Miscellaneous front data:

Rear

Suspension type: Independent

Rear kinematics: Independent
C-Class, Hatchback - Rear Suspension

Rear compliance: Independent
C-Class, Hatchback - Rear Comp.

Right-rear tire: Tire
205/55 R16

Left-rear tire: Tire
205/55 R16

Miscellaneous rear data:

Figure 2 - Look-up table and database selection for a CarSim vehicle model

2.3.2 How it's Used

There are numerous advantages to having the majority of mathematics simplified as a look-up table. As long as the vehicle model contains good data, many non-linear dynamics of a full vehicle model can be simulated in real-time. This is excellent when a full vehicle model is needed for controls and estimator design. When the CarSim model is imported into Matlab and Simulink, access to the internal variables and results are available so the vehicle can be monitored inside a control loop to evaluate the stability and performance.

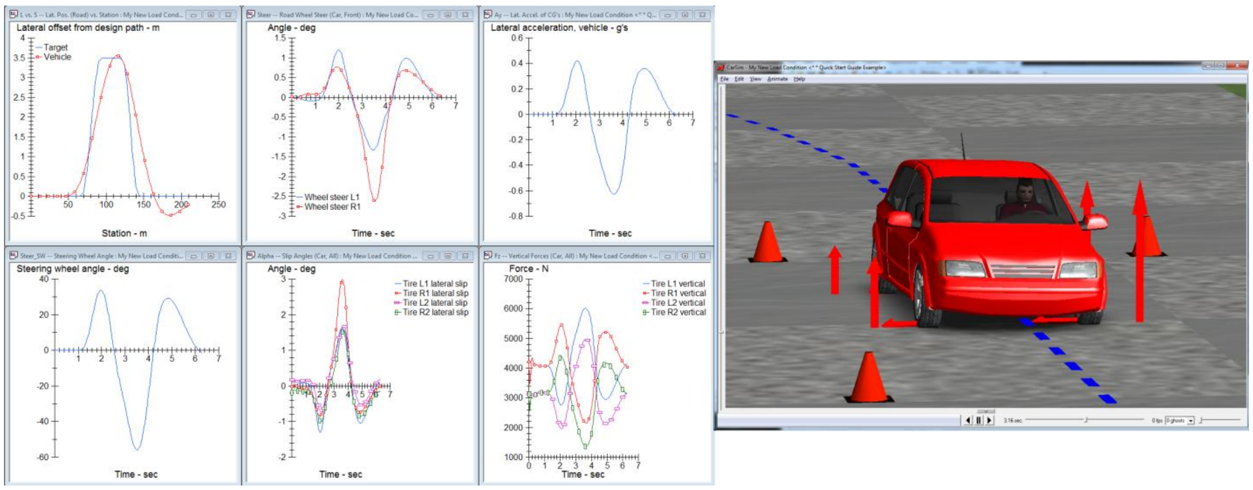


Figure 3 - Quick and Easy generation of CarSim results and animations

However, there are several clear disadvantages to the CarSim vehicle model that limits its use in research. Manipulation of the vehicle model equations is limited, and only the pre-defined list of variables can be output from the model. This makes the software infeasible to perform research that is geared towards particular sub-systems of vehicle. For example, the suspension of the vehicle only exists as that look-up table, so it is usually a better choice to utilize an open simulation environment to test and engineer various suspension components.

2.4 Dymola

Dymola, short for dynamic modelling laboratory, "is a complete tool for modeling and simulation of integrated and complex systems" [8] that utilizes the Modelica modelling language. Modelica is a "non-proprietary, object-oriented, equation based language to conveniently model complex physical systems" [9]. These two entities began from the same source, Dymola being the commercial entity to provide licensed software that best uses the publicly available language definition produced by the Modelica Association. Dymola has since been acquired under Dassault Systèmes to increase integration with the computer-aided design (CAD) software CATIA, while the Modelica Association continues to maintain development of the Modelica modelling language, holds conferences to those who use the software, and promotes the open nature of the language and library to users all over the world.

2.4.1 How it Works

The standard operation of Dymola revolves around two important tabs in the software, the modelling tab and the simulation tab. The modelling tab essentially contains an entire integrated development environment (IDE) to provide comprehensive tools to develop models using Modelica. An annotated screenshot of the workspace is shown in Figure 4. The left panel shows the package browser, which displays all the loaded libraries and packages available to the user, and the component browser, which displays the models and sub-models loaded into the currently working file. The top command bar has four key workspace panels, the Icon View, the Diagram View, the documentation tab and Modelica browser. Each of these panels corresponds to features in the Modelica code. The icon of a model contains a small visual image that represents the current model. The icon will also show any connectors in the model, which allows connections to other models. The diagram of the model contains the active drag-and-drop workspace; components from the package browser are dropped and connected into the diagram to assemble the physical model. The documentation tab displays information about the active model within a dedicated Hypertext Markup Language (HTML) annotation, and the Modelica browser displays the entire raw code of the selected model or package.

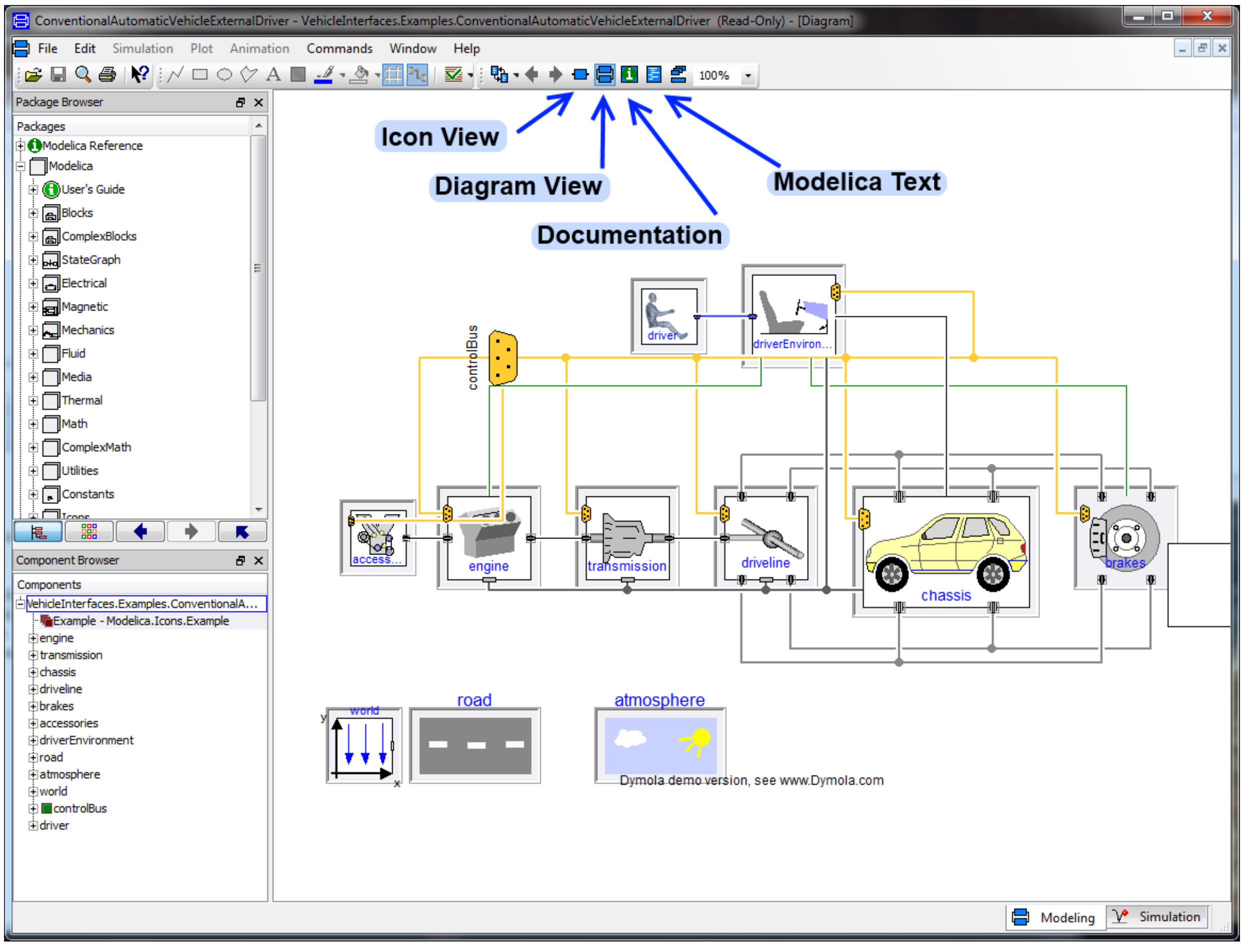


Figure 4 - Dymola Workspace

Modelling in Modelica is unique because it relies on constructing multibody circuits much like an electrical network. There are several notable differences when modelling in this manner. The software requires the kinematic topology of the mechanical system for simulation instead of creating connections through a visualized, geometric space. This simplifies the steps needed to generate a model but complicates the visualization of the model during assembly. The connections assigned between components represent equated variables instead of input-output signal connections. For comparison, the standard block view of Simulink the connection lines between blocks simulate a flow of signals put through various stages of function blocks, where the output is calculated based on the input in a unidirectional procedure. Instead, the Modelica modelling language connects the input and output variables in each block such that each block is accessing the same variable.

After the assembly of the model is completed in the modelling tab, all the simulation and post-processing functions reside within the simulation tab. The component equations within the entire model are automatically assembled into a large matrix to be used to solve and integrate the dynamic behaviour of the system. Several optimization techniques are used to reduce the number of equations the integrator must solve. After the simulation is complete, the only interaction the user must provide is based around the plotting and animation controls to visualize the data. All the system variables are available to the user after the simulation is complete, and 3D visualization can also be provided.

2.4.2 How it’s Used

Dymola provides the best integrated development environment to learn and develop raw Modelica code. The open-source nature of Modelica encourages the sharing of publicly available packages linked from their website [10]. However, similarly to the commercialization of Dymola, Modelon [11] provides professional, licensed toolboxes that encompass all aspects of vehicle simulation. The vehicle dynamics library includes an extensive library of predefined components and subsystems which can be assembled into any class of wheeled vehicle. Modularity of the models is inherent in the modelling language, which allows for the development of component level design, to isolate and test a particular subsystem, as well as system level design, to assemble and modify several subsystems in a higher topology. The largest downside of the software is the lack of popularity of Modelica itself in North America.

Even though large efforts of introduction and documentation surround Modelica, it is difficult for users to transition from existing modelling approaches to the physical modelling within Dymola. In addition, it is marketed within the suite of products available for CATIA, software typically marketed towards the highest level of industry.

2.5 Maple and MapleSim

Maple is an analytical computation engine to "analyze, explore, visualize, and solve mathematical problems" with an intuitive, clickable interface and document publishing system [12]. It is often wrongly compared to Matlab, which can usually perform the same functionalities but in an entirely different way. Matlab primarily operates on the manipulation of large, numerical matrices which enforces an inherent discretization of signals and results. Most operations performed by Maple utilize a symbolic computation engine that can maintain analytical accuracy through algebraic simplification before integration. MapleSim is a newer physical modelling software package that was developed as an add-on to the Maple symbolic computational engine. MapleSim features a block diagram modelling environment to automatically generate equations for complex, multi-domain systems. The solving method of MapleSim differs from the other software packages by using Graph Theoretic methods to automatically and efficiently combine all the component model equations before calculating the response. This allows the software to obtain a solution with higher fidelity and shorter calculation time, as well as allow the creation for deployable code for use in external simulation environments.

2.5.1 How it Works

MapleSim also utilizes Modelica to handle the diagram interface and icon view in the main workspace. The workspace is practically identical to that of Dymola, with the exception of how the model browser and parameters are displayed in different tabs. The package browser displays a modified version of the Modelica standard library with vendor specific changes hidden from the user. For multibody simulations in MapleSim, a proprietary library replaces the standard Modelica library. This MapleSoft multibody library uses graph-theoretic methods to quickly generate a minimum number of equations that describes the simulated model. This method works by first examining how each block is connected to one another, determining a set of fundamental tree or cut-set equations, and then inserts the constitutive equations of each component into the fundamental equation sets [13]. During regular usage, this is a completely automated process that occurs during simulation time that is not seen by the user. The minimum numbers of equations are efficiently and automatically generated to describe the full system without any interaction.

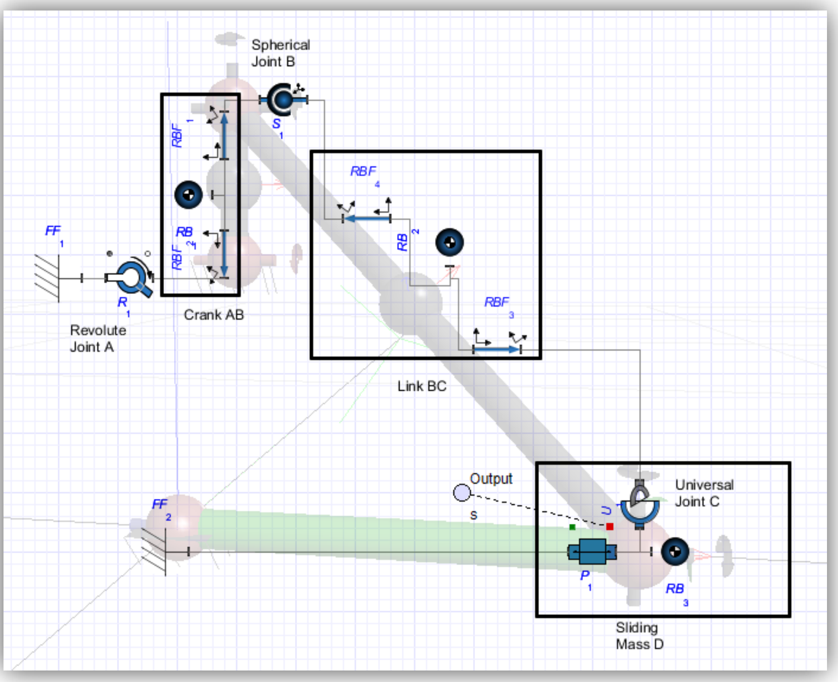


Figure 5 - MapleSim Block Workspace Representation of a 3DRigidSliderCrank

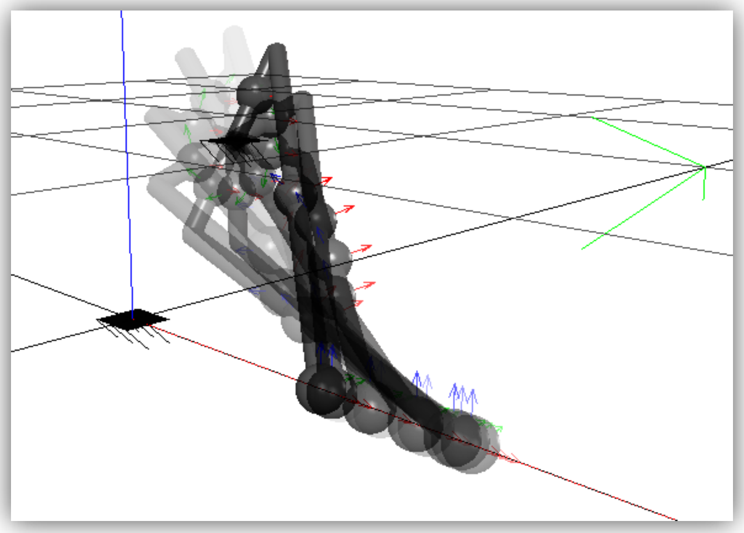


Figure 6 - Animation of the 3DRigidSliderCrank from MapleSim

2.5.2 How it's Used

MapleSim is generally marketed as a more entry-level simulation tool compared to Dymola. Much effort has been put into making the tool as simple and intuitive as possible, such that a new user does not require any insight into how the Modelica code functions. As an analysis tool, MapleSim features a deep connection with the Maple symbolic computation engine through its connectivity to Maple worksheets. This allows for the simplified model equations to be displayed to the user and further analyzed in Maple, for example, to perform sensitivity analysis on parameters or provide frequency domain analysis directly on the model. This extension of analytical mathematical tools does not exist in the other review software.

The popularity of MapleSim is still somewhat limited due to the issues of marketability. It is currently lacking a commercial package as comprehensive as the vehicle dynamics library available in Dymola, but contains several independent component libraries, such as Tire Component Library [14] and Driveline Component Library [15]. Even though MapleSim offers a fast and efficient simulation package, the lack of prepared simulations and required learning curve to begin using the software is discouraging for new users.

2.6 Software Discussion

The common software packages used in vehicle system dynamics are outlined. Even though each package plays a unique role in the field of vehicle systems dynamics, it is not feasible for all the tools to be used in parallel due to the numerous licensing costs and lengthy software tool chain that unnecessarily taxes a user. Different multibody tools use different approaches to create models. In some cases, these skills are not transferrable and contain users to particular software.

The software tool-chain currently required for alternative fuels research long and cumbersome, restricting the capabilities of a mechatronics engineer. It is clear there is an opportunity to develop a unified software platform that combines assisting a user to generate vehicle models while allowing various levels of customization.

2.6.1 Flexible Design Approach

For the Multi-Body solvers in the covered software, the approach to model and simulate the multi-body problem is shown in Figure 7. The geometric approach is followed in basic packages of MSC.Adams and MapleSim, where all the geometric information is used to construct a physical model. This model contains all the bodies of motion as well as joints and constraints that describe physical system. With the constructed model, the software can then automatically assemble and construct a system of equations essentially solving the motion of the system.

For a numerical solver, such as CarSim or Matlab, the approach to simulate the response of a system is dramatically different. Instead of constructing a geometric multi-body model, offline data is used instead which contains a look-up table of a determined force or displacement. This data is either obtained from empirical testing or a dedicated tool to generate curves of a known mechanism. The most notable examples of a numeric approach are the characteristic curves of a common suspension configuration, such as the camber, castor, toe-in, track-width changes for a double-wishbone suspension system, or the measured response of a tire undergoing various loading and slip angles.

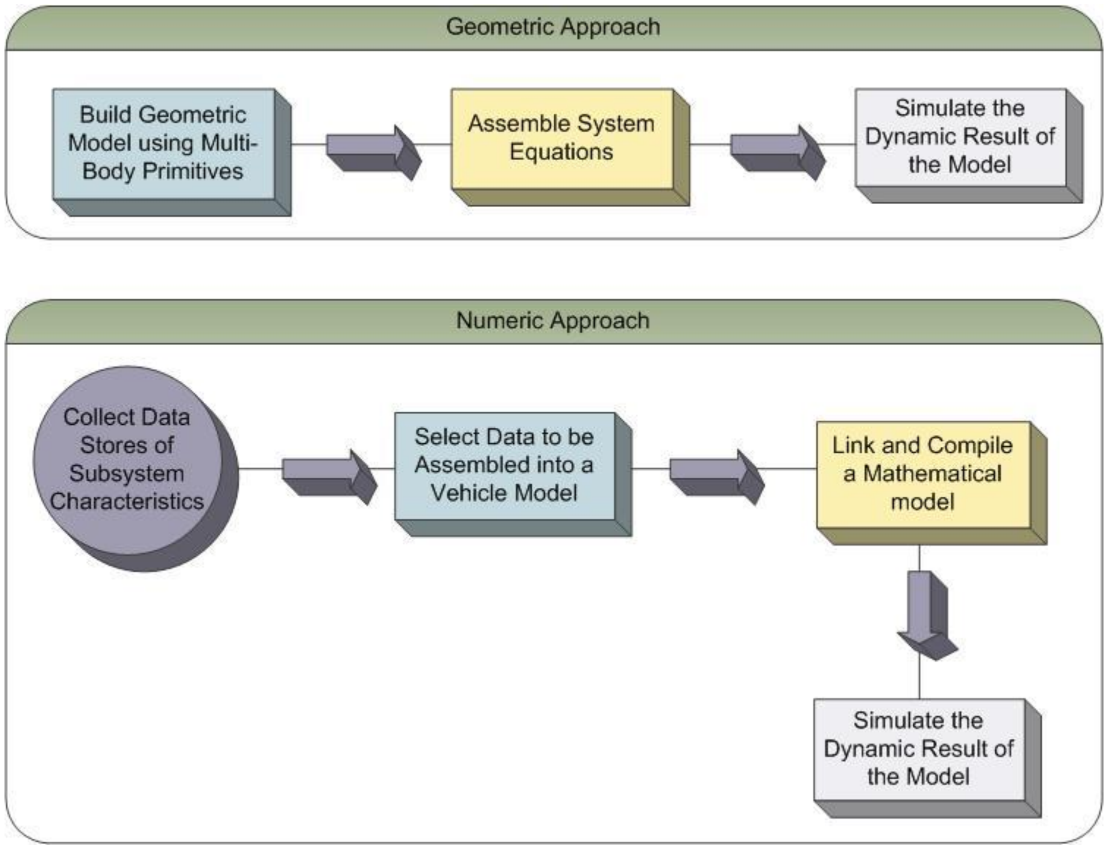

Geometric Approach
Build Geometric Model using Multi-Body Primitives
Assemble System Equations
Simulate the Dynamic Result of the Model
Numeric Approach
Collect Data Stores of Subsystem Characteristics
Select Data to be Assembled into a Vehicle Model
Link and Compile a Mathematical model
Simulate the Dynamic Result of the Model

Figure 7 - Two isolated simulation approaches

It would be preferred to have both of these options in a hybrid design approach. Geometric approach to simulate a geometric model contains all the necessary features to generate the characteristic curves of a suspension model. Instead of re-assembling and solving the same sub-set of suspension equations, the solution can be recorded offline to be used as a look-up table within another simulation. This hybrid approach outlined in Figure 8 would be an extremely desired feature in a simulation tool. The potential fluidity of properly re-using the results and conclusions will also assist in the overall engineering design cycle when any results require any change to the initial model to reiterate another design cycle.

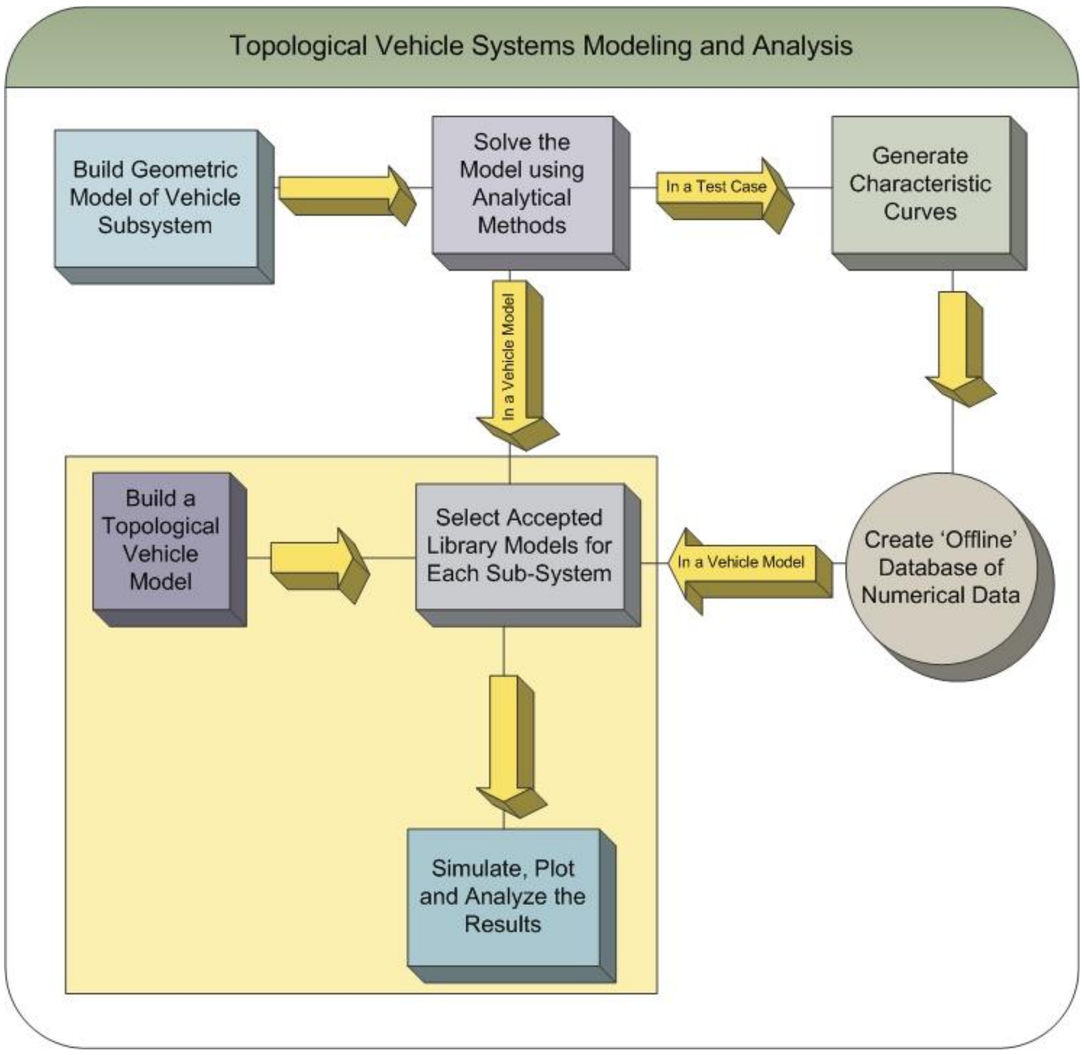

Topological Vehicle Systems Modeling and Analysis
Build Geometric Model of Vehicle Subsystem
Solve the Model using Analytical Methods
In a Test Case
Generate Characteristic Curves
In a Vehicle Model
Build a Topological Vehicle Model
Select Accepted Library Models for Each Sub-System
In a Vehicle Model
Create 'Offline' Database of Numerical Data
Simulate, Plot and Analyze the Results

Figure 8 - Hybrid Design approach

2.6.2 Encapsulated Multi-Domain Simulations

Green energy research involves a marriage of many different simulation domains, which requires a list of different software to be used. Hybrid Electric Vehicles (HEVs) involve utilizing a traditional internal combustion engine, electric motors, and power sources for each. These systems are typically connected in either series or parallel configurations [2] [16].

