# Bidirectional Integration Of Geometric And Dynamic Simulation Tools

Chahé Adourian
Supervisor: Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science and Engineering (CSE program)

# ABSTRACT

Mechanisms to share information from Mechanical Computer Assisted Design (MCAD) to simulation model have been demonstrated using various approaches. However, in all cases the information sharing is unidirectional - from the MCAD to Multi-Body Systems (MBS) simulation - which lacks the bidirectional mapping required in a concurrent engineering context where both models need to develop in parallel while remaining consistent.

We present a modelling library and a model mapping that permits and encourages parallel development of the mechanical assembly in both the MBS simulation and MCAD environments while supporting both bidirectional initial full transfer and incremental updates. Furthermore, with the adopted approach and with a careful selection of the simulation language, MCAD parts can be extended with non-mechanical behaviour in the simulation tool.

# ABRÉGÉ

Des mécanismes pour partager l'information entre un modèle CAD et un modèle de simulation ont été démontrés utilisant divers approches. Pourtant, dans tous les cas, le partage d'information était unidirectionnel - allant du modèle CAD vers le modèle de simulation - donc ne possédant pas les qualités bidirectionnelles nécessaires dans le contexte de l'ingénierie collaborative ou les modèles doivent rester consistants en permanence.

Nous présentons notre librairie de modélisation et notre développement des transformations entre modèles qui permettent et encouragent le développement parallèle de l'assemblage mécanique dans les deux environnements de simulation et de conception CAD. Notre approche supporte le partage et la synchronisation des modèles dans les deux sens et de faon incrémentale si nécessaire. En complément, avec l'approche adopté, les modèles mécaniques peuvent tre associés a des modèles comportementales non mécanique dans l'outil de simulation.

# DEDICATION

This document is dedicated to the graduate students of the McGill University and all of those that pursue science and technology for the betterment of their fellow people and the planet.

# ACKNOWLEDGEMENTS

I would like to thank my supervisor Hans Vangheluwe who introduced me to the Modelica language and the fascinating subject of Modelling and Simulation many years ago. His obvious fascination and interest in the subject inspired me to start and complete this thesis. It is thanks to the efforts of dedicated people like Hans and others that the field of modelling and simulation will be making great progress in years to come. I am thankful for the opportunity and appreciated the patience and advice that I was given over the course of this thesis. I would also like to thank Florence St-Amand and Leo Hartman, colleagues from the Canadian Space Agency, for proof-reading and providing valuable comments on the document. Finally, I would like to thank the CSA for granting me a leave of absence and partial funding for the completion of the course requirements. Without it, this would have been a much longer process.

# Contents

# List of Figures

# List of Tables

# INTRODUCTION

In the development of complex systems, multiple views on the system-to-be-built are often used. These views typically consist of models in different formalisms. Different views usually pertain to various partial aspects of the overall system. In a multi-view approach, individual views will be less complex than a single model describing all aspects of the system. As such, multi-view modelling, like modular, hierarchical modelling, simplifies model development. Most importantly, it becomes possible for individual experts on different aspects of a design to work in isolation on individual views without being encumbered with other aspects. These individual experts can work mostly **concurrently**, thereby considerably speeding up the development process. This realization was the core of Concurrent Engineering [1]. This approach does however have a cost associated with it.

As individual view models evolve, inconsistencies between different views are often introduced. Ensuring consistency between different views requires periodic and concerted efforts from the involved model designers. In general, detecting inconsistencies and recovering from them is a tedious, error-prone and manual process. Automated techniques can alleviate the problem and this has been investigated in the Concurrent Engineering community over the last two decades [2].

We narrow the focus on a representative sub-set of the problem: consistency between geometric Mechanical Computer Assisted Design (MCAD) models of a mechanical system and the corresponding Multi-Body Systems (MBS) dynamics simulation models. Literature review on the subject shows that current integrations of MCAD and MBS simulation tools is unidirectional from the former to the latter: the user develops a mechanical assembly in MCAD tool and this model generates the corresponding MBS model.

Some of the limitation perceived in this unidirectionality are:

1. Using the MCAD tool to develop the mechanical assembly is more time-consuming than if we could use a MBS modelling library. The process of creating a MBS model is usually faster than creating a mechanical assembly in the MCAD environment. In the latter case geometry must be developed to provide the mechanical attach points whereas the first requires only the dimension parameters to position and direct the attach points.