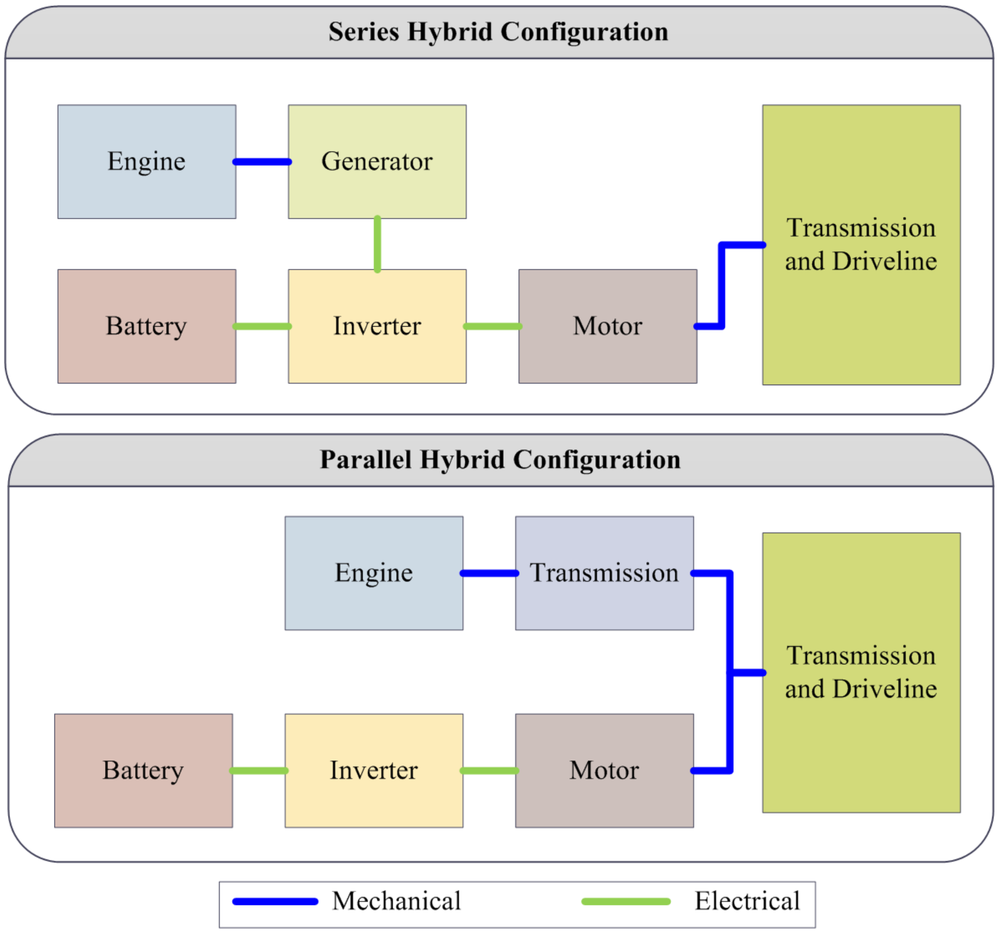

Series Hybrid Configuration
Engine
Generator
Transmission and Driveline
Battery
Inverter
Motor
Parallel Hybrid Configuration
Engine
Transmission
Transmission and Driveline
Battery
Inverter
Motor
Mechanical
Electrical

Figure 9 - Series and Parallel Hybrid Power Management Configurations

For a researcher to simulate the differences of these systems on a full vehicle model, it is clear that some sort of software tool chain would be necessary. MSC.Adams/Car or CarSim would be great tools to model and simulate the multibody dynamics of the vehicle, but these tools would require additional input to model the electrical domains of the system, through another package such as MSC.Adams/Mechatronics, or an external signal linked through Simulink. With numerous extensions and add-ons built on top of a main mechanical workspace, it may not be clear to other engineers how involved the other domains are towards the overall system when critical connections may be hidden in menus or contain text lists of signal connections. It would be advantageous for the workspace to visually reflect the entirety of all domains directly in the workspace to instantly convey the scope of the model.

2.6.3 Flexibility of the Mathematical Model

For controls and estimation design, the primary focus of the researcher is to work on designing, tuning and optimizing a controller to execute complex, high-level algorithms to better a specific operation of the vehicle. At the start of every project, the used model is a large simplification to capture the main physical response to ensure the controller is operating as intended. Gradually, the model increases in fidelity to include more behaviours and effects represented in the real world before testing the controller within a full vehicle model or with real hardware. It would be desirable to have better control when transitioning between more complex sub-systems to more easily test and troubleshoot the finished controller.

Looking at the mathematical model of CarSim, the vehicle model is essentially the assembly of a database of curves and look-up tables which restricts the manipulation of how the vehicle is assembled. In fact, the software is limited to simulate cars, which at most contains 2 axles and an optional trailer. MSC.Adams is fixed on the other end of the spectrum, where the mathematical model is automatically assembled through the connection of joints, linkages and bodies. Assembly of vehicular sub-systems is aided through various templates and features of MSC.Adams/Car, but is very particular on how the mechanism geometry is assembled with respect to the body of the vehicle. It would be desirable to have the software assist the construction and selection of sub-systems such that the full model can be easily generated.

2.6.4 Analytical Accuracy

When a model is assembled from various external modelling environments into a single simulation, the results and calculations each module must be discretized into very small time steps to communicate with one another. Balancing this time step becomes very important when the calculation speed is relevant to implementation. For example, executing very fast control loops on a moving vehicle requires the control, the vehicle model and calculation environment to communicate as close as possible. However, it is easy for calculation and rounding error to occur when the models are not effectively synchronized to one another; high frequency signals may get lost in translation or errors can compound and drift over time. Unifying the simulation environment that specializes in symbolic computation can potentially achieve calculations which are more accurate and more efficient when all the different models are constructed and simulated in the same workspace.

2.6.5 A Unified Platform for Research

This review above was intended to outline the common products and tools for engineers to use in their research, but in this work the research itself is to introduce a new approach in modelling and assembling virtual models. The table on the following page outlines how the reviewed software targets several key areas of concern with regards to develop a new vehicle dynamics simulation tool.

This selection of software was investigated because they each demonstrate at least one key feature cementing its place in industry. MSC.Adams has its beginning as one of the first, successful simulation tool to computationally assemble and solve large systems of multibody equations. Combined with a visual workspace and CAD geometry, it is one of the benchmark simulation tools to verify if other models are operating correctly. A major drawback that the software faces is the potentially large computation time involved in large simulations. CarSim is great to contrast to this measure since the mathematical model is mostly fixed. Both tools use look-up tables to calculate the tire-ground interactions, but the complexity of the suspension and other subsystems are essentially calculated offline as lookup tables resulting in extremely fast simulation times. This makes CarSim a great turn-key solution when a full-vehicle model is needed to quickly setup a complex simulation.

An area of concern for researchers using these tools will always remain the amount of allowed flexibility available within the software itself when new ideas are pursued. For a multi-axle vehicle, CarSim would not be adequate and requires entirely different software, TruckSim, to construct this vehicle. The number of model equation within MSC.Adams can explode without efficiently reducing the number of system equations which may result in simulation times becoming unreasonable.

It is clear that flexibility and modularity of Modelica would be greatly beneficial in the aspect of vehicle dynamics modelling. The concept of a topology-based design naturally mirrors the object-oriented programming style of Modelica, as well as provides a graphical framework in which the models can naturally be presented in drag-and-drop block diagrams. Dymola would be great choice to begin exploring vehicle dynamics with Modelica given the vehicle dynamics package maintained by Modelon, but the solver and equation generation would still be hidden if model equations were needed. Using MapleSim, the tight integration of the Maple symbolic engine provides an open-ended, analytical toolbox to be used an additional outlet when these models need to be analyzed. The freedom to insert and extract model equations, publish and share completed models and maintain the top level, multi-domain topology would provide a new, beneficial environment to research the next generation of vehicles.

	Matlab Simulink	**MSC.Adams**	**CarSim**	**Dymola**	**Maple and MapleSim**
Flexible Design Approach	Model design is entirely left to the user for equation entry. Behaviour of the model must be algorithmically defined.	Contains an open spatial modelling environment for constructing Multibody Mechanisms.	The design of the vehicle is comprised entirely of a database of look-up tables and user-defined curves.	Entire model design must abide by the Modelica modelling language.	Maple provides an open symbolic computation environment. MapleSim provides tools to help generate physical models.
Multi-Domain Simulations	All values are treated as real magnitudes; software does not provide insight into units and domains.	External plug-ins are available to enable simulations of other domains in-line with the Multibody model.	Does not natively handling a flexible multi-domain environment. The mathematical vehicle must be exported.	Supports all domains maintained by the latest Modelica specification.	Multi-domain simulations are inherently supported with the Modelica physical modelling language.
Flexibility in Vehicle Model Generation	All models must be generated offline and entered as equations, unless an external modelling tool is co-simulated in Simulink.	Through the use of communicators, Vehicle models can be assembled and customized with the provided vehicle templates.	The assembly of the mathematical vehicle model is pre-defined.	Contains a vehicle dynamics library maintained by Modelon.	Entire model must be generated from the Multibody Component palette. Drive-line and Tire component libraries are available.
Analytical Accuracy	Since data is manipulated in large matrices, all calculations are discretized according to the timestep.	The simulation is assembled to generate the analytical equations of motion through classical formulation.	Sub-systems are approximated as look-up tables and curves which trades simulation accuracy for a massive reduction in simulation time.	Maintains analytical equations for all components before integration.	The system maintains analytical accuracy for as long as possible before simplified equations are integrated.

Table 1 - Software Summary Chart

Chapter 3

3. Vehicle Modelling and Analysis Using a Topological Approach

After examining the current status of typical software used in vehicle system dynamics, the idea of using a topology-based approach to model and analyze vehicle dynamics was formed. The purpose of this design is to combine the publishing and packaging of vehicle components into an easy-to-use, drag-and-drop environment where models can be constructed and simulated with minimal effort.

An open modelling and development environment is required as a basis of constructing this modelling framework. As explained in Section 2.5. Modelica will be used to construct the hierarchical infrastructure and component referencing which will be presented in MapleSoft's MapleSim.

3.1 Top-Level Definition

Any complex, multi-domain vehicle system can be broken down into many simple components. All vehicles contain a body or a chassis to model the mass of the vehicle, and a suspension and tire system to maintain contact between the vehicle and the ground. As illustrated in Figure 10, if the major sub-systems of a vehicle were placed into a block workspace, each block will connect to a fixed list of other sub-systems. By preparing a set of models which are flexible, scalable and connectible in a consistent manner, a pallet of high-level components can be authored such that connecting a set of those blocks can easily assemble a typically vehicle system.

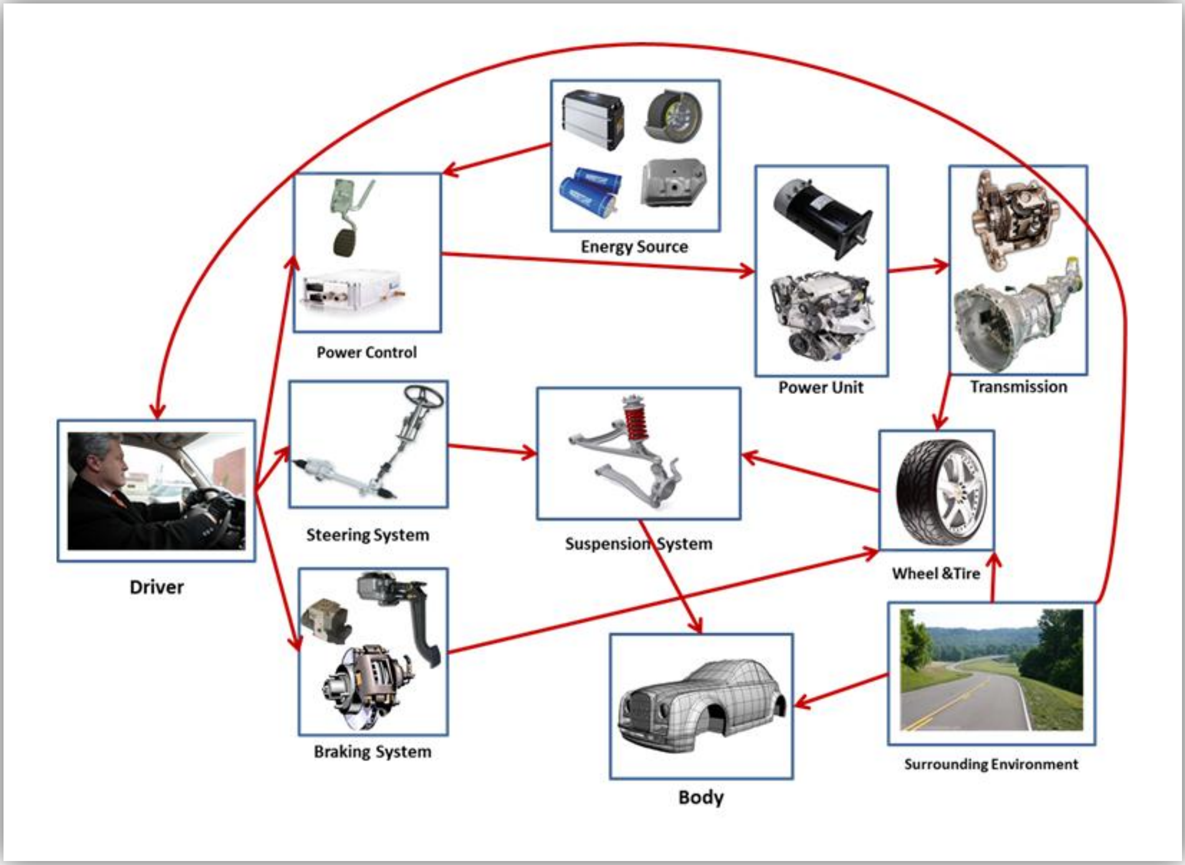


Figure 10 - Major Sub-Systems of a Vehicle [17]

To maintain stability of the vehicle to the ground, a suspension system is connected between the chassis and the tire. Whether the suspension is a simple, linear, spring-damper model or a full multi-body dynamic assembly, it must contain at least one multi-body connection to a wheel and one multi-body connection to the body. In turn, the tire must be connected to the end of the suspension component and interact with the environment. These are both represented by their own high-level blocks. Putting this together, a 4-wheel cart is represented as seen in Figure 11.

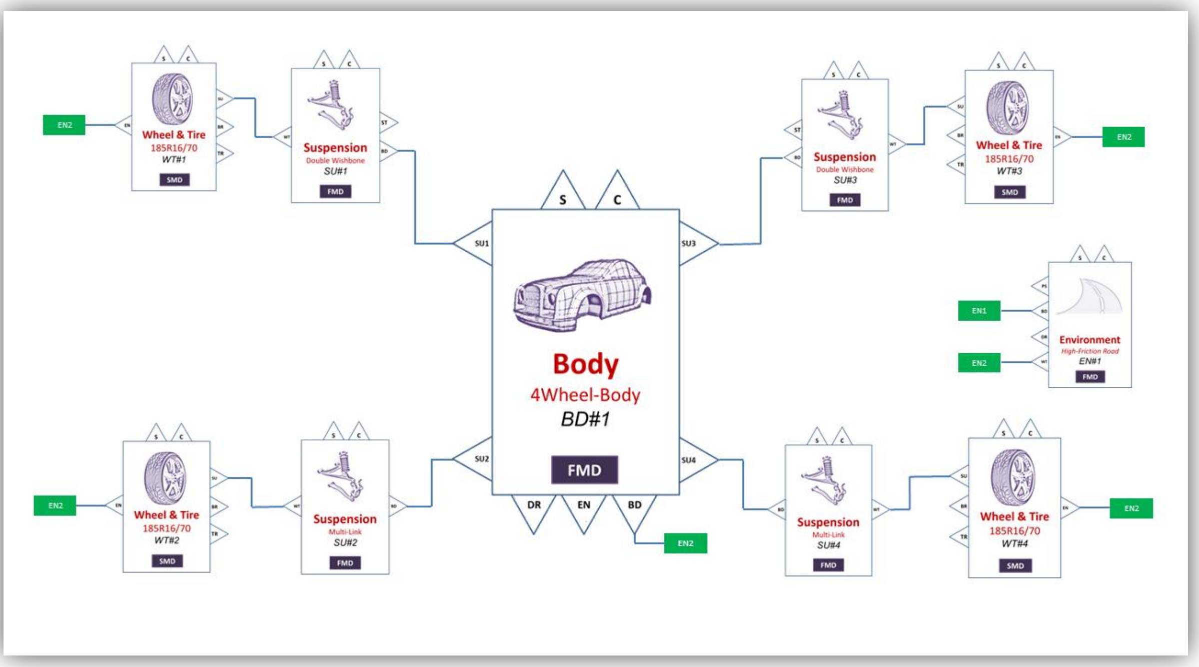


Figure 11 - Essential Sub-Systems of a 4-Wheel Cart [17]

For driven or piloted vehicles, two additional sub-systems are critical, a component that steers the vehicle and a component that powers the vehicle. The component to steer the vehicle is governed by a mechanism that changes the angle the wheel around an axis perpendicular to the ground. There are many mechanisms to obtain this motion, for example a rack and pinion controlling a tie rod, or a sector gear driven recirculating ball connected to a Pitman arm. In any case, the result of all these mechanisms remains the same; they provide precise control to the driver to steer the vehicle where the input to this sub-system is the steering wheel angle, and the output is connected to the suspension tie-rod. Since the essential function of this module remains the same, a single high-level block can be used.

For a vehicle that contains a power source, the benefits of a modular topological model become evident. In some way, shape or form, torque is generated using an energy source and is applied to the wheels. In a conventional, internal combustion engine vehicle, gasoline fuel is combusted in cylinders within an engine and torque is applied to the transmission, which mechanically transfers this power to the wheels. Note that most of this power train lies outside of the standard multi-body domain. Curve fit models, look-up tables or simplified equations are used to represent the operation of the engine, and the 1-dimensional rotation domain is used in the transmission block. These sub-systems can be contained in their own set of blocks to handle these calculations, which will naturally interact with the previously defined high-level blocks.

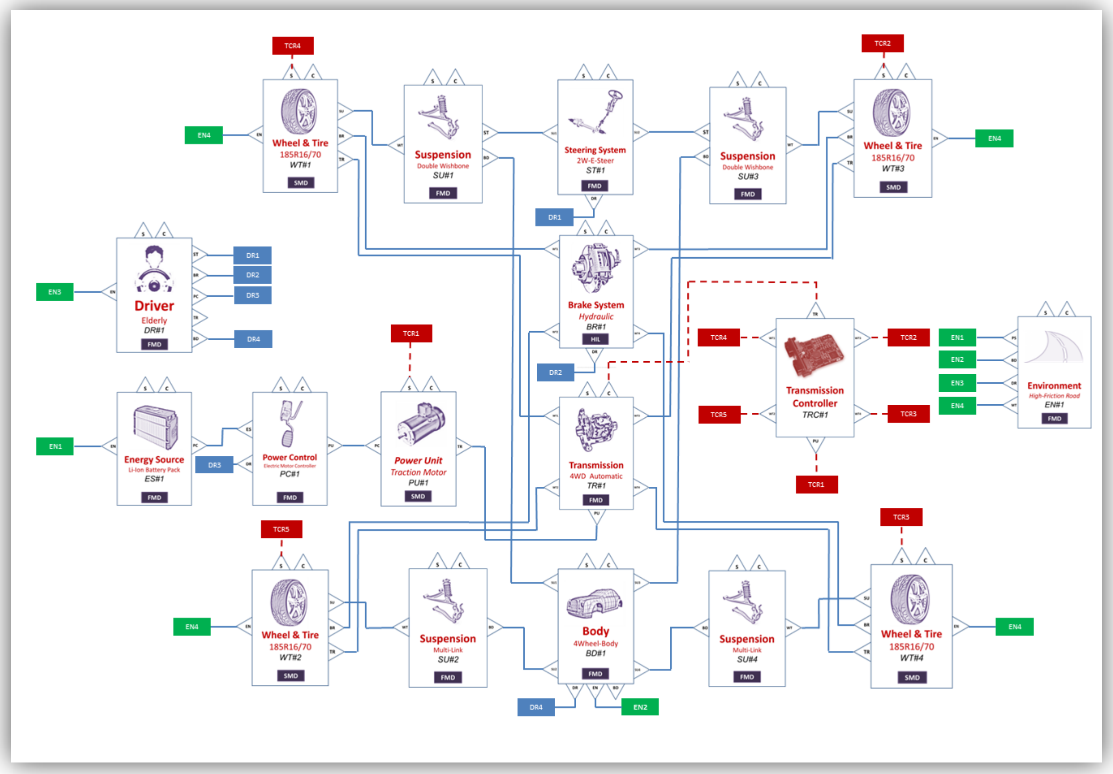


Figure 12 - Major Sub-Systems required modelling a basic electric vehicle [17]

Battery electric and hybrid electric vehicle research requires an extremely flexible design environment. On top of the multi-body vehicle platform, there may be multiple power sources that work in series or parallel to most efficiently generate and transmit torque to the wheels. A comprehensive study regarding different topologies of electric and hybrid electric vehicles outlines the most commonly used configurations [18]. These high-level blocks may contain a mix of an internal combustion engine, electric motors and electric generators, which may look something like Figure 12. The connecting lines represent the transfer of power between components. The workspace of a topology based design vehicle model should reflect these high level components in a similar manner. In addition, to provide greater robustness connecting compatible models and simplifying the user actions as much as possible, the library of compatible models should be directly accessible within the high-level block component, such that the topology of the overall system does not drastically change.

With these fundamental high-level components capturing a general definition of a vehicle system, this design approach allows researchers from other fields to quickly and easily setup and customize vehicle models for their own use. For example, a control engineer aiming to design an estimator to track the mass and inertia of a vehicle through a maneuver may require tweaking of a vehicle model to work, but does not want to stray too far into the world of multi-body dynamics. Instead of sourcing a vehicle model through another software tool, a vehicle model can be put together through an error-checked, drag-and-drop library of components. The fidelity and level of vehicle model will be selectable by changing the internal component equations for any top-level block providing seamless transition between running the control on a simplified full vehicle model, to a potential software-in-the-loop real-time test. The next stage of design iteration will also work very smoothly as version control of sub-system models can handle the refinement of the model.

By including articulation points on the chassis component, it is possible to further expand the possible simulations by chaining multiple vehicles together. A trailer would be modelled as another top-level chassis component that connects through a hitch to the lead vehicle. This configuration can be scaled up to become a tractor with a semi-trailer by expanding the number of axles defined on the chassis block and adding more suspension systems and tires to match a physical model. Because of the object-oriented nature of the workspace, multiple trailers can be instantiated to create models of Long-Chain Vehicles (LCVs) [19] [20]. With generalized, modular, top-level components, an inexperienced user will be able to connect complex models together for simulation and analysis. A theoretical extension of this idea can transform the tire-ground interactions with a tracked vehicle, which may lay the framework for a locomotive. The complexity and fidelity of the individual subsystems can be tailored to provide a range of accuracy depending on the goals of the researcher.

3.2 The SuperBlock

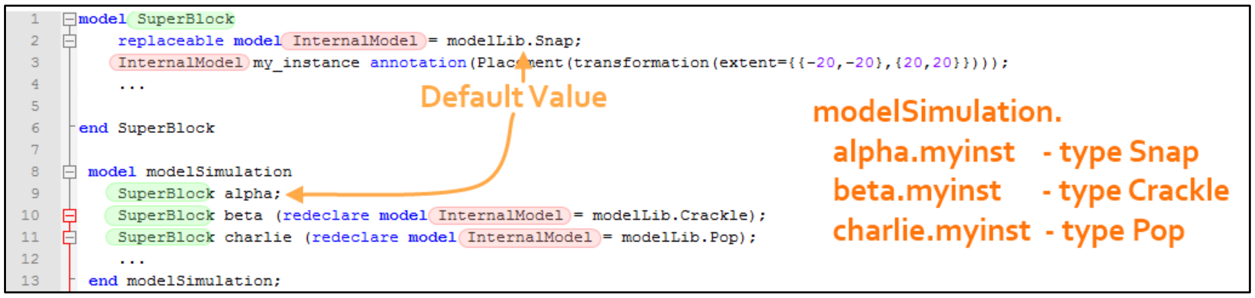
To realize the topology-based design workspace, the generalized high-level blocks must have a certain set of features. First, these blocks must allow the change of internal components with a simple interface. The selection presented to the user must always be compatible with all the current connections that exist on the high-level block. Next, the connection between these components should be automatically checked for compatibility. It is meaningless to cross models of different domains without a proper transducer. Finally, these top-level blocks must be scalable and customizable, such that the library of compatible components can be updated and published with minimal effort. As explained in Section 2.5, the Modelica physical modelling language can be used to include all these features. Each major sub-system will contain a corresponding package that contains the high-level model, named a "SuperBlock".

The object-oriented nature of Modelica as a programming language, combined with the graphical presentation of block icons and diagrams, provides the necessary tools to create these SuperBlocks. To ensure the compatibility of internal component equations, the library components must use a standardized '*interface*'. The technique of creating a standard set of interfaces to define a new package is outlined in the Modelica Standard Library [21] to maximize reusability of models and enforce basic standards. The suspension interface frame will always at least contain two multi-body connections, one to connect to the chassis and one to connect to the tire. All suspension systems must then extend this suspension interface frame. The selection of the sub-system components can be performed through a combination of two advanced Modelica techniques; the technique of "models contain replaceable models", and the *constrainedby*[1] modifier.

To represent a block that can access a library of compatible models within Modelica, a technique named '*Models with Replaceable Models*' can be used to achieve this effect. In the code sample below, the Top-Level class "Super Model" contains the definition of a *replaceable* model, named *InternalComponent*. Now, every instance of a particular type of

[1] The *constrainedby* keyword was introduced into the Modelica Specification version 3.0. This keyword was previously a vendor specific annotation parameter named *choicesAllMatching*.

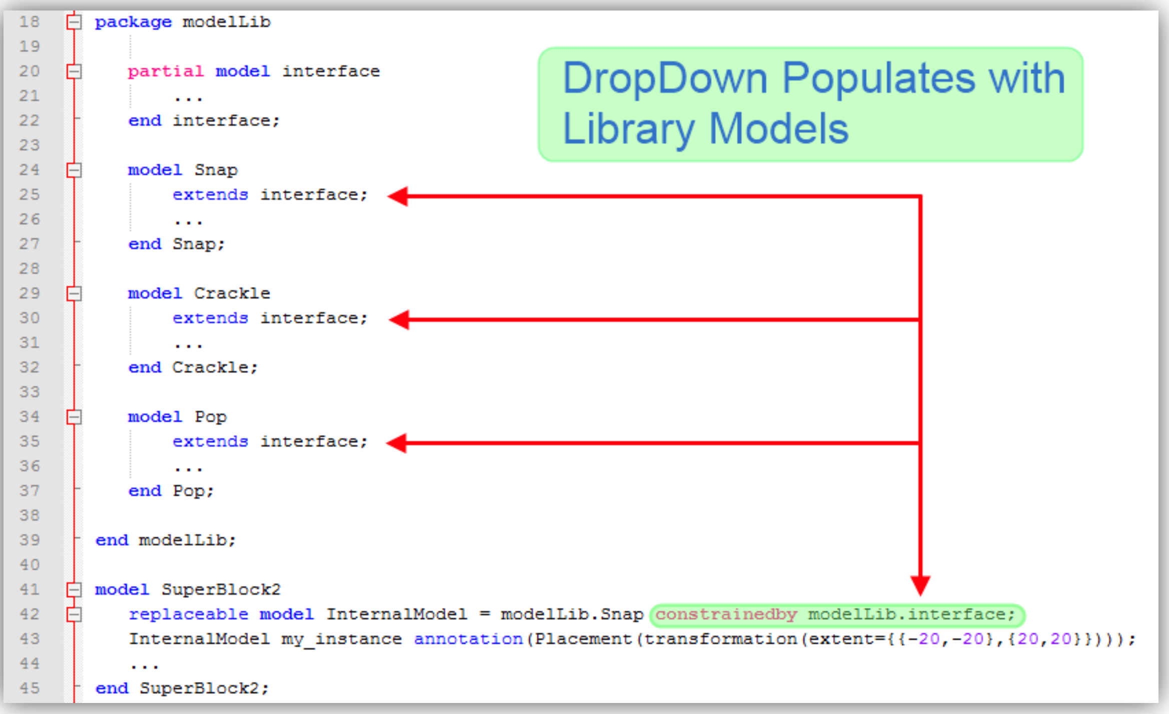
InternalComponent can be replaced through a parameter denoting which model will be used as demonstrated in Figure 13.



```
model SuperBlock
    replaceable model InternalModel = modelLib.Snap;
    InternalModel my_instance annotation(Placement(transformation(extent={{-20,-20},{20,20}})));
    ...

end SuperBlock

 model modelSimulation
    SuperBlock alpha;
    SuperBlock beta (redeclare model InternalModel = modelLib.Crackle);
    SuperBlock charlie (redeclare model InternalModel = modelLib.Pop);
    ...
 end modelSimulation;
```

Figure 13 - Code sample demonstrating a Model with a Replaceable Model

The drop-down menu for accessing the library of compatible models in the models with replaceable models property window can be accomplished in one of two ways. First, all the options, or *choices*, can be hard-coded into the annotation field as shown previously. All of the choices must contain the absolute location of the library model, which would require changes to the Modelica code for every new component added. The more desired approach for populating the drop-down menu is the Modelica keyword *constrainedby* which is used to automatically complete the drop-down list as shown in Figure 14. It works by examining all models in a separate package which contains the same *partial* model. For example, all of the suspension models in the internal suspension library must contain a "*suspension interface*" which is a partial model that only contains a multibody connector to the chassis and a multibody connector to the tire. When the *constrainedby* keyword is used in the Suspension SuperBlock pointing to this suspension interface, every model in the suspension library package will be checked to see if it also extends this "*suspension interface*”. If a library component does, it will automatically appear as a choice in the drop-down menu. Putting this together, once the Top-Level model is placed in the workspace, compatible sub-systems for that model can be displayed and selected as a property without the need to remove and replace the model.



```
package modelLib

    partial model interface
        ...
    end interface;

    model Snap
        extends interface;
        ...
    end Snap;

    model Crackle
        extends interface;
        ...
    end Crackle;

    model Pop
        extends interface;
        ...
    end Pop;

end modelLib;

model SuperBlock2
    replaceable model InternalModel = modelLib.Snap constrainedby modelLib.interface;
    InternalModel my_instance annotation(Placement(transformation(extent={{-20,-20},{20,20}})));
    ...
end SuperBlock2;
```

Figure 14 - Automatic drop-down population with *constrainedby* modifier

Automatic error checking of SuperBlock connections can be inherently performed by the Modelica programming environment through a customized definition of a new connector. A connector is a special model that creates a connection port to use within the component, to interact with the component equations and allow connections to other models outside of the block. This connection of variables must be consistent between models, since it passes a set of analytical variables that must be equated instead of a raw signal value.

3.3 Interaction of the SuperBlocks

Without any added features, the interaction of SuperBlocks is a natural extension of regular Modelica block assembly. The SuperBlocks are connected with regular connection lines in the diagram workspace and *Connect equations* are automatically generated. The method of assembling the equations before simulation is dependent on the specific Modelica Vendor, as they all have their own implementation of assembling, simplifying and optimizing the system equations. However, there are several features that are planned for the SuperBlock framework that will increase the functionality of connecting neighbouring SuperBlocks together.

Connection lines in a Modelica workspace are only designed to equate variables passed within connectors that are perfectly compatible. As stated earlier, connection lines between components represent a shared set of variables between the two blocks; a bi-directional connection that passes the same variable between both components. This behaviour inherently protects components from stray connections. For example, it doesn't make any sense for an electrical resistor to be able to connect to a rigid body mass, as the set of variables between both components do not match. It is possible to extend this behaviour to protect against stray connections between SuperBlock components. Even though it is technically acceptable to connect an Axle of the Chassis component with another axle of another Chassis component, it doesn't make any realistic sense and should never be allowed. This is achievable with customized connectors for each of the SuperBlocks to enforce valid connections. In addition, this allows for each connector to have its own icon and colour-code to intuitively instruct the user in the diagram workspace.

A downside to packaging large sub-system equations inside a SuperBlock structure is the restriction of access to internal components and variables. Unless frames are created on these components, there won't be an easy way to access the variables within the block. An example of this is the CG or Center of Gravity frame connected to the mass of the Chassis. Even though it is defined in the Chassis, a direct connection to that location isn’t always needed. When a user does require this connection, there would be no possible way to attach a probe or connect another multibody frame to this location unless the frame is embedded into the SuperBlock. *Expandable Connectors* are a feature that can also mitigate this disadvantage and allow this vehicle package to interface with various controls and estimation components. These special frames allow a user to connect to any number of variables within the component. The software package can automatically determine which variables exist within the block, or the user accurately specifies the sub-model and variable name. This variable selection is then used as the set of variables that must be equated within the connect equation. The usage of this feature greatly depends on a standardized and documented nomenclature for values and variables such that the names of the variables are consistent within the program and are understood by the user. Expandable connectors offer a highly-customizable option for controllers and estimators to read and access the variables from any number of SuperBlocks.

3.4 SuperBlock Palettes

The pallet of available SuperBlocks realizes the topology-based vehicle modelling concept in a presentable drag-and-drop package. To perform the initial tests of topology-based vehicle modelling, the basic definition of the Chassis and Suspension SuperBlocks are outlined below with all the desired features that are slated for implementation. A note regarding the SuperBlock template is provided at the end to propose how this structure will interact with existing component libraries already prepared by MapleSoft.

3.4.1 Chassis SuperBlock

The chassis SuperBlock is designed to represent a basic model of a generic *vehicle* body. Most simplified chassis models consist of a rigid body, representing the sprung mass, and offsets, providing the structural translations from the sprung mass to the axles, which is reflected in this component. The coordinate system used is shown in Figure 15. The number of axles for a vehicle body is scalable using a set of checkboxes, which changes the number of connections on the block. There are several other options to allow common additional features to the model without the need to break-open the model, such as front and rear articulation frames, the sensor and control frames, and an external multi-body connection to the center of mass. These options appear in the property window as check boxes which create or remove the appropriate connectors on the block.

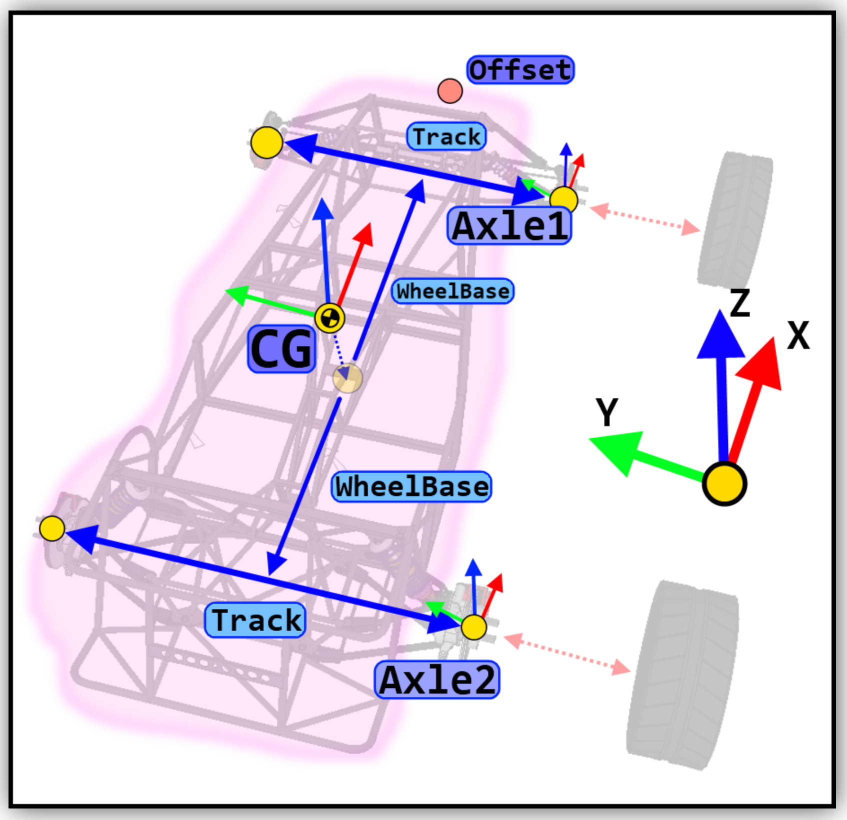


Figure 15 - Basic coordinates of a simple chassis

The Chassis SuperBlock is currently limited between 1 and 7 axles. The coordinate axis is assumed to be the ISO vehicle standard coordinate axis as shown in Figure 15. Defining the dimensions for each axle begins with a definition of an initialization point of the CG location within the global space. The other vehicle dimensions, such as the axles and the front and rear articulation points, are defined with respect to the CG location. When these user options are combined with a set of selectable icons, this model allows the chassis component to represent a wide variety of 'vehicles' or body components that are used in various vehicle simulations.

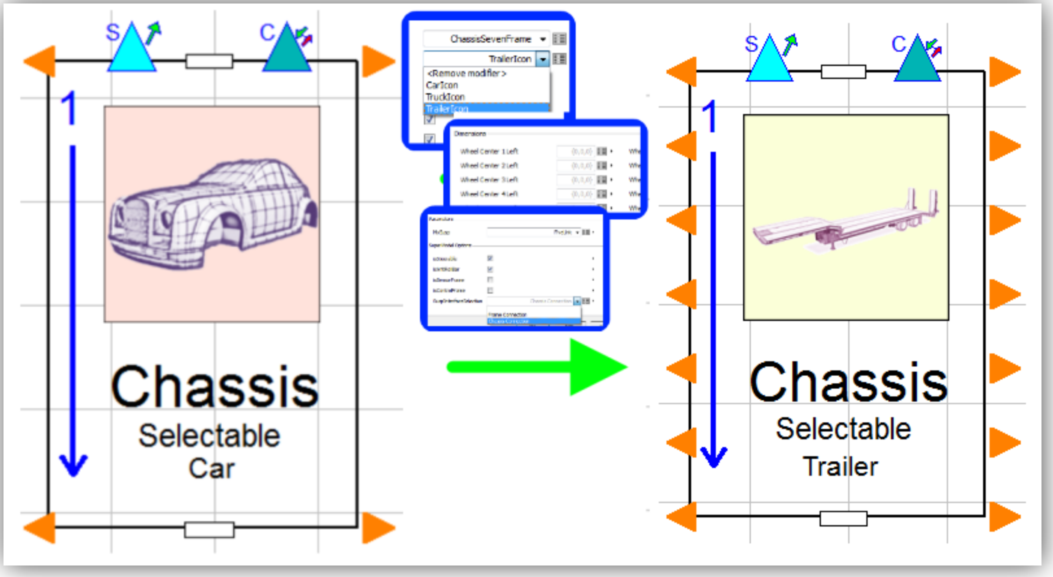


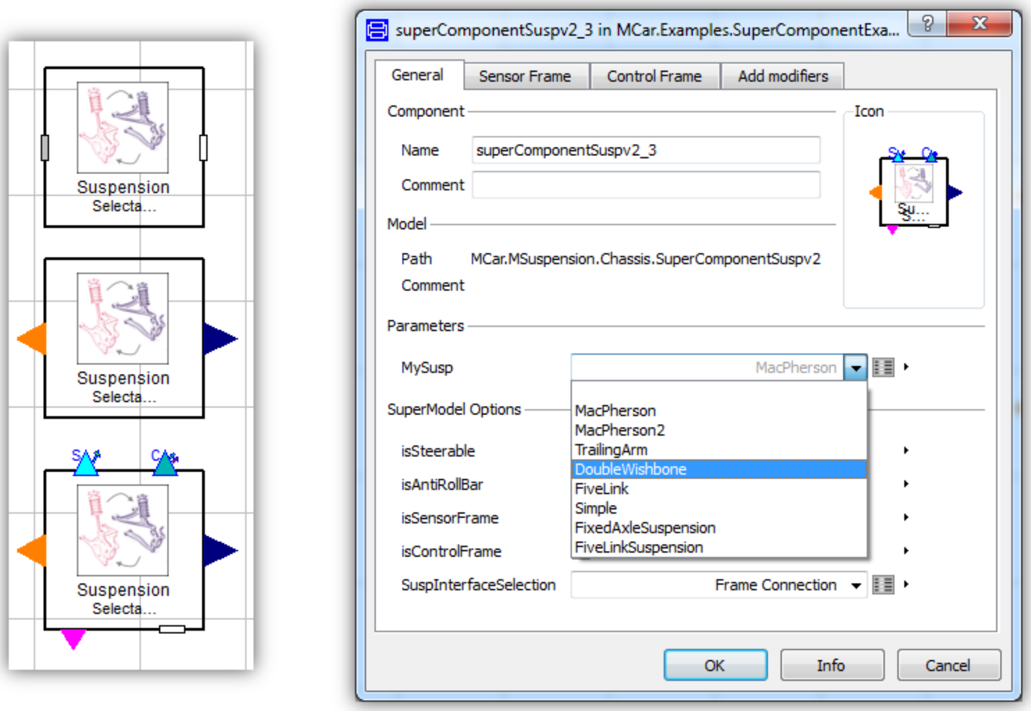
Figure 16 - A sample transformation of a chassis component into a trailer

In most simplified vehicle models, the sprung mass is typically represented by a single rigid body mass. All other internal components that connect to the rigid body mass has an appropriate dimension field included in the block properties. With simple images and concise labeling, the component property window clearly notifies the user regarding the purpose of each option. For more advanced chassis models, we can attach external components to represent other models. This allows a single chassis block the ability to represent most vehicle body components configurable through a user-friendly graphical interface.

3.4.2 Suspension SuperBlock

The suspension for a vehicle is a major area of research, and the Suspension SuperBlock must reflect this complexity. The suspension SuperBlock is largely based on the advanced Modelica programming technique of utilizing a replacement of internal components using the Models with

Replaceable Models. The SuperBlock has a *Suspension Interface* that contains connectors to a Chassis Component and a Tire component. By definition, all vehicle suspensions must at least have these connections. This is a standard Modelica multibody frame and is the constraining class for the *constrainedBy* replacement keyword modifier. Aside from that selection governing the selected internal component, there are several other options that toggle connection points for the Suspension block. These are the anti-roll bar connection, the Steering rod connection and the sensor and control connection points. Once a completed suspension model is in a working state, it can be published by placing the model into the appropriate library folder. This allows for very secure control over model versions when they are developed and used in separate areas of research.



superComponentSuspv2_3 in MCar.Examples.SuperComponentExa...

Component
Name: superComponentSuspv2_3
Comment:

Model
Path: MCar.MSuspension.Chassis.SuperComponentSuspv2
Comment:

Parameters
MySusp: MacPherson
- MacPherson
- MacPherson2
- TrailingArm
- DoubleWishbone
- FiveLink
- Simple
- FixedAxleSuspension
- FiveLinkSuspension

SuperModel Options
isSteerable
isAntiRollBar
isSensorFrame
isControlFrame
SuspInterfaceSelection: Frame Connection

Figure 17 - Parameters for a suspension SuperBlock as seen in Dymola

Since different suspension configurations have a differing number of parameters or hard-points, the cleanest way to assign parameters for these suspension systems is through a Record, as known in Modelica, or a Parameter Block, as known in MapleSim. After a particular internal suspension component is selected, the model property window contains an option to edit the properties of the instance of the internal component. In Dymola, there is an *edit* button beside the drop-down selection that opens the property window in the sub-model. In MapleSim, this is named the *Advanced Variable* drop-down option but is currently full of bugs. Note that in this sub-model properties window, the options to configure the model is depends on how the internal component was authored; it is up to the component to enable all the relevant and useful parameters in the property window with a compatible coordinate axis and nomenclature. Since different suspension systems of similar topologies only differ by the hard-point values, the technique of Models containing Replaceable Models can also be used on the Record, or the Parameter Block, so a particular parameter set can be published and stored inside the package. This would result in another level of drop-down selection to access a library of suspension specific data points.

3.4.3 Sensors, Controllers

The idea of integrating sensors, controllers and estimators in the topological workspace is not a trivial task. With standard Modelica connectors, all the connections must define their shared variables within the model prior to any other connections. Clearly, this would not be appropriate to leave up to the user since this requires changing the internal connections every time. There are two ways to solve this issue. Accessing parameters within components can be achieved through utilizing numerous sensor blocks to measure and obtain a signal connection. This may potentially clutter the workspace but will be extremely effective in constructing the desired outputs one variable at a time. Another potential way to achieve the same result in a cleaner fashion would be the usage of *expandable* connectors in Modelica. The keyword *expandable* modifies the connector such that the parameters shared through the connect equation can dynamically change. This works by attempting to match the parameters type-by-type and name-by-name between the two connections. Typically, a prompt is displayed to the user to define the connected variables when the connection is made.

For modelling the behaviour of a driver, basic driver models are typically formulated as basic transfer function. This has roots from the early days of driver modelling in control systems [22] [23], where the goal of the control system is to minimize the off-track error between the vehicle's position and the target path. This can be easily implemented within the Modelica workspace due to the symbolic nature of the Modelica programming language. Expandable connectors allow for any model signals to be accessed through the connector, which in the driver model, will not only include the physical connection of the steering wheel, but may access variables from the rest of the workspace. In Dymola, a window prompt provides a helpful list of possible variables which could be accessed through this connection.

3.4.4 SuperBlock Summary

At this point, it is important to reinforce the idea that this proposition is the outline for a framework of models, instead of the representation of a completed library of verified sub-systems. These SuperBlocks provide the essential building blocks to construct and store conventional simplified models of heavily researched vehicle sub-systems. Once ample analysis happens to be conducted and concluded on a particular sub-system, this SuperBlock structure allows the author to easily publish and deploy his or her work for others to use. Several additional simulation components are intended to be in the final product. These include the drivetrain SuperBlock and the environment SuperBlock. Development of these components resides in the next phase of research because the decisions to sort, organize and connect powertrain components have not been finalized. The environment block is intended to contain information regarding the road condition, target path, and environment visualizations, but limitations of the tire component library do not support these features.

This extra level of abstraction to manipulate the selection of sub-systems in a SuperBlock opens the possibility to easily investigate the topological arrangement for any new vehicle. The overall framework has had the advanced Modelica concepts tested within Dymola to ensure that the Modelica specification can support these features, but since implementation within MapleSim will require all the multibody components to change, parts of the Modelica code must be rewritten to be compatible. Moreover, since annotations may differ between the different Modelica applications, the compatibility of all the features needs to be re-examined. In the next chapter, successful implementation will be demonstrated beginning the future for the MapleCar.

Chapter 4

4. Implementation of the Topological Vehicle Modelling and Analysis Tool

The concept for a topology-based vehicle modelling tool began as a simple idea drafted in a slide-show presentation. This presentation outlined the topological commonalities when combining models of certain subsystems within a full vehicle model. These modular sub-systems were typically not limited to simulations involving only multi-body dynamics; many electrical and control elements are required alongside any hybrid or electric vehicle. Using these ideas, a proof-of-concept package was designed using the Modelica modelling language, and was successfully implemented within the MapleSim simulation platform beginning the development of MapleCar. This chapter describes the technical details on how MapleCar is structured and deployed within MapleSim. Example simulations are provided in the next chapter using the various SuperBlocks in the MapleCar package to demonstrate the beneficial features provided by this work.