2. The use of the MCAD model in the design process as the source of the dynamics model forbids or at the least discourages the use of the MBS tool as the tool where the mechanical model is first developed. The lack of a reverse (MBS $\rightarrow$ MCAD) mapping forces the modeller to duplicate any changes to the MBS model by creating an equivalent geometry. This must be carried out manually and is prone to cause confusion and generate errors.

3. The MBS model of a mechanical assembly can no longer be changed as the changes will be overriden on the next MCAD $\rightarrow$ MBS mapping or Modifications made in the dynamics model on the mechanical assembly must be recreated manually on the MCAD side.

Two usage scenarios are provided as example and which require a bidirectional model transformation capability:

- A mechanical assembly starts as a MBS block diagram where body mass properties, joint locations and joint types are defined except for the geometric shape. Using the block diagram model, a skeleton MCAD model could be created.

- The MCAD model is used to generate the initial MBS simulation model. This model evolves and requires the MCAD model to be updated to remain consistent. This reverse mapping can be used to suggest changes to the MCAD model to recover consistency.

## Thesis objectives

The objective of this thesis is to explore how specialized MCAD modelling tools on the one hand and simulation tools on the other can be combined to demonstrate parallel evolution and synchronization of their respective models. A bidirectional link must allow for the separate and parallel evolution of either MCAD or dynamics models with the capability to recover from inconsistencies. From this analysis a proof-of-concept bidirectional link must be demonstrated preferably through implementation combined with argumentation whenever the implementation is unavailable.

## Contribution

Mechanisms to share information from MCAD to simulation model have been demonstrated using various approaches. However, in all cases the information sharing is unidirectional - from the MCAD to MBS simulation - which lacks the bidirectional mapping required in a concurrent engineering context where both models need to develop in parallel while remaining consistent.

We present a modelling library and a model mapping that permits and encourages parallel development of the mechanical assembly in both the MBS simulation and MCAD environments while supporting both bidirectional initial full transfer and incremental updates. Furthermore, with the adopted approach and with a careful selection of the simulation language, MCAD parts can be extended with non-mechanical behaviour in the simulation tool.

## Overview

Chapter 1 introduces concurrent engineering as a process at the heart of any complex design process. Modelling integration and evolution are introduced as the mechanisms to manage design complexity and ensure harmonious and consistent design evolution. Automating the various links between models is argued as the crucial element required for an efficient consistency recovery method between models. Finally Mechatronics design is introduced as a representative example of a concurrent engineering process and will be used as the framework where we make the contribution of this thesis.

Chapter 2 introduces the subject of modelling and simulation of physical systems. The fundamentals of simulating Differential Equations (DE) and discrete systems is explained. Causal and a-causal modelling and simulation are described. A brief overview of a-causal modelling languages is provided and finally the Modelica language is described which will be used as the simulation language for this thesis.

Chapter 3 introduces MCAD modelling as well as all the elements relevant for integration with the dynamics model. The subject of mechanical systems modelling and simulation is discussed and particularly the details of the Modelica Mechanics library which we will be using is provided. Section 3.1 covers the literature review on integration between MCAD and Simulation. Section 3.2 covers the modelling the dynamics of mechanical systems using various simulation tools. Section 3.3 covers some of the most common MCAD or geometric tools and their constraints. Section 3.5 discusses current implementations of tools that combine the use of MCAD and dynamics simulation tools.

Chapter 4 discusses the contribution of this thesis.

Chapter 5 provides the thesis conclusion and future work.

# 1 CONCURRENT ENGINEERING

## 1.1 Introduction

In today's engineering world, emphasis is increasingly on reduced time to market and shorter development cycles. In any complex design, the system is decomposed into a multitude of subsystems and areas of concern in order to make the process more manageable. Specialised tools for each area of concern simplify the design process and alleviate the burden on the user. However, areas of concern to not stand isolated from each other and there will be a multitude of relations, constraints, dependencies between them. When the subsystems are developed independently, system integration sessions must be planned where the engineers and designers meet to discuss the interfaces between their various design and to ensure that all dependencies, relations and constraints are met. This integration process can be a very time-consuming and labour intensive process.