4.1 Package File Structure

A suggested file structure for new packages is outlined in the Modelica Coding Practises guide [21]. There are several advantages to dividing any large packages into several files. The package will be easy to read from a development point of view, where sub-packages exist as separate Modelica files. The order of the import exists in a separate file named *package.order*. Multiple persons would be able to work on updating MapleCar components at the same time using any revision tracking software since the whole package will be built in modular parts ensuring the gradual development of a working package. The file management for the package also opens the opportunity to deploy a secured package to users while allowing custom user-defined libraries to be included.

The file structure for MapleCar is as follows:

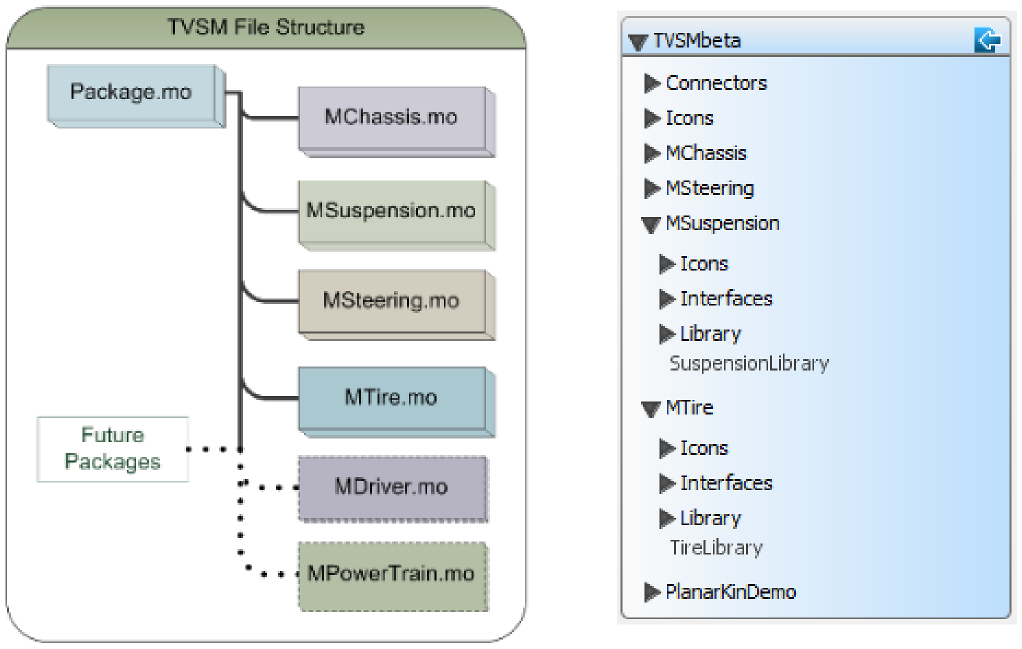


Figure 18 - MapleCar File Structure

When a folder is loaded as a package, there must be a file named *package.mo* that contains the definition of the package, and must be have the same name as the folder. Refer to the Modelica Language Specification [10] for more information regarding specific details related to authoring Modelica packages.

4.2 Layout of a SuperBlock Palette

One of the desired features within the initial MapleCar concept was the functionality of representing a high-level sub-system, for example a Suspension, as a single block which would have the different types of internal models adjustable through parameters. This idea is essentially a programming task to organize and access models in a clear, concise way implemented within the MapleSim work environment. For this idea to work, a straight-forward, high-level package structure must be defined for the sake of a clear component palette structure. Naturally, each high-level vehicle sub-system is represented by its own package, therefore the chassis, suspension, tire, and etc. sub-systems each appear as a separate drop-down package on the palette tree.

Within each of these sub-system palettes, the standard structure is followed with the exception of the *library* component. As seen in Figure 18, each package and sub-package is represented as a tree-structure dropdown in the side panel. The *library* component is the working name for the sub-system SuperBlock which accesses the compatible components in the l*ibrary* folder. All objects in the package are alphabetically ordered which further demands a standardized nomenclature for each model and package, but this is not a pressing issue.

4.3 Example of a MapleCar SuperBlock

The individual SuperBlock palettes are based on a generic SuperBlock template that was created after the advanced Modelica features were successfully tested in MapleSim6. A second round of a proof-of-concept SuperBlock was demonstrated using the proprietary MapleSim multibody components which utilize graph-theoretic solution methods. This SuperBlock was named "Planar

Kinematic Library" to clearly demonstrate sub-model equation changes using a drop-down parameter.

The first step to develop any SuperBlock for MapleCar is to first obtain a working model without using any Modelica features. In this case, a set of basic kinematic mechanisms were modelled in MapleSim as seen in Figure 19. As mentioned in Chapter 3, it is important to design these mechanisms with the same overall interfaces. All of the mechanisms have two multibody frames, the left connected to a fixed frame and the right connected to a rigid body mass and a tracing visualization. Each of these mechanisms is driven with a single 1-D rotational frame.

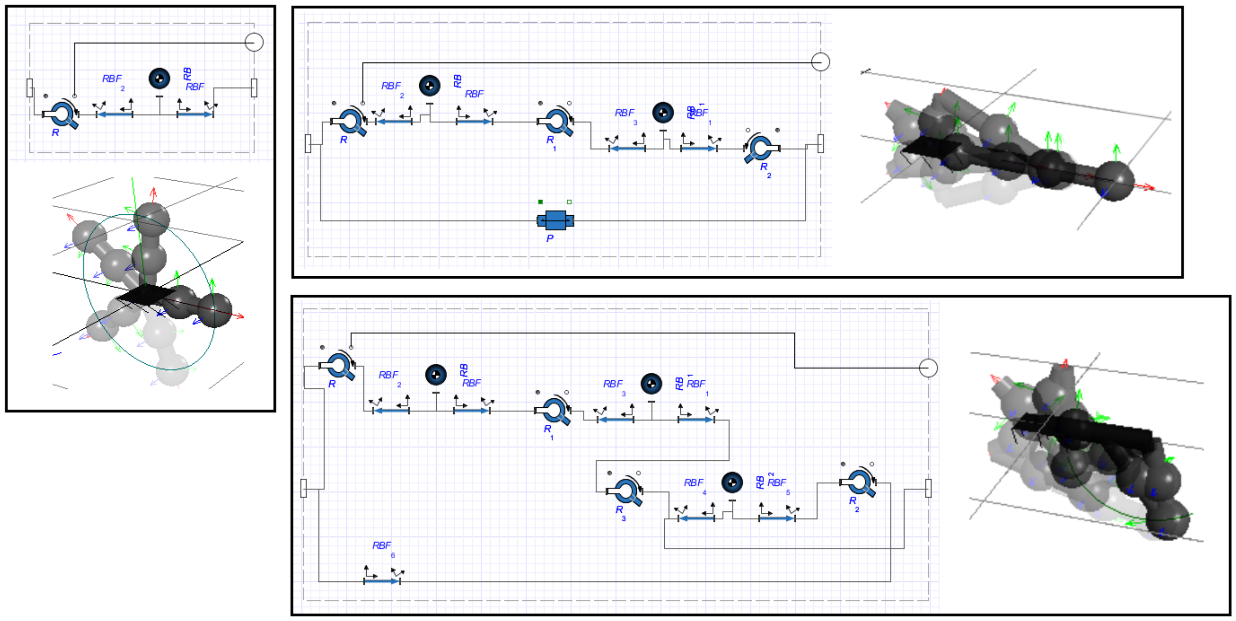
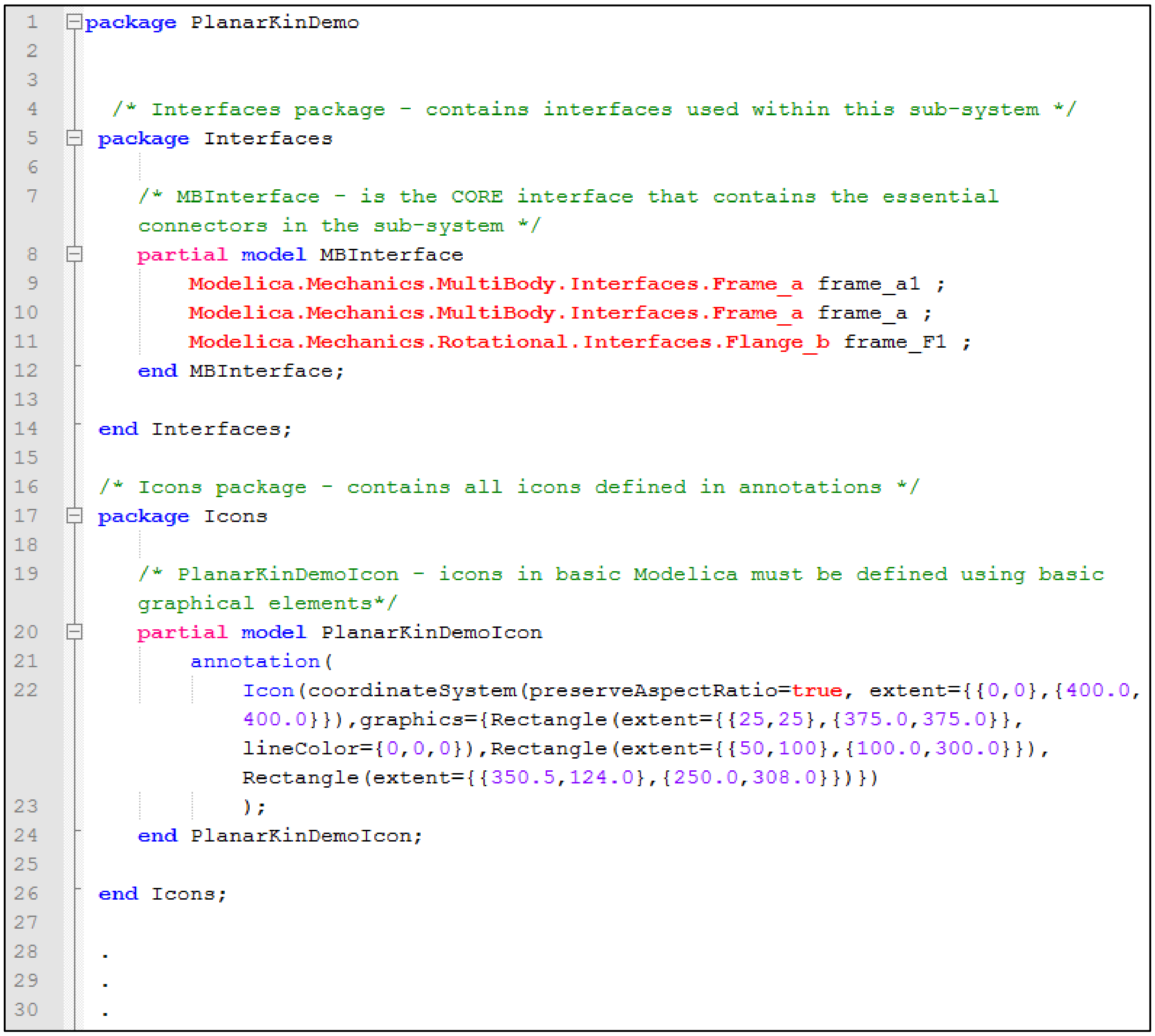


Figure 19 - MapleSim models of basic mechanisms – revolute link, slider-crank, and four-bar

Once these library components have been modelled within MapleSim and the base interface has been identified, the construction of a SuperBlock can continue. An entire new package must be declared in accordance to MapleSim package naming conventions. The sub-packages of the base interface and model icons can be defined as seen in Figure 20. Note that extraneous annotations have been removed from frame placements and model graphics.

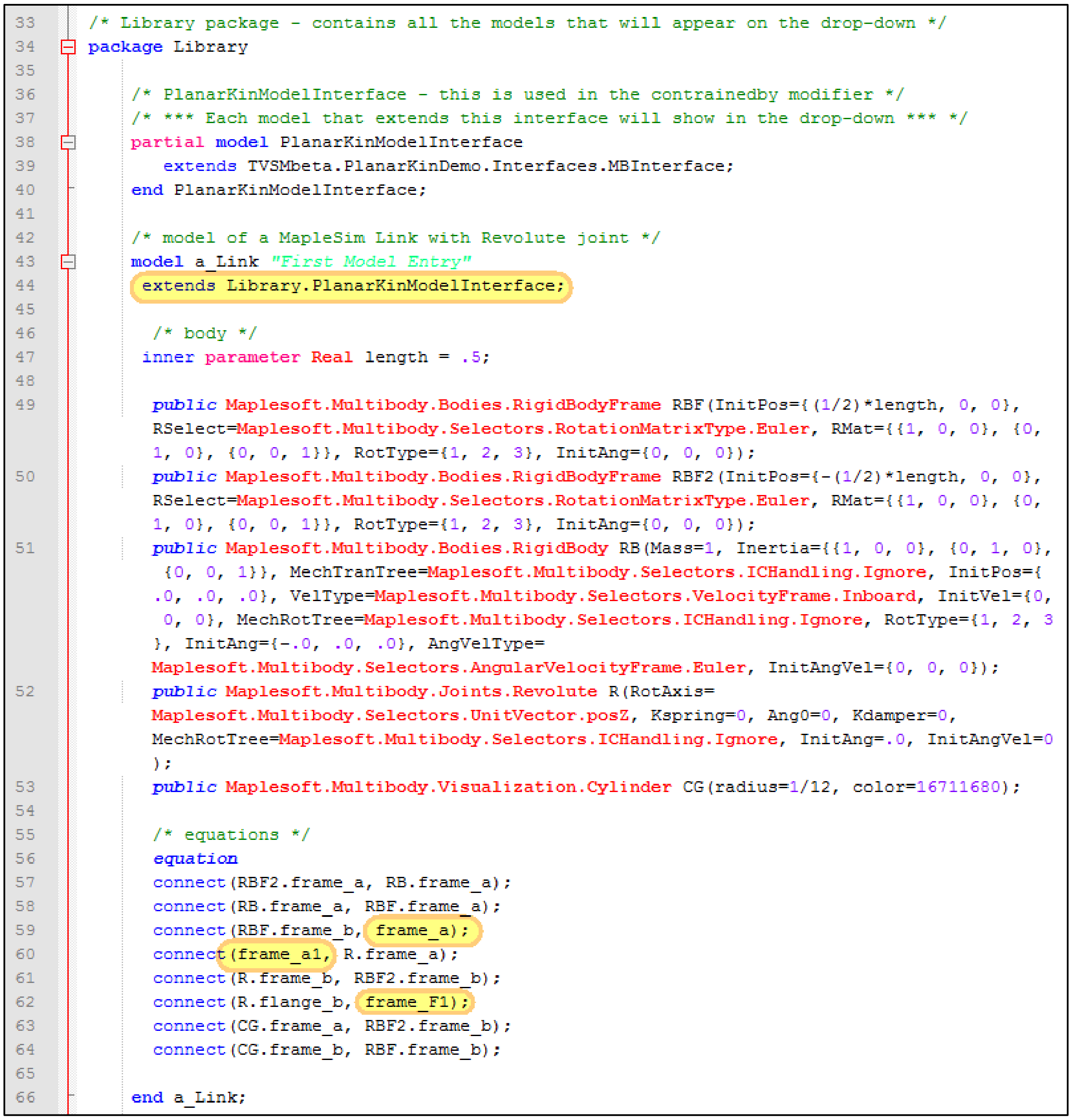


```
package PlanarKinDemo


  /* Interfaces package - contains interfaces used within this sub-system */
 package Interfaces

    /* MBInterface - is the CORE interface that contains the essential
    connectors in the sub-system */
    partial model MBInterface
        Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a1 ;
        Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a ;
        Modelica.Mechanics.Rotational.Interfaces.Flange_b frame_F1 ;
    end MBInterface;

 end Interfaces;

 /* Icons package - contains all icons defined in annotations */
 package Icons

    /* PlanarKinDemoIcon - icons in basic Modelica must be defined using basic
    graphical elements*/
    partial model PlanarKinDemoIcon
        annotation(
            Icon(coordinateSystem(preserveAspectRatio=true, extent={{0,0},{400.0,
            400.0}}),graphics={Rectangle(extent={{25,25},{375.0,375.0}},
            lineColor={0,0,0}),Rectangle(extent={{50,100},{100.0,300.0}}),
            Rectangle(extent={{350.5,124.0},{250.0,308.0}}))
            );
    end PlanarKinDemoIcon;

 end Icons;

 .
 .
 .
```

Figure 20 - Kinematic SuperBlock Demo - Interface and Icons

Next, the package of library components must be declared using the original MapleSim components. It is important to use MapleSim6 in this step as the new Modelica source code can be viewed within MapleSim6. The exact Modelica code to generate the working MapleSim model can be referenced and used within the library package. This will essentially ‘lock-in’ the model such that no further changes can be made unless the source code is modified; MapleSim cannot edit or change any imported Modelica packages. This library package will be the working folder that holds all the components the SuperBlock will access. The *constrainedby* interface must also be declared here matching the same programming technique explained in Chapter 3.

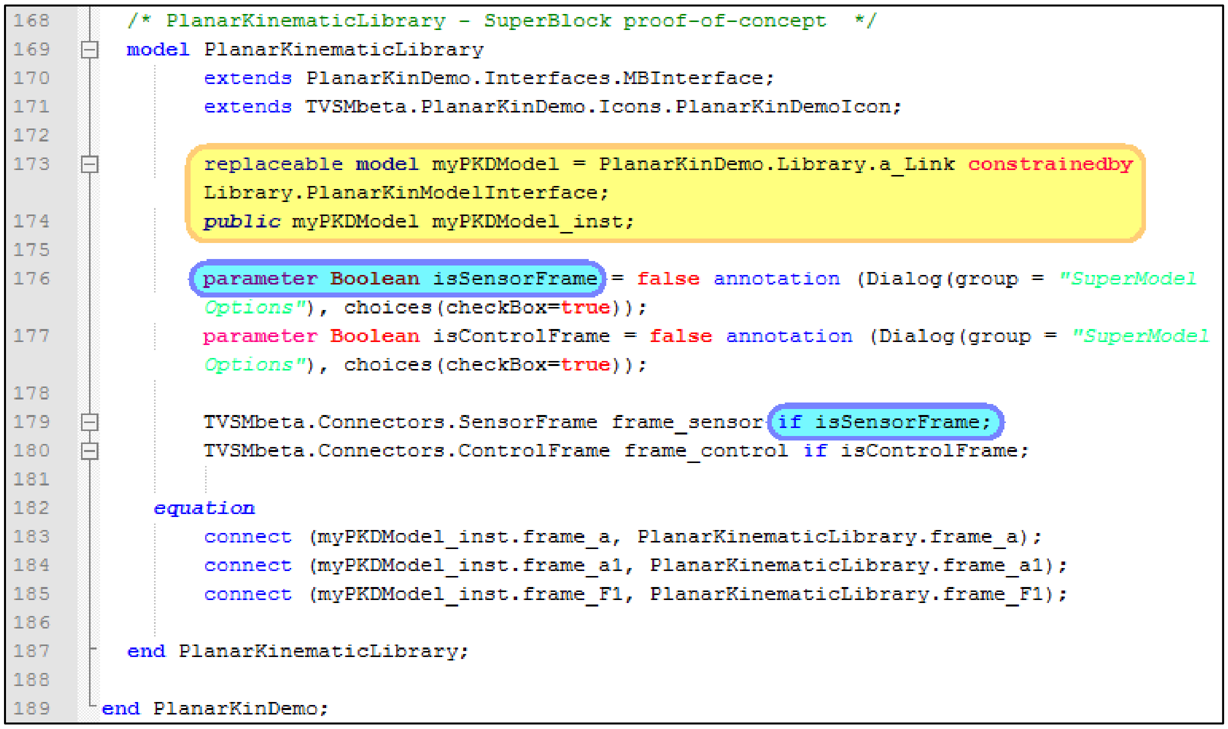
Figure 21 shows this Modelica code with the first library component, a simple link with a revolute joint. When the MapleSim Modelica code is copied, it must be manually updated to properly utilize the added partial frame required for the *constrainedby* modifier. The partial frame must be extended in the library component and the connect equations must be updated. These changes are highlighted in the same figure. It should be clear that since only 3 connectors exist in the base interface, there must be 3 changes in the connect equations. For brevity, the source code for the slider-crank and four-bar library components will not be shown.



```
/* Library package - contains all the models that will appear on the drop-down */
package Library

    /* PlanarKinModelInterface - this is used in the contrainedby modifier */
    /* *** Each model that extends this interface will show in the drop-down *** */
    partial model PlanarKinModelInterface
       extends TVSMbeta.PlanarKinDemo.Interfaces.MBInterface;
    end PlanarKinModelInterface;

    /* model of a MapleSim Link with Revolute joint */
    model a_Link "First Model Entry"
     extends Library.PlanarKinModelInterface;

      /* body */
     inner parameter Real length = .5;

      public Maplesoft.Multibody.Bodies.RigidBodyFrame RBF(InitPos={(1/2)*length, 0, 0},
      RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0,
      1, 0}, {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0});
      public Maplesoft.Multibody.Bodies.RigidBodyFrame RBF2(InitPos={-(1/2)*length, 0, 0},
      RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0,
      1, 0}, {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0});
      public Maplesoft.Multibody.Bodies.RigidBody RB(Mass=1, Inertia={{1, 0, 0}, {0, 1, 0},
       {0, 0, 1}}, MechTranTree=Maplesoft.Multibody.Selectors.ICHandling.Ignore, InitPos={
      .0, .0, .0}, VelType=Maplesoft.Multibody.Selectors.VelocityFrame.Inboard, InitVel={0,
       0, 0}, MechRotTree=Maplesoft.Multibody.Selectors.ICHandling.Ignore, RotType={1, 2, 3
      }, InitAng={-.0, .0, .0}, AngVelType=
      Maplesoft.Multibody.Selectors.AngularVelocityFrame.Euler, InitAngVel={0, 0, 0});
      public Maplesoft.Multibody.Joints.Revolute R(RotAxis=
      Maplesoft.Multibody.Selectors.UnitVector.posZ, Kspring=0, Ang0=0, Kdamper=0,
      MechRotTree=Maplesoft.Multibody.Selectors.ICHandling.Ignore, InitAng=.0, InitAngVel=0
      );
      public Maplesoft.Multibody.Visualization.Cylinder CG(radius=1/12, color=16711680);

      /* equations */
      equation
      connect(RBF2.frame_a, RB.frame_a);
      connect(RB.frame_a, RBF.frame_a);
      connect(RBF.frame_b, frame_a);
      connect(frame_a1, R.frame_a);
      connect(R.frame_b, RBF2.frame_b);
      connect(R.flange_b, frame_F1);
      connect(CG.frame_a, RBF2.frame_b);
      connect(CG.frame_b, RBF.frame_b);

    end a_Link;
```

Figure 21 - Kinematic Library SuperBlock Demo – component library

Finally, the SuperBlock model can be completed using the *models with replaceable models* programming technique as seen in Figure 22. The interface of the SuperBlock should match the interface used in the library components. An icon from the icons package should be extended to graphically inform the user which SuperBlock this model is. The connection equations simply connect the SuperBlock connections with the sub-model connections. Since all selected components utilize the same frame, these connection equations will never change. Also highlighted in Figure 22 is the optional syntax to create toggleable frames. The simple *if* [Boolean] syntax will enable or disable the connector which will toggle the connection point in the workspace. This is primarily how the chassis SuperBlock is programmed; there is no need to have a drop-down selection to select how many axles will be displayed when each axle can have its own Boolean checkbox parameter. When these optional frames are disabled, MapleSim will automatically void the connection such that the simulation will ignore the dangling equation.



```
/* PlanarKinematicLibrary - SuperBlock proof-of-concept  */
model PlanarKinematicLibrary
      extends PlanarKinDemo.Interfaces.MBInterface;
      extends TVSMbeta.PlanarKinDemo.Icons.PlanarKinDemoIcon;

      replaceable model myPKDModel = PlanarKinDemo.Library.a_Link constrainedby
      Library.PlanarKinModelInterface;
      public myPKDModel myPKDModel_inst;

      parameter Boolean isSensorFrame = false annotation (Dialog(group = "SuperModel
      Options"), choices(checkBox=true));
      parameter Boolean isControlFrame = false annotation (Dialog(group = "SuperModel
      Options"), choices(checkBox=true));

      TVSMbeta.Connectors.SensorFrame frame_sensor if isSensorFrame;
      TVSMbeta.Connectors.ControlFrame frame_control if isControlFrame;

    equation
      connect (myPKDModel_inst.frame_a, PlanarKinematicLibrary.frame_a);
      connect (myPKDModel_inst.frame_a1, PlanarKinematicLibrary.frame_a1);
      connect (myPKDModel_inst.frame_F1, PlanarKinematicLibrary.frame_F1);

  end PlanarKinematicLibrary;

end PlanarKinDemo;
```

Figure 22 - Kinematic SuperBlock Source Code

With this SuperBlock Modelica package imported into MapleSim, all the core functionalities desired in the MapleCar package are achieved. To visually reiterate the referencing and selection of these SuperBlock features, Figure 23 shows a basic visual description of the SuperBlock.

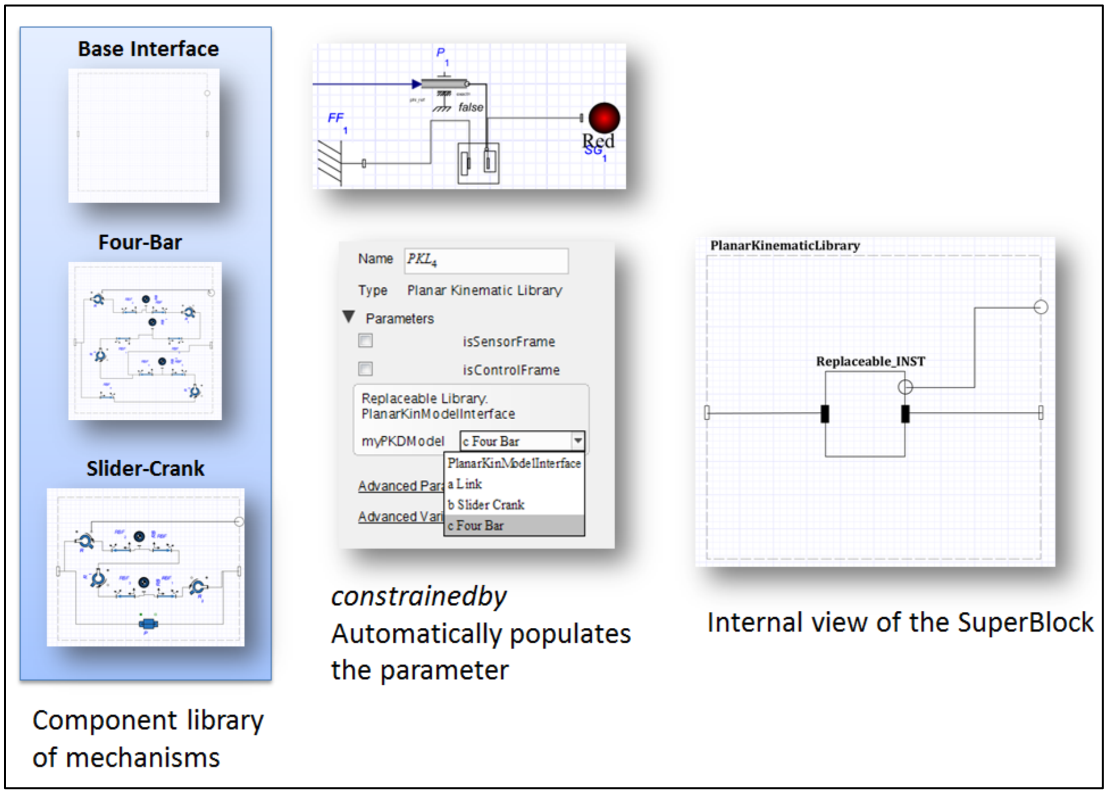


Figure 23 - Visual description of the SuperBlock

4.4 Extending Pre-Existing Packages

A tire model library and driveline toolkit currently exists as an external retail toolbox add-on that can be used in MapleSim. The Tire component library consists of a set of models that contain verified textbook equations to simulate the tire-ground interactions. Currently, vertical road excitations cannot be processed with the MapleSoft Tire Component Library so it automatically connects to a global, flat ground-plane. Therefore, the only connection needed for these Tire Models is a *MapleSoft.Multibody* frame which always connects to a *Maplesoft.Tires.TireBody* component. Because this consistency in frames is ideal for Topological Vehicle design, these models can be used as part of the library components in a SuperBlock model structure. This essentially encloses the additional toolbox package within its own SuperBlock, which will make it easier for a user to quickly switch and test the response of a vehicle with differing Tire models.

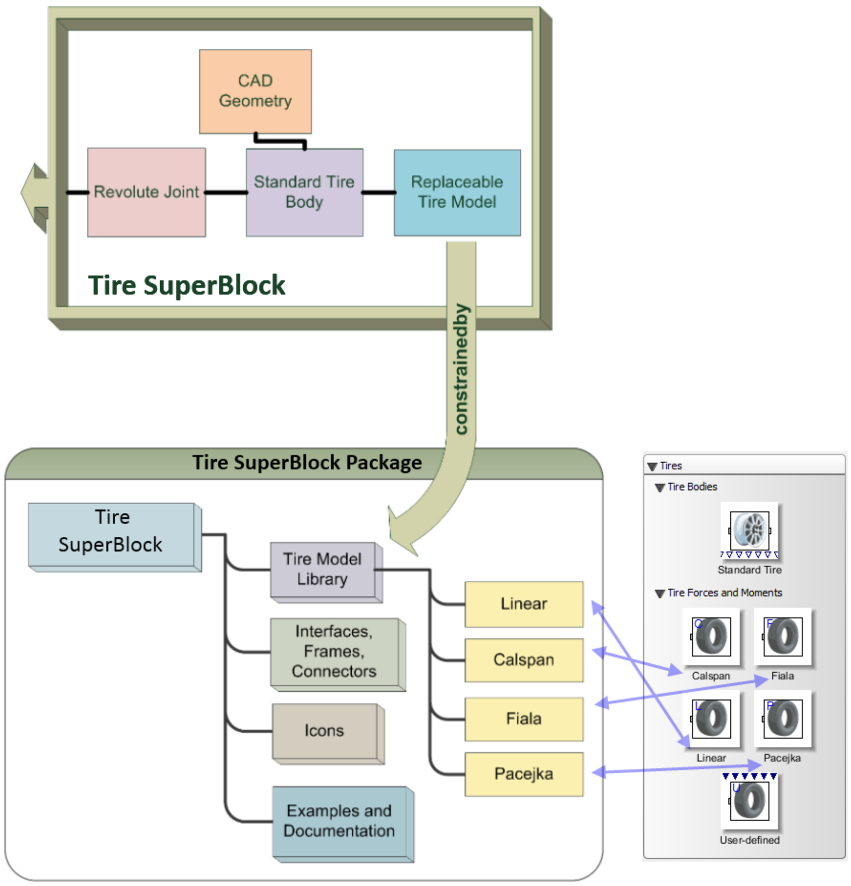


Figure 24 – Tire SuperBlock Package Structure utilizing the built-in Tire Library Components

Another package recently published by MapleSoft is the Drive-line Component Library. This library "covers all aspects of the power-train, from the engine to the differential, wheels and road loads" [15]. Again, the generic template of a SuperBlock will be used to provide a wrapper package to contain a library of completed drive-line transmission systems.

4.5 MapleCar in MapleSim

A large portion of MapleSim6 was to further develop the integration of the newer Modelica Specification along with the recent versions of the Modelica Standard Library. With the latest version of MapleSim, loading the MapleCar package is as simple as importing the proper folder. The raw Modelica code is parsed and read into the MapleSim workspace and appears as a new custom Modelica palette on the left side. As a proof-of-concept package, most of the core features are fully functional and vehicle models can be easily constructed and simulated. The other SuperBlock packages have all been derived from the given example in this chapter. Included in the appendix is the raw source code for several MapleCar SuperBlocks. As with any other programming project, the code is expected to evolve over time according to updates to MapleSim and Modelica.

It is important to note that there are many compatibility issues when using custom Modelica code within MapleSim6. Managing any instance of a SuperBlock model only exists on the main SuperBlock level. With Dymola, each instantiated SuperBlock has clear controls to handle the sub-model instance. This includes modifying the parameters, provide visual feedback of the sub-model diagram, and display full run-time visualization of the SuperBlock. The drop-down parameter generated with the *contrainedby* option contained an extra *edit* button to allow changes to the selected model's parameters which would be used within the SuperBlock instance. This streamlines using and configuring the SuperBlock in the model since the selected component in the parameter list can be viewed and parameterised from a single workspace. This feature is also available in the MapleSim through an a*dvanced parameters options* button, but it does not properly display the selected sub-model. In addition, examining the diagram in any

SuperBlock sub-model will always display the default model even though the selection correctly changes the sub-model component. These bugs make working with SuperBlocks very troublesome and impractical for any real analysis. However, patches and version updates to the software will hopefully rectify these issues such that the MapleCar might eventually become a polished product.

With respect to the actual development of MapleCar, there were countless hurdles to overcome. It is very clear that MapleSim was initially intended to not include such intense Modelica customization. The Modelica language is a great vehicle to quickly develop an application that contains drag-and-drop blocks which contain analytical component equations. This is ideal when combined with the power of Maple, a symbolic computation engine. But since this work is primarily focused on developing Modelica code, many undocumented nuances had to be manually ironed out. The presentation of icons and graphical elements like lines and textboxes exist outside the domain of Modelica; application specific elements are used, such as fonts, gradients and images, which do not appear in the Modelica code. The workspace coordinate system is also modified from the default Modelica specification making the construction Modelica based model icons very difficult.

Chapter 5

5. Simulation Examples

The initial concept for MapleCar has been successfully implemented in MapleSim6. On the whole, the project presented here is only the cornerstone and blueprint for the foundation of the grand vision provided to both MapleSoft and Dr. Khajepour. Basic examples will be presented, demonstrating the model construction procedure and sample analysis using MapleCar. Everything that will be simulated uses standard MapleSim components. For each model, there exists an equal and equivalent MapleSim model that does not use any of the advanced Modelica features. As such, little emphasis will be put on validating any of these results. If there is a validated vehicle model in MapleSim provided by another party, it can be organized and placed inside the various SuperBlock libraries for use within MapleCar.

5.1 Construction of a Simple Vehicle Model

A roll-pitch vehicle model is one of the simplest vehicle models that exists in three dimensions. It consists of a lumped sprung mass connected to four spring-mass-dampers at each corner of the vehicle. Depending on the specific type of simulation, either a second static spring-mass-damper is added to the end to emulate the stiffness and damping of the tire, or an actual tire model is used to simulate the motion of the vehicle on a road. This model is provided within the tire examples library included with the tire component toolbox and is a great reference to begin modelling. The basic components of this simple vehicle model are shown in Figure 25. This example will demonstrate the needed steps to recreate the same model using MapleCar. Furthermore, changes to the suspension models and tire models will be shown.

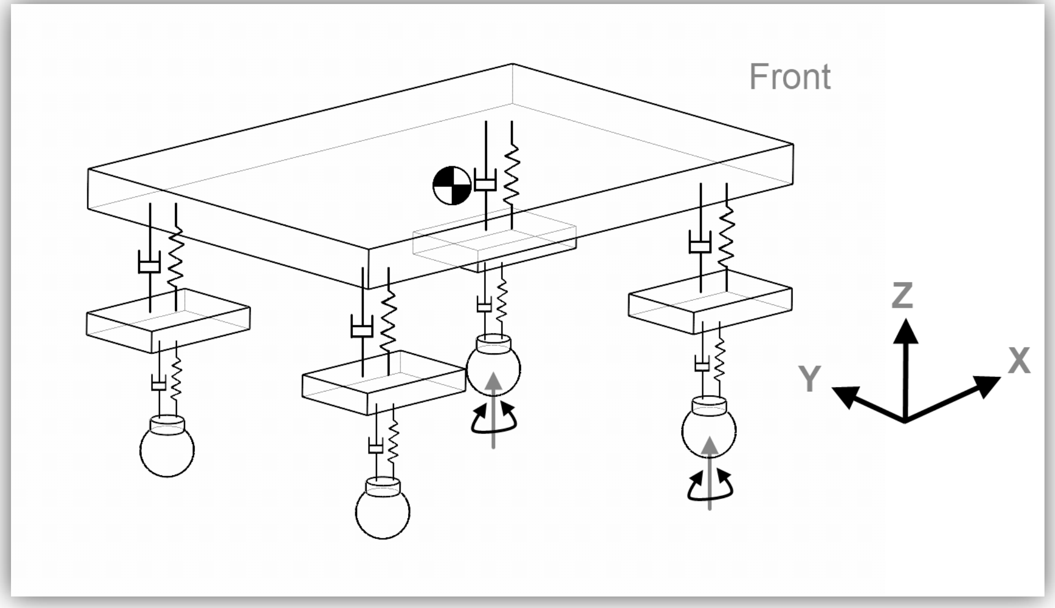


Figure 25 – Chassis and Suspension of a simple Roll-Pitch vehicle model

Figure 26 shows the diagram view of the MapleSim model. When looking at the MapleSim workspace, it is clear which MapleSim blocks are simplifications of real components. The center of mass is a simplification of the chassis shown in the blue border. The suspension is simplified in the prismatic joint allowing for one degree of vertical motion shown in the green borders. The tire is modelled using the Standard Tire Body and a set of known Tire Models within the FTSS sub-model. If a researcher is unsatisfied with the amount of simplifications, the components of the vehicle model can be replaced with higher fidelity models. For example, the suspension can be replaced with a complete multi-body model, or the steering can be replaced with the action of rack and pinion mechanism. It is up to the modeller to decide what behaviours the vehicle model should capture.

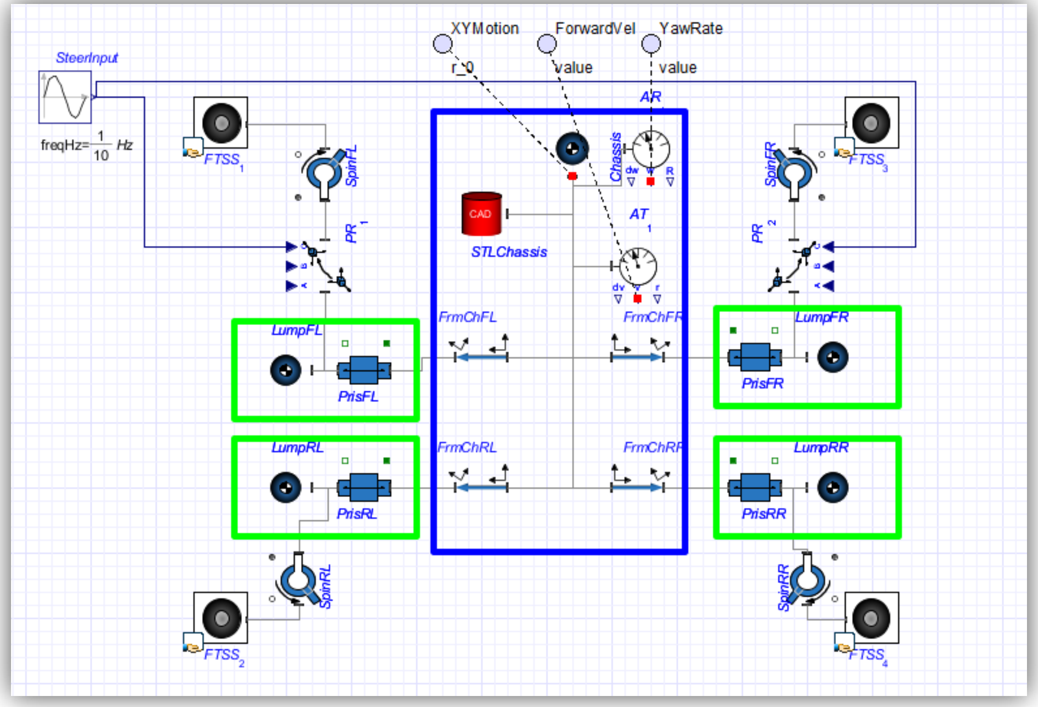


Figure 26 - MapleSim model of a Simple Vehicle Model (From Tire Examples)

However, when this model is being used by multiple people and contains components from different users, several problems may arise. MapleSim is not strong at conveying the definition of shared components and sub-systems to the user. Creating a shared sub-system defines a new sub-system within the active MapleSim model. This can then be added into the User-Defined MapleSim library. However, when tweaking and updating this MapleSim Library, there is no explicit definition from the user for the shared sub-system to be instantiated from the current MapleSim model in focus or from a custom *User-Defined Library*.

The potential issue with naming convention also applies to the naming of parameters and parameter blocks. For large sub-systems, there might be a need to have a large list of parameters. However, parameters must be added one at a time and cannot be easily reordered. A uniform naming convention must be understood by all the users working with the model. Between defining parameters external to a model, within a parameter set or within a sub-model, the workspace can very easily hide parameters making complex models difficult to understand. Creating components is essentially physical programming of machines and mechanisms and requires the same level of care and attention as regular programming.

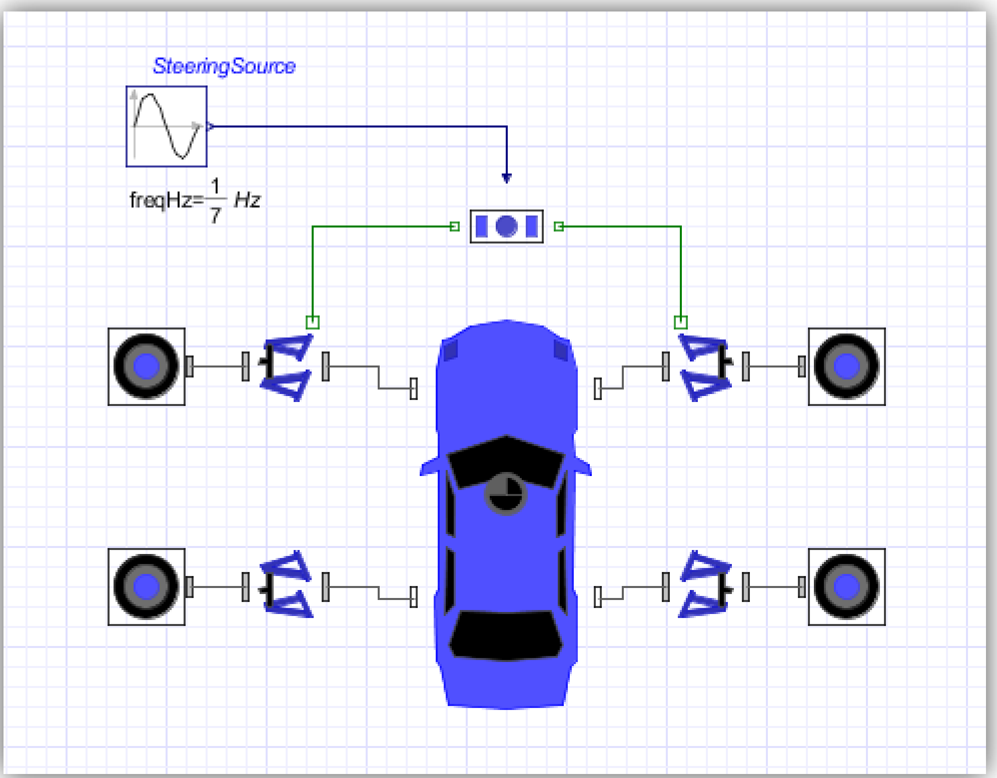
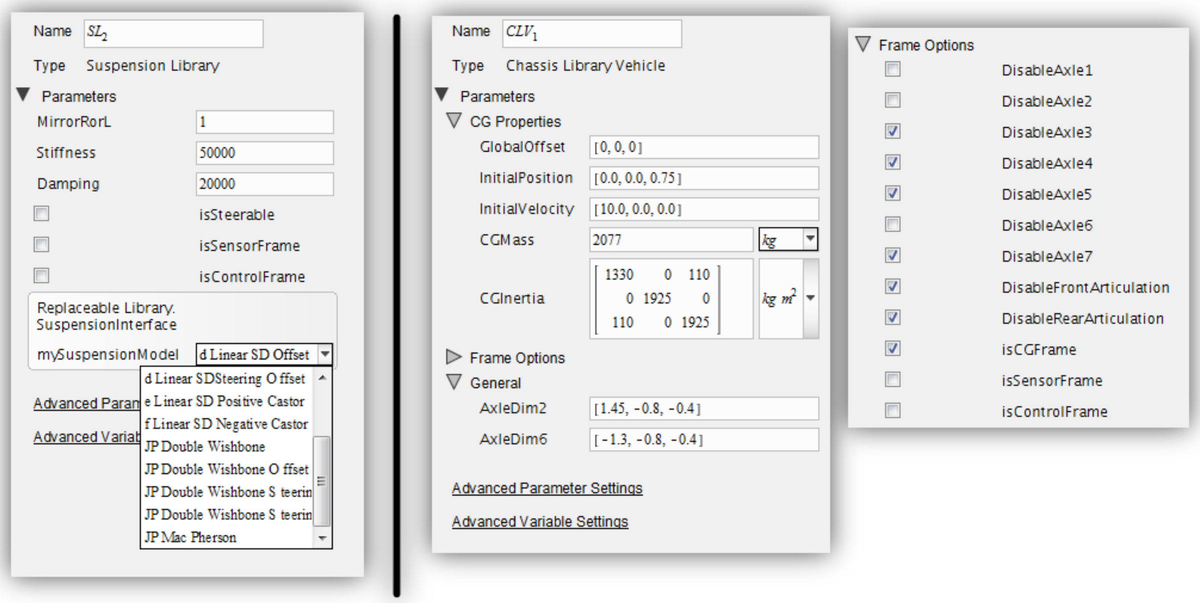


Figure 27 - The simple vehicle model constructed with SuperBlocks

With the MapleCar package, the topology of the vehicle is a straight-forward assembly of SuperBlocks. A chassis SuperBlock is dropped into the workspace to model the sprung mass of the vehicle, and four suspension SuperBlocks are inserted to connect to the chassis. Four tire SuperBlocks are connected to the suspensions and a steering SuperBlock connects to the front, steerable suspension systems. The conversion to a steerable suspension system is done through a checkbox, *isSteerable*, in the suspension SuperBlock. Once the overall topology of the vehicle is assembled, the main simulation does not need any further change



Name	SL_2
Type	Suspension Library
Parameters	
MirrorRorL	1
Stiffness	50000
Damping	20000
☐	isSteerable
☐	isSensorFrame
☐	isControlFrame
Replaceable Library. SuspensionInterface	
mySuspensionModel	d Linear SD Offset

d Linear SDSteering Offset
e Linear SD Positive Castor
f Linear SD Negative Castor
JP Double Wishbone
JP Double Wishbone Offset
JP Double Wishbone Steerin
JP Double Wishbone Steerin
JP Mac Pherson

Advanced Param
Advanced Variab

Name	CLV_1
Type	Chassis Library Vehicle
Parameters	
CG Properties	
GlobalOffset	[0, 0, 0]
InitialPosition	[0.0, 0.0, 0.75]
InitialVelocity	[10.0, 0.0, 0.0]
CGMass	2077 *kg*
CGInertia	$\begin{bmatrix} 1330 & 0 & 110 \\ 0 & 1925 & 0 \\ 110 & 0 & 1925 \end{bmatrix}$ $kg\ m^2$
Frame Options	
General	
AxleDim2	[1.45, -0.8, -0.4]
AxleDim6	[-1.3, -0.8, -0.4]

Advanced Parameter Settings
Advanced Variable Settings

Frame Options	
☐	DisableAxle1
☐	DisableAxle2
✓	DisableAxle3
✓	DisableAxle4
✓	DisableAxle5
☐	DisableAxle6
✓	DisableAxle7
✓	DisableFrontArticulation
✓	DisableRearArticulation
✓	isCGFrame
☐	isSensorFrame
☐	isControlFrame

Figure 28 – Parameters for MapleCar SuperBlocks; Left: Suspension, Right: Chassis

The same Roll-Pitch vehicle model is easily constructed with high-level SuperBlocks as shown in Figure 27. The block icons are very clear in letting the user know what each sub-system represents as well as having the connectors logically placed with respect to the icon. All of the important parameters are immediately accessible from the SuperBlocks, and since the model of each vehicle sub-system can be changed with a simple drop-down menu, comparative analysis between differing tire models can be easily performed.