Another problem with design processes might stem from the approach used. For example, the traditional 'Waterfall Model' where the project moves through the requirements definition, design, implementation, verification and maintenance phases sequentially. The problem with this approach is that the various phases are carried through independently of the others. When in the requirements phase, little formal attention is given to the other phases. The requirements are written and then the design phase starts. By the time a problem is found in the design which would ideally require changes in the requirements, the project has progressed too far to do this efficiently and so the rework is considerable. The problem is amplified even more when a problem is found in the later phases that has an impact on activity carried in the earlier phases. The rework involved to correct such a problem is greatly increased.

Opposed to this is the Concurrent Engineering (CE) Model which encourages taking into considerations all phases of the project from the start. CE is a work methodology based on the parallelisation of tasks (i.e.. concurrently). It refers to an approach used in product development in which functions of design engineering, manufacturing engineering and other functions are integrated to reduce the elapsed time required to bring a new product to the market.

Figure 1.1 provides a graphical representation of the Waterfall and the CE models.

The advantage of the CE is apparent in the iterative process. All phases are exercised in quick iteration. As such the impact of later phases on earlier ones is detected quickly and the amount of rework minimised.

## 1.2 Unidirectional Versus Bidirectional Tool Integration

A similarity exists between the "Waterfall Model" and unidirectional tool-integration on the one-hand and the "CE Model" and bidirectional tool-integration on the other. In the waterfall model, the information travels mostly one way, up through the various project phases. In a unidirectional

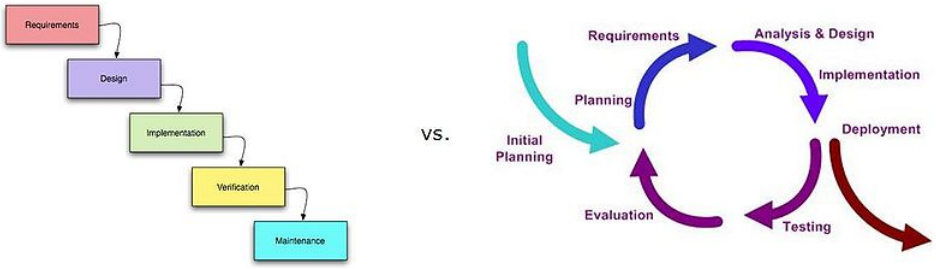



Figure 1.1: Waterfall versus iterative development methods [Wikipedia]

integration of tools, the information travels one way, from one tool to the next but never (or rarely) in the opposite direction. The process in either case discourages both early and frequent feedbacks. In the CE model, the information circulates. In bidirectional tool-integration, the information circulates back and forth. The process both enables and encourages feedback to occur quickly and frequently.

It is not enough then that a good process such as CE be selected, the tools used and specifically the tool-integrations available will play a crucial role in the successful and efficient completion of the project. Therefore if the CE model is to be successful, it is important that applications and models be integrated in a bidirectional fashion whenever possible.

The following sections will elaborate on unidirectional and bidirectional integration

### 1.2.1 Model integration and evolution

When integrating multiple models, when one model has information needed by another, and these models are evolving in parallel, it is crucial that mechanisms be provided that not only allow the importing of model information but also the updating of previously imported models.

The following example illustrates unidirectional integration without update versus unidirectional integration with update. For bidirectional integration, this capacity must be implemented on either side as both will receive information and require update at some point.

Integrated software application suites, products that integrate a large number of associated tools and functionality, try to provide the mechanisms for their various tools/models to combine into a coherent whole with consistency issues handled automatically or semi-automatically for the user. This is illustrated well by the bidirectional integration between a MCAD and Finite Element Analysis (FEA) tool exemplified by two commercial products: Solid Edge and ANSYS respectively.

The process of creating an FEA model starts with a source MCAD model on which various operations are applied in order to generate the finite-element mesh: specify material properties, insert new modelling elements between various parts of the CAD model, etc. The operations applied are usually very dependent on the shape of the CAD model which means that when the geometry changes some of these operations may no longer be applicable or would require appropriate adaptation and at the very least must be reapplied. In a non-integrated MCAD–FEA environment whenever an update is made to the MCAD model, the FEA would have to be recreated and all FEA operations reapplied. This rework will discourage the designer and the frequency of FEA analyses will decrease. As the delays between FEA analyses increases, the problems accumulating in the MCAD model will become greater such that the average cost of repairs increases as well. Most of these issues are avoided in an integrated MCAD–FEA environment such as possible with ANSYS [3]. ANSYS supports intelligent associative links between the source MCAD and the FEA models being manipulated. This maintains

the FEA synchronised with the latest MCAD model changes and avoids having to recreate the FEA model manually every time. This is an example implementation of intelligent data sharing between two modelling tools. In this case, the FEA tool detects changes to the underlying CAD model and determines how to reapply or modify the FEA generation rules with minimal or no user intervention.