5.2 Four-Wheel Steering Model

The concept of Four-Wheel Steering (4WS) vehicles is a specialized feature designed to improve the steering response of a vehicle, increasing straight-line stability at high speeds, and decreasing the turning radius of a vehicle at low speeds. Each benefit is achievable through the addition of steering to the set of rear-wheels. Depending on the state of the vehicle, the rear-wheels will either steer with or against the front-wheel. This is automatically controlled with a computer and actuators so its operation is transparent to the driver [24].

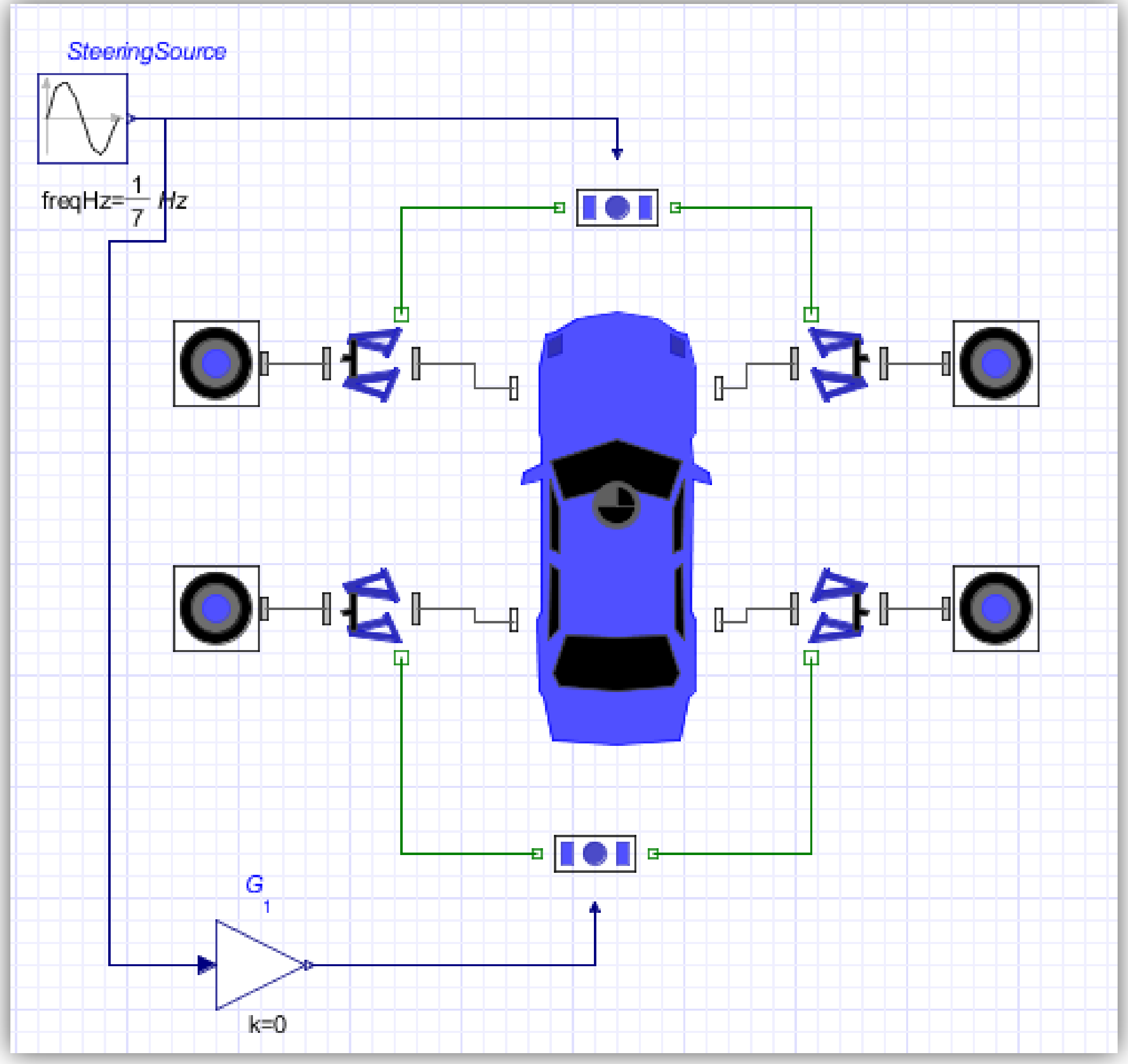


Figure 29 - Construction of a Four-Wheel Steer model using SuperBlocks

To enable the rear axle of a vehicle to steer, the suspension needs the steering from enabled through the checkbox in the parameters list. Furthermore, the selected model must be a steerable suspension model. A separate steering mechanism must also be introduced for the rear axle.

5.2.1 Low-Speed Turning Radius Improvement

At very low speeds, a tighter turning radius is preferred for parking and U-Turn maneuvers. This is important for cities with smaller streets that may not accommodate low speed maneuverability [24]. As we can see in Figure 30, various amounts of negative steering gain results in a tighter turning radius (a higher yaw rate) without significantly changing the steering limits on the front axle. This occurs because both axles, the front and the rear, contribute to the overall yaw rate of the vehicle by creating a larger difference in steer angle compared to a normal front axle steering.

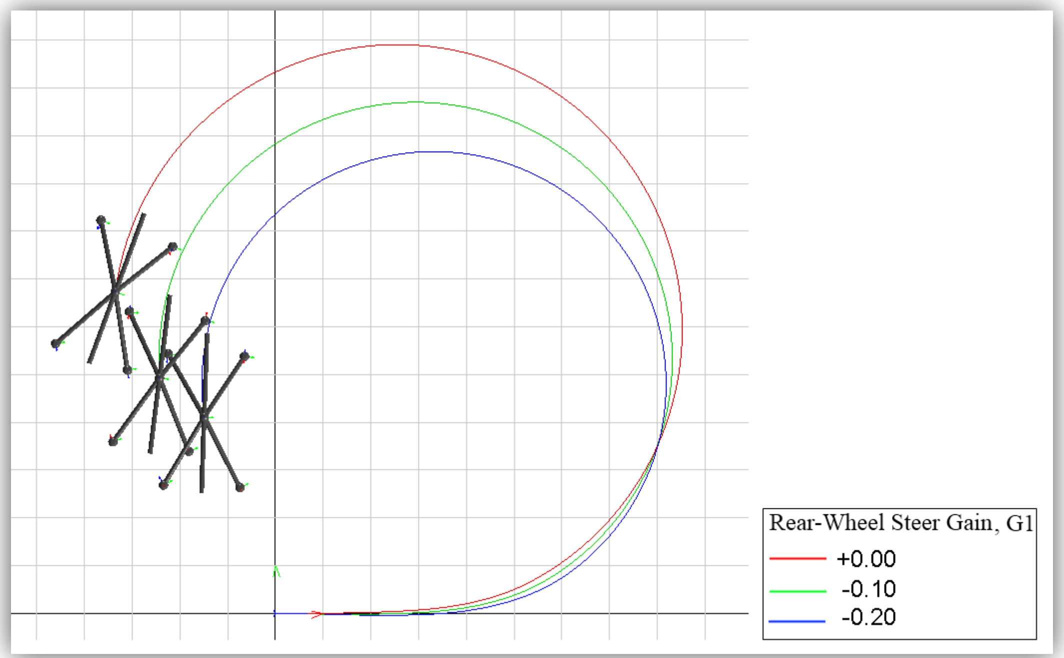


Figure 30 - Comparative visualization of various 4WS gains

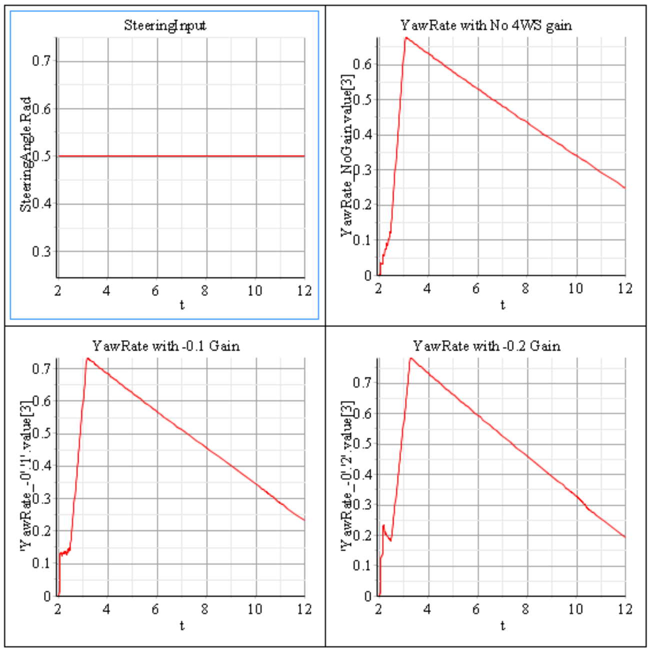


Figure 31 – Steer Input and Yaw Rates of 4WS a comparative simulation

The above results demonstrate this four-wheel steer action. Notice that in the case of no four-wheel steering, where the rear angle gain is 0, the turning radius is the largest in the figure. This is clearly reflected in the yaw-rate graph; the peak yaw-rate of the case of no 4WS is about 0.678 radians per second, whereas with -0.10 rear-wheel steering gain the peak yaw-rate of the vehicle is 0.730 radians per second and with -0.20 rear-wheel steering gain the peak yaw-rate is 0.783 radians per second. The yaw-rate in this result decreases because the model is passively rolling resulting in a consistent loss of speed. The model also uses the Fiala tire model which includes a basic representation of rolling resistance. However, this increase in peak yaw-rate for larger negative steering gain does follow natural intuition of the situation. In a bicycle model, a perfect counter symmetric rear steering gain of -1.0 during a maneuver with a steering angle of

90 degrees would cause the bicycle model to spin on the spot; the turning radius of the bicycle model would be zero resulting in a perfect spin.

In any case, the MapleCar package allows for an analysis such as this one to be easily assembled and simulated. Using the powerful MapleSim multibody solver combined with the tire component library, the parallel simulation of 3 simple vehicle models each with four-wheel steering enabled only takes 59 seconds to run a 10 second analysis. For this model, MapleSim determines that 423 equations exist within the model. Many of these equations are most likely extraneous component equations that results from the extra chassis offsets hidden from the user. Simplification of these equations is automatically determined and the solver automatically removes 369 equations resulting in 54 equations to determine the entire model.

5.2.2 High-Speed Straight Line Stability

At highway speeds, emergency lane change maneuvers may risk the yaw control of the vehicle due to the tire slipping. 'Skidding out' is especially dangerous with unexpected changes of road condition due to weather conditions. With high speeds, yaw stability is paramount in ensuring the safety of its passengers.

With a four-wheel steering vehicle, it is preferred for the rear wheels to turn in the same direction of the front wheels during high-way speeds. This reduces the amount of yaw experienced by the vehicle, converting it into an intentional sideways drift. The result of this simulation is shown below.

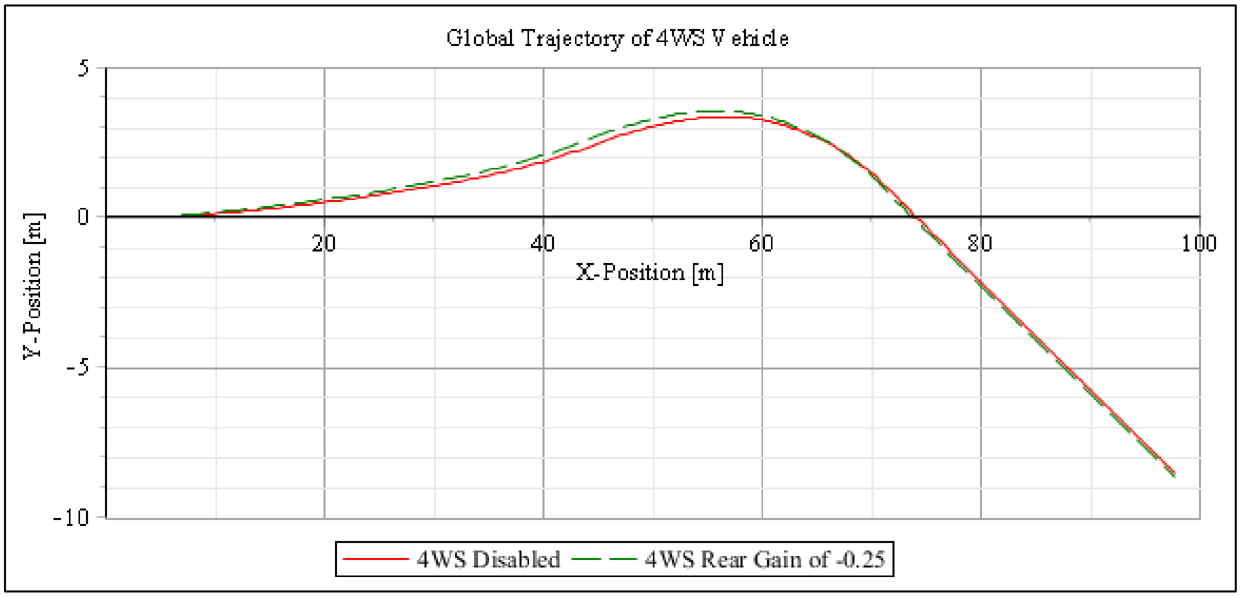


Figure 32 - High Speed Four Wheel Steering - Global Path

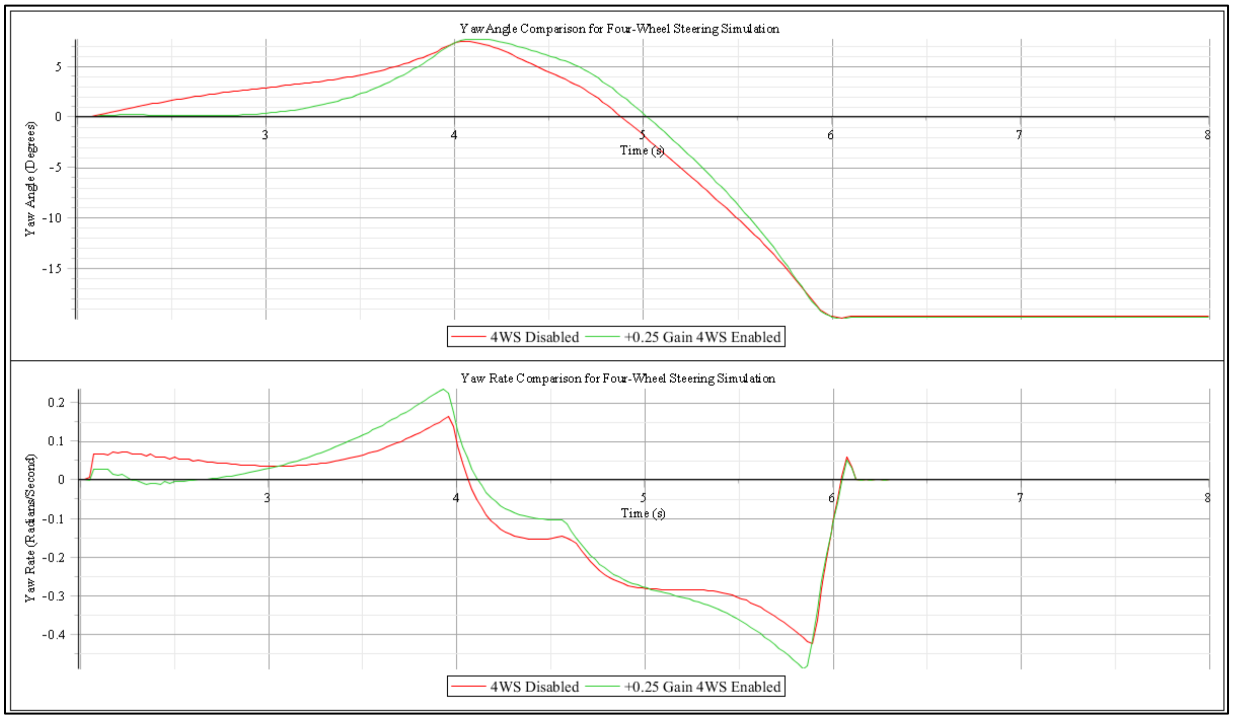


Figure 33 - High Speed Four Wheel Steering - Yaw Measures

In Figure 32, it is clear that the vehicle with a positive 0.1 gain of rear-wheel steering in the same direction slightly increases the amount of travel in the lateral direction. Meanwhile, looking at the results of the yaw angles experienced between the vehicles in Figure 33, the vehicle with 4WS actually turns less at the beginning of the maneuver. This response should yield a slightly greater feeling of yaw control when traveling at high speeds. To further investigate the straight-line stability of a vehicle at high speeds, a simulation can also be performed using a trailer to further examine the benefits of 4WS.

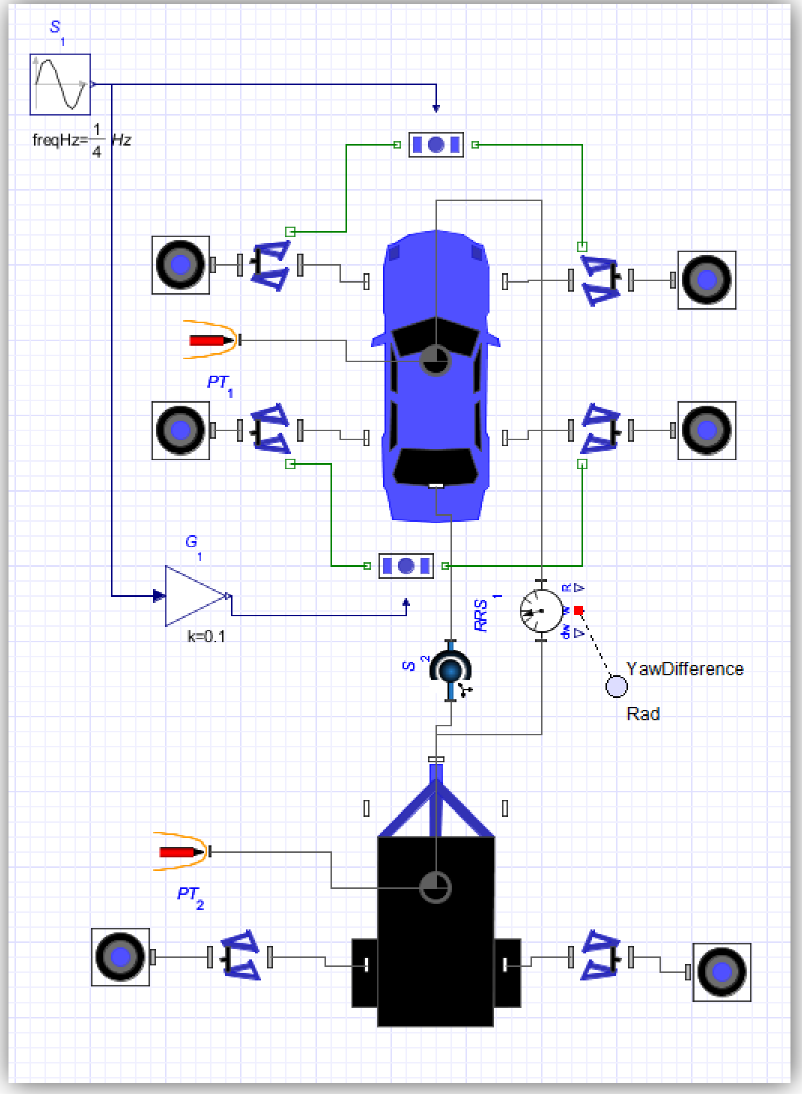


Figure 34 - 4WS Car with Trailer

In the diagram, a trailer is attached to the vehicle through a rear-articulation joint that is enabled through the chassis. The trailer itself contains a single axle with a stiff suspension and a front-articulation joint to facilitate the constraint of the hitch. The hitch is modelled between these two Chassis models as an ideal spherical joint. The same lane change maneuver is performed.

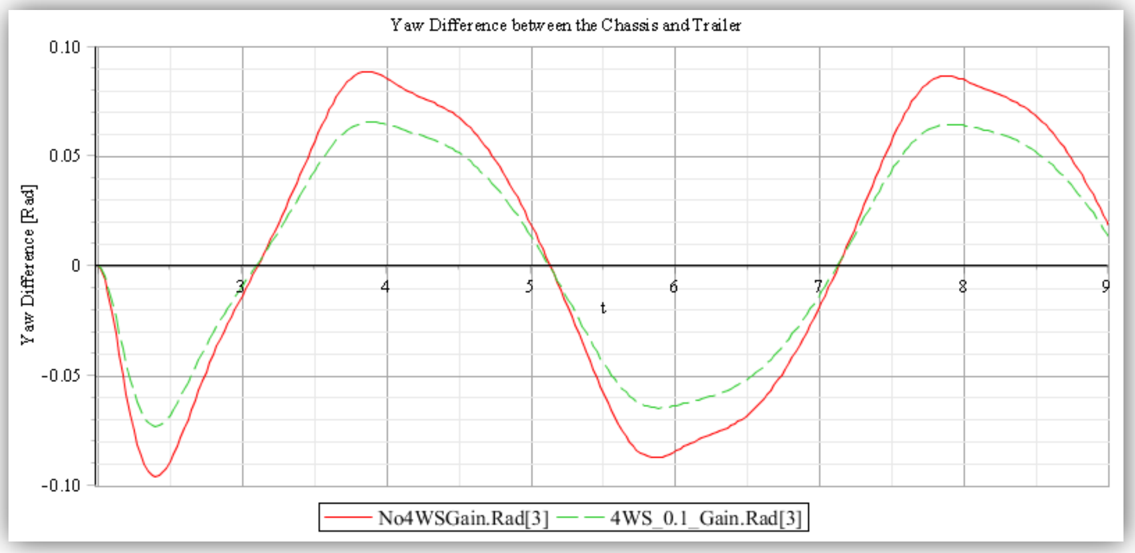


Figure 35 - 4WS Car with Trailer - Yaw Rate Difference

Looking at the difference in magnitude between the two simulations, performing the sine-steer maneuver at high speeds results in a slightly less yaw rate difference between the car and the trailer. If the sine-steer would be periodically continue, the snaking effect might be experienced by the trailer. This occurs because a new critical frequency exists in the model due to the spherical articulation joint. With a more intelligent controller varying the amount of rear-steering in the system, this snaking effect may be attenuated to provide even better control for the driver.

In both of these simulations, the total runtime of the software is less than one minute. This is a fairly quick total simulation time due to the simplification process automatically performed by MapleSim. The first simulation contains two of the same vehicle models with slightly different rear-wheel steering gains. The total number of equations present during simulation time is 470. This is clearly an inflated figure since the model contains many redundant equations due to the extra components within each chassis SuperBlock. In any case, MapleSim automatically reduces this large number of equations to only 42 resulting in the short overall simulation time.

5.3 Long-Combination Vehicles (LCV) and Road Train Simulations

Due to the constantly increasing demand on the transportation of goods, effective strategies to improve the payload of each highway vehicle are being explored. Without much change to the current vehicle technology, articulated highway vehicles are in the process of being converted to road trains, or long-combination vehicles [20] [25]. These vehicles contain the same tractor, but instead of only a single semi-trailer being hauled, multiple stages are connected with one another.

The B-Double road train configuration is a slight modification to the standard tractor and semi-trailer combination typically seen on roads. With the created vehicle-systems package, large and complex simulations can be easily constructed due to the accessible modularity of the components. This is ideal for quickly constructing simulations to perform analysis on these unique vehicles. Figure 36 shows a basic kinematic diagram of a tractor pulling two trailers connected by articulation joints. This model can be quickly assembled in the simulation space using the chassis and suspension SuperBlocks to automatically assemble the basic equations to model a B-Double road train configuration. In MapleSim, this model will look like Figure 37.

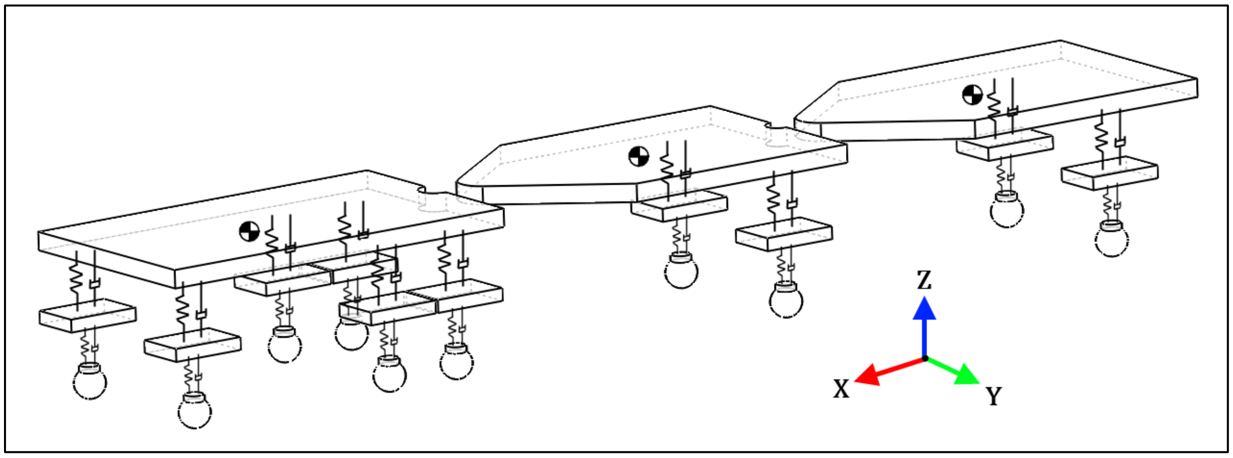


Figure 36 - Kinematic Diagram of a Simple B-Double LCV

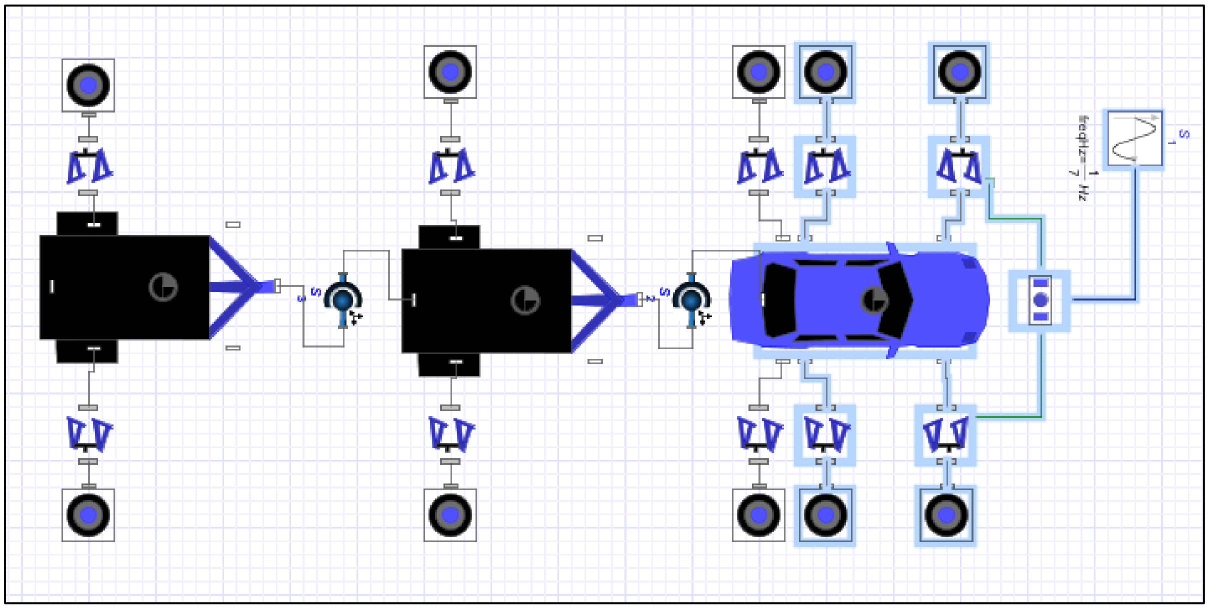


Figure 37 – Workspace diagram of a LCV

Reviewing the chassis SuperBlock parameters, modifying the chassis to support 3 axles in a LCV model is simple as clicking a few checkboxes and adding more suspension and tire SuperBlocks into the workspace. The trailers are the same chassis SuperBlocks with a different icon to communicate what vehicle it represents. With MapleCar, the initial setup of these large, complex simulations becomes a trivial procedure. A simple analysis that is relevant in these types of articulated vehicles is investigating off-track error that occurs during low speed maneuvers due to the geometry of the trailers connected by their points of articulation.

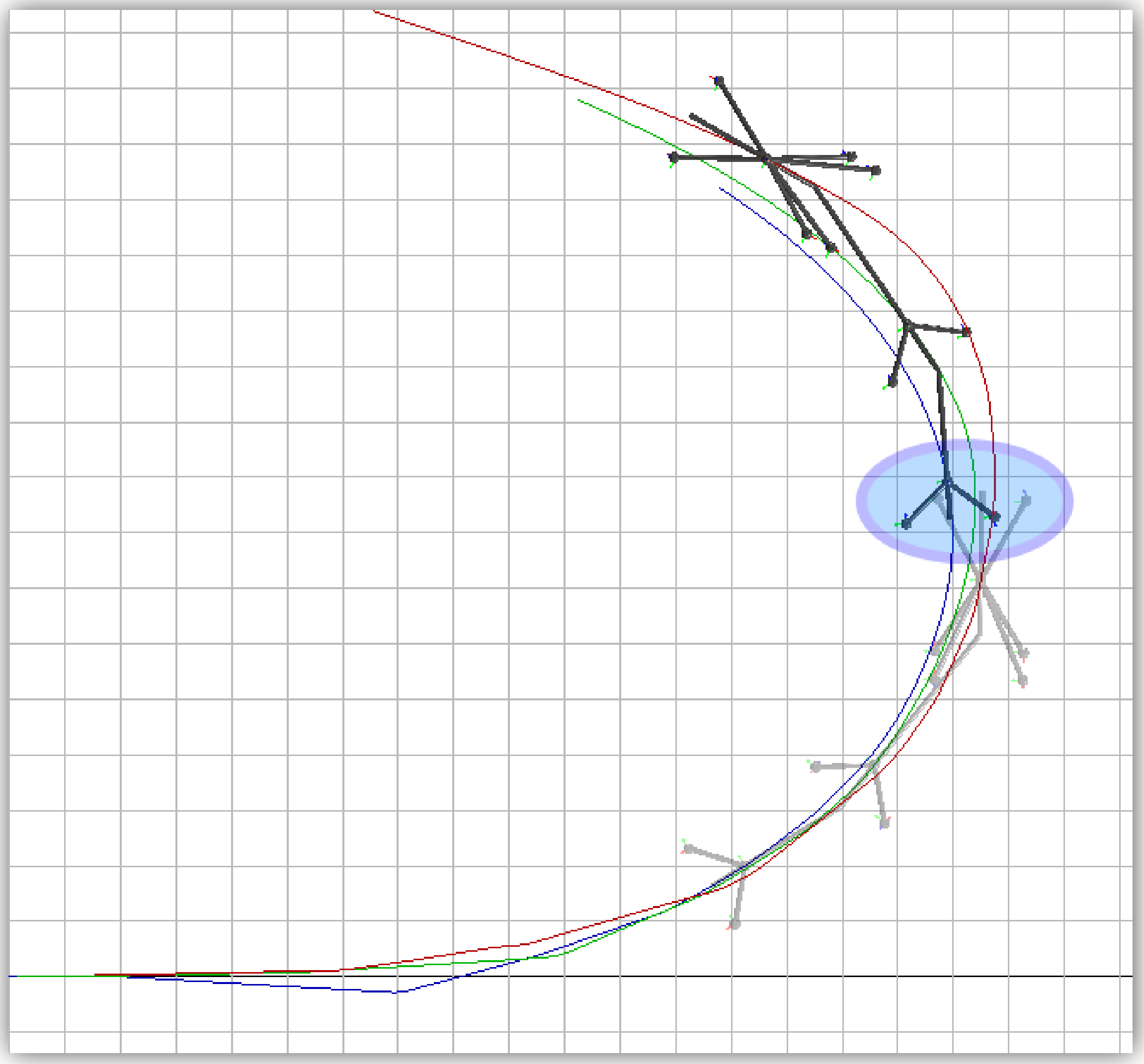


Figure 38 - Example trajectory of a long combination vehicle

Figure 38 shows the global trajectory of a basic long-chain vehicle from the top. The tractor features 3 axles, one in the front and 2 in the rear. Connected behind it is the primary trailer measuring 4 meters long. Behind the primary trailer is the secondary trailer, measuring 2.7 meters. The figure shows an overlay of two snapshots of the result when a sine steer input is applied. Off-tracking error will always occur since the trailers are being pulled forward without steering. In this short example, a point of concern is the unexpected drift of the trailer when turning left. This is the reason trucks make such wide turns; the tractor may clear the corner just fine, but the trailer drags closer into the turn and may invade the space of another vehicle. The total simulation time of this model is only 140 seconds.

Chapter 6

6. Conclusions and Recommendations

The software landscape surrounding the software used in automotive engineering is well established, but still contains plenty of room for improvement. A large problem is the stagnation of innovation regarding the software tools available to engineers; there has been little effect to curb the trade-off between speed and accuracy. On one hand, CarSim is great to quickly drop in a vehicle model but will always suffer from the discretization and numerical error due to curve fit equations and look-up tables. On the other hand, a multi-body analysis tool like MSC.Adams can provide a great deal of accuracy numerically solving the dynamics of a model, but will greatly suffer from inefficiently solving large, complex models.

The purpose of MapleCar is to take the initial step in creating a new modelling environment that is not trapped in modifying old methodology to solve new problems. Modelica provides a platform to apply innovative programming techniques into a simulation tool without sacrificing the use of analytical equations. The flexibility and modularity of components opens the door to utilizing a single simulation tool to model a wide variety of vehicles, instead requiring a user to continually extend a chain of tools or incorporate add-ons onto their existing ones. The SuperBlocks provide convenience to engineers using MapleSim in two important ways. First, the customizable, high-level blocks provide an immediate solution to assist a user into seeing result from the simulation. It is very intimidating for anybody to begin creating a functional vehicle model with no prior saved sub-systems and no immediate visualization. Instead of the handful of examples provided by MapleSim that must be cut and pasted together, a collection of modular components can snap together and a minimum result can be observed. From there, it becomes much easier to modify the provided parameters to simulate a particular scenario reducing the amount of effort to work with the software. The second advantage MapleCar contributes is better handling and packaging of completed models to be shared and distributed. Difficulty can arise when a user must manually keep track of which file contains which model containing which sub-systems. The task to assemble a new simulation from a collection of files resembles programming using notepad and pasting snippets of code from other text files. MapleCar provides the first step in creating an integrated development environment to ease the modelling of vehicle systems. With this topological organizational structure combined with MapleSim, the graph-theoretic methods applied during a simulation to automatically reduce and simplify the model equations provides superior efficiency compared to other simulation tools.

This exciting new perspective on vehicle modelling has great potential. Large, easy to use blocks are simply placed and connected in the MapleSim workspace, and the advanced multibody solution method utilizing graph-theoretic methods quickly provides a simplified collection of system equations. This can then be paired with the many features of Maple to perform advanced, analytical analysis on the equations governing the vehicle. These equations can also be deployed in various real-time targets using various connectors and compilers. The possibilities of this software are limitless, but boundaries and structure must be defined within the open sand box of MapleSim for it to continue to grow.

Working with the MapleSim, it is clear that the free manipulation of Modelica was not originally the focus; Modelica was used as a specialized vehicle to facilitate their unique solution methods. MapleSim6 contains enough compatibility with Modelica to import and execute the raw Modelica code developed in this work. However, compatibility must drastically increase for this type of package development to proceed. Modelica by itself has the essential features to refine the user-friendliness and polish the SuperBlock components, but a huge amount of distress surrounds importing Modelica code into the MapleSim environment. The continuation of this idea ultimately hinges on the contribution and involvement with MapleSoft. In particular, MapleSim should continue to work towards increasing its compatibility with Modelica. Without any access to the selected models using the *models with replaceable models* technique, the usability of this tool for real analysis quickly diminishes.

The presentation of many aspects of MapleSim and MapleCar is also an issue that needs to be addressed. First, without any graphical visualization of CAD geometry inside some of the SuperBlocks, the ability to demonstrate and present the different vehicle models during simulation using the MapleCar package is not attractive. A strong advantage MapleCar holds over other vehicle modelling software is the flexibility and modularity of its components to quickly draft and model different vehicle types and topologies. Graphs and charts will never properly reflect the visualization of a morphing sedan, compact car, truck or highway vehicle in the visualized result.

In the end, the initial value of MapleCar will be dependent on the breadth of models available to a new user. Therefore, it is important to continue the use of both MapleSim and MapleCar to model different components and sub-systems of a variety vehicle models. The libraries that exist in the various SuperBlock palettes have their beginnings as regular MapleSim models. Without increasing this collection of prepared, MapleCar will only exist as an empty framework. Moreover, the involvement of engineers not involved with the immediate development of MapleCar is important. Continuous feedback on the usability of the tool is critical in the success of this idea. This work is ultimately designed to benefit the engineers and researchers involved in the modelling and simulation of vehicles. Without their constant input and participation, it will be difficult to pitch a product that is not even used by its creators.

References

[1] B. D. Solomon and A. Banerjee, “A global survey of hydrogen energy research, development and policy,” *Energy Policy*, vol. 34, no. 7, pp. 781–792, May 2006.

[2] K. Çağatay Bayindir, M. A. Gözüküçük, and A. Teke, “A comprehensive overview of hybrid electric vehicle: Powertrain configurations, powertrain control techniques and electronic control units,” *Energy Conversion and Management*, vol. 52, no. 2, pp. 1305–1313, 2011.

[3] “Adams for Multibody Dynamics.” [Online]. Available: http://www.mscsoftware.com/Products/CAE-Tools/Adams.aspx. [Accessed: 03-Dec-2012].

[4] MSC Software, “Adams - Multibody Dynamics for Functional Virtual Prototyping.” .

[5] M. S. Fallah, R. Bhat, and W. F. Xie, “New model and simulation of Macpherson suspension system for ride control applications,” *Vehicle System Dynamics*, vol. 47, no. 2, pp. 195–220, 2009.

[6] J. J. Zhu, “UWSpace: Study of Vehicle Dynamics with Planar Suspension Systems (PSS),” Doctor of Philosophy, University of Waterloo, 2011.

[7] “CarSim Overview,” *CarSim Overview*. [Online]. Available: http://www.carsim.com/products/carsim/index.php. [Accessed: 03-Dec-2012].

[8] “Multi-Engineering Modeling and Simulation - Dymola - CATIA - Dassault Systèmes,” *Dymola*. [Online]. Available: http://www.3ds.com/products/catia/portfolio/dymola. [Accessed: 03-Dec-2012].

[9] “Modelica,” *Modelica Standard Library*. [Online]. Available: https://build.openmodelica.org/Documentation/Modelica.html. [Accessed: 03-Dec-2012].

[10] Modelica Association, “Modelica Language Specification Version 3.3.” [Online]. Available: https://www.modelica.org/documents/ModelicaSpec33.pdf. [Accessed: 03-Dec-2012].

[11] “Modelica Vehicle Dynamics Library.” [Online]. Available: http://www.modelon.com/products/modelica-libraries/vehicle-dynamics-library/. [Accessed: 08-Jan-2013].

[12] “MapleSim – High Performance Physical Modeling and Simulation – Technical Computing Software,” *MapleSim*. [Online]. Available: http://www.maplesoft.com/products/maplesim/. [Accessed: 03-Dec-2012].

[13] J. McPhee, “On the use of linear graph theory in multibody system dynamics,” *Nonlinear Dynamics*, vol. 9, no. 1, pp. 73–90, 1996.

[14] “MapleSim Tire Library - Maplesoft.” [Online]. Available: http://www.maplesoft.com/products/toolboxes/tire_component/. [Accessed: 03-Dec-2012].

[15] “MapleSim^{TM} Driveline Library.” [Online]. Available: http://www.maplesoft.com/products/toolboxes/driveline/. [Accessed: 03-Dec-2012].

[16] H. Peng, “Configuration, Sizing and Control of Power-Split Hybrid Vehicles,” *Proc. ICMEM*, 2007.

[17] A. Khajepour, A. Goodarzi, and A. Kasaiezadeh, “MapleCar Steps ahead,” MapleSoft Waterloo, 18-Nov-2011.

[18] K. Chau and Y. Wong, “Overview of power management in hybrid electric vehicles,” *Energy Conversion and Management*, vol. 43, no. 15, pp. 1953–1968, Oct. 2002.

[19] D. L. Harkey, F. M. Council, and C. V. Zegeer, “Operational characteristics of longer combination vehicles and related geometric design issues,” *Transportation Research Record: Journal of the Transportation Research Board*, vol. 1523, no. 3, pp. 22–28, 1996.

[20] I. Åkerman and R. Jonsson, “European Modular System for road freight transport–experiences and possibilities,” *TFK report*, 2007.

[21] M. Tiller, *Introduction to Physical Modeling With Modelica*. Springer, 2001.

[22] R. A. Hess and A. Modjtahedzadeh, “A control theoretic model of driver steering behavior,” *Control Systems Magazine, IEEE*, vol. 10, no. 5, pp. 3–8, 1990.

[23] R. Sharp, D. Casanova, and P. Symonds, “A mathematical model for driver steering control, with design, tuning and performance results,” *Vehicle System Dynamics*, vol. 33, no. 5, pp. 289–326, 2000.

[24] Y. Furukawa, N. Yuhara, S. Sano, H. Takeda, and Y. Matsushita, “A Review of Four-Wheel Steering Studies from the Viewpoint of Vehicle Dynamics and Control,” *Vehicle System Dynamics*, vol. 18, no. 1–3, pp. 151–186, 1989.

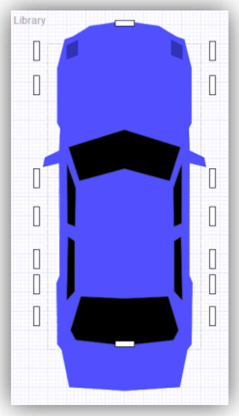
[25] P. Blow, J. Woodrooffe, and P. Sweatman, “Vehicle Stability and Control Research for U.S. Comprehensive Truck Size and Weight (TS & W) Study,” SAE International, Warrendale, PA, 982819, Nov. 1998.

Appendix A – Chassis SuperBlock source code

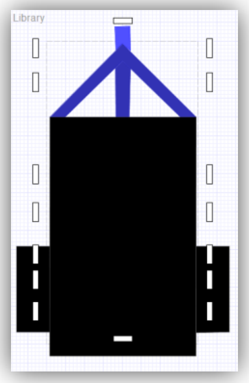
The chassis SuperBlock is the generalized component used inside the MapleCar modelling package. It features an extensive list of options and parameters to transform this component into any desired chassis model.

Source Code

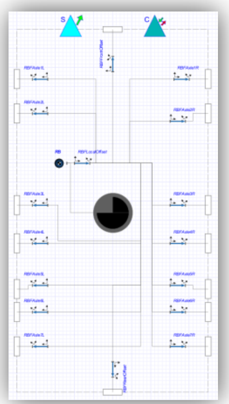
```
package MChassis

package Interfaces
end Interfaces;

package Icons

  // ///////////////////////////////
  // // ChassisIcon2
  // // This model contains the icon of a ‘car’ defined in Modelica basic annotations
```



```
 partial model ChassisIcon2
  annotation(
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}}),
      graphics={
        Polygon(
          points={{0,461}, {90,447}, {136,419}, {156,412}, {175,338}, {175,194},
          {168,180}, {170,121}, {208,103}, {213,79}, {170,89}, {169,-206}, {177,-
          226}, {175,-386}, {164,-400}, {144,-498}, {0,-507}, {-144,-498}, {-164,-400
          }, {-175,-386}, {-177,-226}, {-169,-206}, {-170,89}, {-213,79}, {-
          208,103}, {-170,121}, {-168,180}, {-175,194}, {-175,338}, {-156,412}, {-136,419
          }, {-90,447}, {0,461}},
          lineColor={0,0,255},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={80,80,255},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={ {122,411}, {151,396}, {151,361}, {122,372}, {122,411} },
          lineColor={30,30,200},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={50,50,180},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={ {-122,411}, {-151,396}, {-151,361}, {-122,372}, {-122,411} },
          lineColor={30,30,200},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={50,50,180},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={ {0,176}, {146,130}, {118,42}, {0,74}, {-118,42}, {-146,130}, {0,176} },
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={0,0,0},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={{128,24}, {150,87}, {149,-69}, {129,-78}, {128,24}},
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={00,0,0},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={{130,-116}, {148,-104}, {151,-271}, {130,-247}, {130,-116}},
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={00,0,0},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={{-128,24}, {-150,87}, {-149,-69}, {-129,-78}, {-128,24}},
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={00,0,0},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={{-130,-116}, {-148,-104}, {-151,-271}, {-130,-247}, {-130,-116}},
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={00,0,0},
          fillPattern=FillPattern.Solid),
        Polygon(
          points={{0,-266}, {114,-261}, {141,-349}, {128,-375}, {0,-389}, {-128,-375}, {-
          141,-349}, {-114,-261}, {0,-266}},
          lineColor={75,75,75},
          linethickness=0.5,
          smooth=Smooth.None,
          fillColor={00,0,0},
          fillPattern=FillPattern.Solid)
      })
     );
  end ChassisIcon2;

  // ////////////////////////////
  // // TrailerIcon
  // // This model contains the icon of a 'trailer' defined in Modelica basic annotations
```



```
partial model TrailerIcon
  annotation(
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}}),
    graphics={
      Polygon(
        points={{-20,439},{20,440},{21,375},{0,394},{-18,377},{-20,439}},
        lineColor={0,0,255},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={80,80,255},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{-192,200},{0,394},{195,200},{153,200},{0,349},{-152,200},{-192,200}},
        lineColor={30,30,200},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={50,50,180},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{-19,330},{0,349},{21,329},{18,200},{-18,198},{-19,330}},
        lineColor={30,30,200},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={50,50,180},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{-161,165},{164,165},{163,-401},{-162,-401},{-161,165}},
        lineColor={75,75,75},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={0,0,0},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{-192,200},{195,200},{194,-431},{-193,-431},{-192,200}},
        lineColor={75,75,75},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={0,0,0},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{195,-141},{282,-141},{282,-368},{195,-369},{195,-141}},
        lineColor={75,75,75},
        linethickness=0.5,
        smooth=Smooth.None,
        fillColor={0,0,0},
        fillPattern=FillPattern.Solid),
      Polygon(
        points={{-280,-141},{-193,-141},{-193,-368},{-280,-368},{-280,-141}},
        lineColor={75,75,75},
        linethickness=0.5,
          smooth=Smooth.None,
          fillColor={0,0,0},
          fillPattern=FillPattern.Solid)
        }
      )
    );
  end TrailerIcon;
end Icons;
package Library
/* Insert any specific sprung, chassis models here */
end Library;

// //////////////////////////////
// // ChassisLibraryFrameTest2
// // contains the active SuperChassis model without an icon
```



```
model ChassisLibraryFrameTest2

  /* initial Conditions and Mass Properties */
  inner parameter Real GlobalOffset = {0, 0, 0} annotation (Dialog(tab="CG Properties"));
  inner parameter Real InitialPosition = {0, 0, 0.75} annotation (Dialog(tab="CG Properties"));
  inner parameter Real InitialVelocity = {25, 0, 0} annotation (Dialog(tab="CG Properties"));
  inner parameter Modelica.SIunits.Mass CGMass = 2077 annotation (Dialog(tab="CG Properties"));
  inner parameter Modelica.SIunits.Inertia CGInertia[3,3] = {{1330, 0, 110}, {0, 1925, 0}, {
    110, 0, 1925}} annotation (Dialog(tab="CG Properties"));

  /* SUPERCOMPONENT FRAME OPTIONS */
  parameter Boolean DisableAxle1 = true annotation(Evaluate=true, Dialog(tab="Frame Options"));
  parameter Boolean DisableAxle2 = false annotation(Evaluate=true, Dialog(tab="Frame Options"
    ));
  parameter Boolean DisableAxle3 = true annotation(Evaluate=true, Dialog(tab="Frame Options"));
  parameter Boolean DisableAxle4 = true annotation(Evaluate=true, Dialog(tab="Frame Options"));
  parameter Boolean DisableAxle5 = true annotation(Evaluate=true, Dialog(tab="Frame Options"));
  parameter Boolean DisableAxle6 = false annotation(Evaluate=true, Dialog(tab="Frame Options"
    ));
  parameter Boolean DisableAxle7 = true annotation(Evaluate=true, Dialog(tab="Frame Options"));
  parameter Boolean DisableFrontArticulation = true annotation(Evaluate=true, Dialog(tab=
    "Frame Options"));
  parameter Boolean DisableRearArticulation = true annotation(Evaluate=true, Dialog(tab=
    "Frame Options"));
  inner parameter Real AxleDim1 = {0, 0, 0} annotation (Dialog(enable=not DisableAxle1));
  inner parameter Real AxleDim2 = {1.45, -0.8, -0.4} annotation (Dialog(enable=not
    DisableAxle2));
  inner parameter Real AxleDim3 = {0, 0, 0} annotation (Dialog(enable=not DisableAxle3));
  inner parameter Real AxleDim4 = {0, 0, 0} annotation (Dialog(enable=not DisableAxle4));
  inner parameter Real AxleDim5 = {0, 0, 0} annotation (Dialog(enable=not DisableAxle5));
  inner parameter Real AxleDim6 = {-1.3, -0.8, -0.4} annotation (Dialog(enable=not
    DisableAxle6));
  inner parameter Real AxleDim7 = {0, 0, 0} annotation (Dialog(enable=not DisableAxle7));
  inner parameter Real FrontADim = {1.6, 0, 0} annotation (Dialog(enable=not
    DisableFrontArticulation));
  inner parameter Real RearADim = {-1.75, 0, 0} annotation (Dialog(enable=not
    DisableRearArticulation));
  parameter Boolean isCGFrame = true annotation (Dialog(tab="Frame Options"));
  parameter Boolean isSensorFrame = false annotation (Dialog(tab = "Frame Options"));
  parameter Boolean isControlFrame = false annotation (Dialog(tab = "Frame Options"));

  // FRAMES
  TVSMbeta.Connectors.SensorFrame frame_sensor if isSensorFrame annotation (Placement(
    transformation(extent={{-145,375},{-75,445}})));
  TVSMbeta.Connectors.ControlFrame frame_control if isControlFrame annotation (Placement(
    transformation(extent={{75,375},{145,445}})));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle1L if not DisableAxle1
    annotation(Placement(transformation(origin={-230.0,332.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle1R if not DisableAxle1
    annotation(Placement(transformation(origin={230.0,332.5},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle2L if not DisableAxle2
    annotation(Placement(transformation(origin={-230.0,242.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle2R if not DisableAxle2
    annotation(Placement(transformation(origin={230.0,242.5},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle3L if not DisableAxle3
    annotation(Placement(transformation(origin={-230.0,0},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle3R if not DisableAxle3
    annotation(Placement(transformation(origin={230.0,0},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle4L if not DisableAxle4
    annotation(Placement(transformation(origin={-230.0,-100},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle4R if not DisableAxle4
    annotation(Placement(transformation(origin={230.0,-100},extent={{-25.0,-25.0},{25.0,25.0}},
    rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle5L if not DisableAxle5
    annotation(Placement(transformation(origin={-230.0,-212.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle5R if not DisableAxle5
    annotation(Placement(transformation(origin={230.0,-212.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle6L if not DisableAxle6
    annotation(Placement(transformation(origin={-230.0,-278.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle6R if not DisableAxle6
    annotation(Placement(transformation(origin={230.0,-278.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle7L if not DisableAxle7
    annotation(Placement(transformation(origin={-230.0,-362.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameAxle7R if not DisableAxle7
    annotation(Placement(transformation(origin={230.0,-362.5},extent={{-25.0,-25.0},{25.0,25.0
    }},rotation=0)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameFront if not
    DisableFrontArticulation annotation(Placement(transformation(origin={0,405},extent={{-25.0,-
    25.0},{25.0,25.0}},rotation=-90)));
  public Modelica.Mechanics.MultiBody.Interfaces.Frame_b frameRear if not
    DisableRearArticulation annotation(Placement(transformation(origin={0,-435},extent={{-25.0,-
    25.0},{25.0,25.0}},rotation=90)));
  public TVSMbeta.Connectors.CGFrame frameCG if isCGFrame annotation(Placement(transformation(
    origin={0,0},extent={{-60.0,-80.0},{60.0,40.0}},rotation=0)));

  // COMPONENTS
  public Maplesoft.Multibody.Bodies.RigidBody RB(Mass=1, Inertia={{1, 0, 0}, {0, 1, 0}, {0, 0,
    1}}, MechTranTree=Maplesoft.Multibody.Selectors.ICHandling.Guess, InitPos=InitialPosition,
    VelType=Maplesoft.Multibody.Selectors.VelocityFrame.Inboard, InitVel=InitialVelocity,
    MechRotTree=Maplesoft.Multibody.Selectors.ICHandling.Ignore, RotType={1, 2, 3}, InitAng={-.0
```

```
    , .0, .0}, AngVelType=Maplesoft.Multibody.Selectors.AngularVelocityFrame.Euler, InitAngVel={
    0, 0, 0}) annotation(Placement(transformation(origin={45.0,422.5},extent={{-20.0,-20.0},{
    20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFLocalOffset(InitPos=GlobalOffset,
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={115.0,372.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=-90)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFFrontOffset(InitPos=FrontADim, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={215.0,
    482.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=-270)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFRearOffset(InitPos=RearADim, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={235.0,
    242.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=-90)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle1R(InitPos=AxleDim1, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0,
    542.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle1L(InitPos=AxleDim1.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={65.0,522.5},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle2R(InitPos=AxleDim2, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={345.0,
    372.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle2L(InitPos=AxleDim2.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={65.0,292.5},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle3R(InitPos=AxleDim3, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0,
    172.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle3L(InitPos=AxleDim3.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={95.0,162.5},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle4R(InitPos=AxleDim4, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0,
    52.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle4L(InitPos=AxleDim4.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={105.0,52.5},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle5R(InitPos=AxleDim5, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0
    ,-70},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle5L(InitPos=AxleDim5.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={105.0,-70},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle6R(InitPos=AxleDim6, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0
    ,-170},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle6L(InitPos=AxleDim6.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={105.0,-170},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle7R(InitPos=AxleDim7, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0}, {0, 0, 1
    }}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={355.0
    ,-250},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  public Maplesoft.Multibody.Bodies.RigidBodyFrame RBFAxle7L(InitPos=AxleDim7.*{1, -1, 1},
    RSelect=Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat={{1, 0, 0}, {0, 1, 0},
    {0, 0, 1}}, RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin
    ={105.0,-250},extent={{20.0,-20.0},{-20.0,20.0}},rotation=0)));
  Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_S if isSteerable annotation (Placement
    (transformation(
    extent={{15,-20},{-15,20}},
    rotation=90,
    origin={-65,-110})));
equation
  connect(RB.frame_a, RBFLocalOffset.frame_a) annotation(Line(points={{65.0,422.5},{115.0,
    422.5},{115.0,392.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFFrontOffset.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{215.0,462.5
    },{215.0,336.5},{115.0,336.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFRearOffset.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{235.0,262.5
    },{235.0,342.5},{115.0,342.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle2L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{85.0,292.5},{
    115.0,292.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle2R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{325.0,372.5},{
    298.0,372.5},{298.0,316.5},{115.0,316.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle1L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{85.0,522.5},{
    148.0,522.5},{148.0,336.5},{115.0,336.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle1R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{335.0,542.5},{
    298.0,542.5},{298.0,316.5},{115.0,316.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle1L.frame_b, frameAxle1L) annotation(Line(points={{45.0,522.5},{28.0,522.5},{
    28.0,532.5},{-2.0,532.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle2L.frame_b, frameAxle2L) annotation(Line(points={{45.0,292.5},{15.0,292.5},{
    15.0,442.5},{-5.0,442.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFFrontOffset.frame_b, frameFront) annotation(Line(points={{215.0,502.5},{215.0,
    540.5},{225.0,540.5},{225.0,584.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle1R.frame_b, frameAxle1R) annotation(Line(points={{375.0,542.5},{411.0,542.5
    },{411.0,532.5},{455.0,532.5}},color={95,95,95},smooth=Smooth.None));
  connect(RB.frame_a, frameCG) annotation(Line(points={{67.0,419.5},{165.0,419.5},{165.0,399.5
    },{255.0,399.5},{255.0,332.5},{450.0,332.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle2R.frame_b, frameAxle2R) annotation(Line(points={{365.0,372.5},{406.0,372.5
    },{406.0,442.5},{455.0,442.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle3L.frame_b, frameAxle3L) annotation(Line(points={{75.0,162.5},{-5.0,162.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle3R.frame_b, frameAxle3R) annotation(Line(points={{375.0,172.5},{455.0,172.5
    }},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle3L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{115.0,162.5},{
    129.0,162.5},{129.0,337.5},{115.0,337.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle3R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{335.0,172.5},{
    115.0,172.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle4L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{125.0,52.5},{
    125.0,352.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle4R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{335.0,52.5},{
    115.0,52.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle4L.frame_b, frameAxle4L) annotation(Line(points={{85.0,52.5},{-5.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle4R.frame_b, frameAxle4R) annotation(Line(points={{375.0,52.5},{455.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFRearOffset.frame_b, frameRear) annotation(Line(points={{235.0,222.5},{235.0,112.5
    },{230.0,112.5},{230.0,-0.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle5L.frame_b, frameAxle5L) annotation(Line(points={{85.0,52.5},{-5.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle5R.frame_b, frameAxle5R) annotation(Line(points={{375.0,52.5},{455.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle5L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{85.0,292.5},{
    115.0,292.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle5R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{325.0,372.5},{
    298.0,372.5},{298.0,316.5},{115.0,316.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle6L.frame_b, frameAxle6L) annotation(Line(points={{85.0,52.5},{-5.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle6R.frame_b, frameAxle6R) annotation(Line(points={{375.0,52.5},{455.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle6L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{85.0,292.5},{
    115.0,292.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle6R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{325.0,372.5},{
    298.0,372.5},{298.0,316.5},{115.0,316.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  connect(RBFAxle7L.frame_b, frameAxle7L) annotation(Line(points={{85.0,52.5},{-5.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle7R.frame_b, frameAxle7R) annotation(Line(points={{375.0,52.5},{455.0,52.5}},
    color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle7L.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{85.0,292.5},{
    115.0,292.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None));
  connect(RBFAxle7R.frame_a, RBFLocalOffset.frame_b) annotation(Line(points={{325.0,372.5},{
    298.0,372.5},{298.0,316.5},{115.0,316.5},{115.0,352.5}},color={95,95,95},smooth=Smooth.None
    ));
  annotation(
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}})),
    Diagram(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,300.0
      }}))
    );
end ChassisLibraryFrameTest2;

// ////////////////////////////
// // ChassisLibraryVehicle
// // contains the same ChassisLibraryFrameTest2 but also contains a 'car' icon
model ChassisLibraryVehicle
  extends TVSMbeta.MChassis.Icons.ChassisIcon2;
  extends TVSMbeta.MChassis.ChassisLibraryFrameTest2;
  annotation(
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}})),
    Diagram(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}}))
    );
end ChassisLibraryVehicle;

// ////////////////////////////
// // ChassisLibraryVehicle
// // contains the same ChassisLibraryFrameTest2 but also contains a 'trailer' icon
model ChassisLibraryTrailer
  extends TVSMbeta.MChassis.Icons.TrailerIcon;
  extends TVSMbeta.MChassis.ChassisLibraryFrameTest2;
  annotation(
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}})),
    Diagram(coordinateSystem(preserveAspectRatio=true, extent={{-200,-400},{200.0,400.0}}))
    );
end ChassisLibraryTrailer;

package Examples
end Examples;

end MChassis;
```

Appendix B – Suspension SuperBlock source code

The suspension SuperBlock is the generalized component used inside the MapleCar modelling package. It features the use of the advanced Modelica technique, *a model with a replaceable model*, which populates the models within the library package as a selectable parameter.

Source Code

```
package MSuspension

package Interfaces

  partial model MBInterface
    Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a1
      annotation (Placement(transformation(origin={-100,0}, extent={{-30,-  35},{30,35}})));
    Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a
      annotation (Placement(transformation(origin={100,0}, extent={{-30,-  35},{30,35}}))); //
    annotation(
      Icon(coordinateSystem(extent={{-100,-100},{100.0,100.0}})),
      Diagram(coordinateSystem(extent={{-100,-100},{100.0,100.0}}))
      );
  end MBInterface;

  partial model MBInterface2
    Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a1
      annotation (Placement(transformation(origin={-100,0}, extent={{-30,-  35},{30,35}})));
    Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a
      annotation (Placement(transformation(origin={100,0}, extent={{-30,-  35},{30,35}}))); //
    annotation(
      Icon(coordinateSystem(extent={{-100,-100},{100.0,100.0}})),
      Diagram(coordinateSystem(extent={{-100,-100},{100.0,100.0}}))
      );
  end MBInterface2;

end Interfaces;

package Icons
  partial model SuspensionIcon2
    annotation(
      Icon(coordinateSystem(preserveAspectRatio=true, extent={{-200,-200},{200.0,200.0}}),
        graphics={
          Polygon(
            points={{-63,175},{-57,170},{-47,175},{-41,170},{-102,27},{-109,37},{-117,28},{-
            122,37},{-63,175}},
            lineColor={30,30,200},
            linethickness=0.5,
            smooth=Smooth.None,
            fillColor={50,50,180},
            fillPattern=FillPattern.Solid),
          Polygon(
            points={{-44,158},{129,101},{104,68},{-97,38},{-78,79},{67,95},{-56,132},{-44,
            158}},
            lineColor={30,30,200},
            linethickness=0.5,
            smooth=Smooth.None,
            fillColor={50,50,180},
            fillPattern=FillPattern.Solid),
          Polygon(
            points={{-77,-27},{-71,-33},{-63,-25},{-59,-34},{-113,-159},{-119,-152},{-129,-
            157},{-132,-150},{-77,-27}},
            lineColor={30,30,200},
            linethickness=0.5,
            smooth=Smooth.None,
            fillColor={50,50,180},
            fillPattern=FillPattern.Solid),
          Polygon(
            points={{-65,-48},{66,-77},{66,-93},{73,-105},{86,-105},{92,-96},{93,-82},{109,-
            87},{97,-127},{-108,-150},{-95,-120},{57,-102},{-74,-73},{-65,-48}},
            lineColor={30,30,200},
            linethickness=0.5,
            smooth=Smooth.None,
            fillColor={50,50,180},
            fillPattern=FillPattern.Solid),
          Polygon(
            points={{67,61},{95,66},{94,-7},{131,-10},{128,-36},{94,-35},{92,-96},{86,-105
            },{73,-105},{66,-93},{67,61}},
            lineColor={75,75,75},
            linethickness=0.5,
            smooth=Smooth.None,
            fillColor={00,0,0},
            fillPattern=FillPattern.Solid)
        }
      )
    );
  end SuspensionIcon2;
end Icons;

package Library
  partial model SuspensionInterface
    extends TVSMbeta.MSuspension.Interfaces.MBInterface2;
    outer parameter Integer MirrorRorL;
    outer parameter Real Stiffness;
    outer parameter Real Damping;
    Modelica.Mechanics.Translational.Interfaces.Flange_b frame_F7 annotation(Placement(
      transformation(origin={-110,-80},extent={{-15.0,-15.0},{15.0,15.0}},rotation=0)));
    annotation(
      Icon(coordinateSystem(extent={{-100,-100},{100.0,100.0}})),
      Diagram(coordinateSystem(extent={{-100,-100},{100.0,100.0}}))
      );
  end SuspensionInterface;

  /* LINEAR SPRING DAMPER - SIMPLE COMPONENT - OFFSET INCLUDED */
  model d_LinearSD_Offset
    extends Library.SuspensionInterface;
    /* IMPORTS */
    import Modelica.Constants.inf;
    import Pi=Modelica.Constants.pi;
    import pi=Modelica.Constants.pi;
    /* PARAMETERS */
    inner parameter Real Ks = Stiffness;
    inner parameter Real Lo = .12;
    inner parameter Real Kd = Damping;
    /* Offset */
    public Maplesoft.Multibody.Bodies.RigidBodyFrame WCOffset (InitPos={0, 0, -Lo}, RSelect=
      Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat=[1, 0, 0; 0, 1, 0; 0, 0, 1
      ], RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={
      620.0,10.0},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
    /* MAIN BODY */
    public Maplesoft.Multibody.Joints.Prismatic P(TranAxis=
      Maplesoft.Multibody.Selectors.UnitVector.posZ, Kspring=Ks, L0=Lo, Kdamper=Kd,
      MechTranTree=Maplesoft.Multibody.Selectors.ICHandling.Guess, InitPos=Lo, InitVel=0)
      annotation(Placement(transformation(origin={70.0,70.0},extent={{-20.0,-20.0},{20.0,20.0
      }},rotation=90)));
  /* EQUATIONS */
  equation
    connect(frame_a1, WCOffset.frame_b) annotation(Line(points={{36.0,378.9774475097656},{
      16.0,378.9774475097656},{16.0,375.9774475097656},{-1.0,375.9774475097656}},color={95,95,
      95},smooth=Smooth.None));
    connect(WCOffset.frame_a, P.frame_b) annotation(Line(points={{36.0,378.9774475097656},{
      16.0,378.9774475097656},{16.0,375.9774475097656},{-1.0,375.9774475097656}},color={95,95,
      95},smooth=Smooth.None));
    connect(P.frame_a, frame_a) annotation(Line(points={{70.0,50.0},{70.0,29.0},{75.0,29.0
      },{75.0,0.0}},color={95,95,95},smooth=Smooth.None));
    annotation(
      Diagram(coordinateSystem(preserveAspectRatio=true, extent={{0,0},{150.0,150.0}}),
        graphics),
      Icon(coordinateSystem(preserveAspectRatio=true, extent={{0,0},{200.0,200.0}}),
        graphics={Rectangle(extent={{0,0},{200.0,200.0}}, lineColor={0,0,0}),Rectangle(
        extent={{0.0,0.0},{200.0,200.0}})})
      );
  end d_LinearSD_Offset;

/* LINEAR SPRING DAMPER WITH STEERING - SIMPLE COMPONENT WITH IDEAL STEERING - OFFSET
INCLUDED */
model d_LinearSDSteering_Offset
  extends Library.SuspensionInterface;
  /* IMPORTS */
  import Modelica.Constants.inf;
  import Pi=Modelica.Constants.pi;
  import pi=Modelica.Constants.pi;
  /* PARAMETERS */
  inner parameter Real Ks = Stiffness;
  inner parameter Real Lo = .12;
  inner parameter Real Kd = Damping;
  inner parameter Real TieRodRatio = 1;
  /* Offset */
  public Maplesoft.Multibody.Bodies.RigidBodyFrame WCOffset (InitPos={0, 0, -Lo}, RSelect=
    Maplesoft.Multibody.Selectors.RotationMatrixType.Euler, RMat=[1, 0, 0; 0, 1, 0; 0, 0, 1
    ], RotType={1, 2, 3}, InitAng={0, 0, 0}) annotation(Placement(transformation(origin={
    620.0,10.0},extent={{-20.0,-20.0},{20.0,20.0}},rotation=0)));
  /* MAIN BODY */
  public Maplesoft.Multibody.Joints.Prismatic P(TranAxis=
    Maplesoft.Multibody.Selectors.UnitVector.posZ, Kspring=Ks, L0=Lo, Kdamper=Kd,
    MechTranTree=Maplesoft.Multibody.Selectors.ICHandling.Guess, InitPos=Lo, InitVel=0)
    annotation(Placement(transformation(origin={70.0,70.0},extent={{-20.0,-20.0},{20.0,20.0
    }},rotation=90)));
  public Maplesoft.Multibody.Joints.Revolute R1(RotAxis=
    Maplesoft.Multibody.Selectors.UnitVector.posZ, Kspring=0, Ang0=0, Kdamper=0, MechRotTree
    =Maplesoft.Multibody.Selectors.ICHandling.Ignore, InitAng=.0, InitAngVel=0) annotation(
    Placement(transformation(origin={139.5,165.5},extent={{-20.0,20.0},{20.0,-20.0}},
    rotation=-90)));
  public Modelica.Mechanics.Translational.Components.IdealGearR2T IGR2T1(useSupportR=false
    , useSupportT=false, ratio=TieRodRatio) annotation(Placement(transformation(origin={79.5
    ,125.5},extent={{-20.0,-20.0},{20.0,20.0}},rotation=-90)));
    /* EQUATIONS */
equation
  connect(frame_a1, WCOffset.frame_b) annotation(Line(points={{36.0,378.9774475097656},{
    16.0,378.9774475097656},{16.0,375.9774475097656},{-1.0,375.9774475097656}},color={95,95,
    95},smooth=Smooth.None));
  connect(WCOffset.frame_a, P.frame_b) annotation(Line(points={{36.0,378.9774475097656},{
    16.0,378.9774475097656},{16.0,375.9774475097656},{-1.0,375.9774475097656}},color={95,95,
    95},smooth=Smooth.None));
  connect(P.frame_a, R1.frame_a) annotation(Line(points={{70.0,50.0},{70.0,29.0},{75.0,
    29.0},{75.0,0.0}},color={95,95,95},smooth=Smooth.None));
  connect(R1.frame_b, frame_a) annotation(Line(points={{70.0,50.0},{70.0,29.0},{75.0,29.0
    },{75.0,0.0}},color={95,95,95},smooth=Smooth.None));
  connect(IGR2T1.flangeR, R1.flange_b) annotation(Line(points={{69.5,135.5},{69.5,150.5},{
    118.5,150.5}},color={0,0,0},smooth=Smooth.None));
  connect(IGR2T1.flangeT, frame_F7) annotation(Line(points={{69.5,95.5},{69.5,80.5},{-1.5,
    80.5}},color={0,127,0},smooth=Smooth.None));
  annotation(
    Diagram(coordinateSystem(preserveAspectRatio=true, extent={{0,0},{150.0,150.0}}),
      graphics),
    Icon(coordinateSystem(preserveAspectRatio=true, extent={{0,0},{200.0,200.0}}),graphics
      ={Rectangle(extent={{0,0},{200.0,200.0}}, lineColor={0,0,0}),Rectangle(extent={{0.0,0.0
      },{200.0,200.0}}))
    );
  end d_LinearSDSteering_Offset;
end Library;

// /////////////////////////////////////////////////////////////////////
// // SuspensionLibrary - is the live superblock for Suspension systems
model SuspensionLibrary
  extends TVSMbeta.MSuspension.Interfaces.MBInterface;
  extends TVSMbeta.MSuspension.Icons.SuspensionIcon2;

  /* SUPERCOMPONENT ANNOTATION OPTIONS */
  replaceable model mySuspensionModel = MSuspension.Library.d_LinearSD_Offset constrainedby
    Library.SuspensionInterface;
  public mySuspensionModel mySuspensionModel_inst annotation(Placement(transformation(extent
    ={{-20,-20},{20,20}})));
  inner parameter Integer MirrorRorL = 1;
  inner parameter Real Stiffness = 5000000;
  inner parameter Real Damping = 200000;
  parameter Boolean isSteerable = false annotation (Dialog(group="SuperModel Options"),choices
    (checkBox=true));
  parameter Boolean isSensorFrame = false annotation (Dialog(group = "SuperModel Options"),
    choices(checkBox=true));
  parameter Boolean isControlFrame = false annotation (Dialog(group = "SuperModel Options"),
    choices(checkBox=true));
  TVSMbeta.Connectors.SensorFrame frame_sensor if isSensorFrame annotation (Placement(
    transformation(extent={{-170,130},{-130,180}})));
  TVSMbeta.Connectors.ControlFrame frame_control if isControlFrame annotation (Placement(
    transformation(extent={{130,130},{170,180}})));
  Modelica.Mechanics.Translational.Interfaces.Flange_b frame_S if isSteerable annotation (
    Placement(transformation(
    extent={{15,-15},{-15,15}},
    rotation=90,
    origin={-65,-110})));
/* EQUATIONS */
equation
  connect (mySuspensionModel_inst.frame_a, SuspensionLibrary.frame_a) annotation(Line(
    points={{92.0,110.0},{143.0,110.0}},color={0,0,127},smooth=Smooth.None)) ;
  connect (mySuspensionModel_inst.frame_a1, SuspensionLibrary.frame_a1) annotation(Line(
    points={{217.0,110.0},{263.0,110.0},{263.0,131.0},{310.0,131.0}})) ;
  connect (mySuspensionModel_inst.frame_F7, SuspensionLibrary.frame_S) annotation(Line(
    points={{217.0,110.0},{263.0,110.0},{263.0,131.0},{310.0,131.0}})) ;
  annotation (
    Diagram(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-200,-200},{200,200}})),
    Icon(coordinateSystem(
      preserveAspectRatio=true,
      extent={{-200,-200},{200,200}})));
end SuspensionLibrary;

package Examples
end Examples;

end MSuspension;
```