### 1.2.2 Manual Versus Automated Consistency Recovery

Such cases are the exception rather than the rule. In most cases the user must carry out the synchronisation manually and perform this task repeatedly throughout the evolution of the models concerned. The manual process has many serious effects that not only slow down the design process, but also affect the way in which the design process itself is carried out. The first effect is that frequent synchronisation of the models is discouraged due to the effort required. This will reduce the total number of synchronisations, increase the number of discrepancies that have to be resolved at each iteration and, more importantly, lead to an increase in the per-discrepancy repair cost. It is cheaper on average to detect and repair a discrepancy early than late. Another effect is on the selection of the optimal design process, or design-path as we will call it. The cost of a given design-path is dependent on the cost of the individual processes and the frequency with which they are executed or used. If the cost of a given process changes, not only would the cost of the optimal design-path change, but that design-path may no longer be optimal. If a process is costly, the user will generally tend to avoid or minimise its use. As such automating a given process can have a considerable impact on the design process since it suddenly reduces the cost of that process. The optimal way to order and use the various tools and models may be affected and a new optimal design process may emerge.

### 1.2.3 Multi-model Integration And Consistency

Multiple models linked together to describe a single system will have intra-model and inter-model consistency rules defined. We will briefly discuss the relation between these consistency relations.

#### Inter-model consistency

Inter-model consistency rules define relations that must hold between the various models. These impose additional constraints on individual models in order to meet the system level constraints.

As an example of inter-model consistency, we take the geometric and dynamic models of two gears connected by a chain. Inter-model consistency could be defined in this case to be that the dynamic and geometric gear models agree on the individual gear mass, inertia and radii.

#### Intra-model consistency

Intra-model consistency rules are all rules particular to a given modelling tool to ensure the correctness of the model. Each modelling tool will usually come with its built-in consistency checks. For example a language compiler will verify that the language syntax is followed, that function calls obey the function declaration, etc. A MCAD tool will make sure that we cannot add assembly relations between two parts if that will result in an over-constrained system.

#### System model validity

We can extend the geometry model's internal consistency rules to require that there be no physical part interference under either static or dynamic conditions. Solid Edge can run these physical interference checks under both conditions. This is obviously specific to the geometry model view and not possible or appropriate within the dynamics modelling tool (Modelica). Therefore, although inter-model consistency hasn't changed, assuming that the two gears are too close and touching, with this new rule

our geometric model is no longer self-consistent.

Therefore, in order to have a valid system model we require that all models be consistent both individually as well as at the system level. Once both levels of consistency are satisfied we can proceed with the design of the system.

## 1.3 Mechatronics

Mechatronics (systems combining mechanics and electronics) is the subject that combines both modelling and simulation for mechanical systems including subservient components such sensors, actuators, controllers and other disciplines. In that respect it is a representative concurrent engineering problem. Robots are mechatronic systems composed of mechanical assembly with multiple rigid (or non-rigid) parts, connected together via joints of various types (revolute, prismatic, ...). The mechanical assembly itself determines the degrees of freedom of the robot, but is not enough to give it motion. Various actuators are then added in order to control the joints. Actuators might be electrical motors, hydraulic devices, electro-magnets, or any device that plays an active role. Sensors are used in order to provide the required feedback necessary for robust and precise operations. For example, a revolute joint could be equipped with angle measuring sensors. Prismatic actuators could be equipped with displacement sensors. The robot's grapple could be equipped with pressure sensors to avoid exerting too much force on objects being manipulated. The list is endless and limited only by the imagination of the designer. In order to have a functional system, something must command the actuators and receive feedback from the sensors. This is the system controller usually implemented through digital electronics, such as embedded computers running the control software or digital circuitry.

A multitude of engineering disciplines combine to create a mechatronic system. We have mentioned the mechanical assembly, electrical motors and interfaces, hydraulic mechanisms, various sensors and the control electronics. All the subsystems must be modelled properly for the system to be simulated accurately. The mechanical assembly dimensions, mass properties, joint types and positions must be up-to-date in the simulation for the control mechanisms to achieve intended objectives. When a control system is designed, the control logic makes assumptions about the system it is controlling. For example, given a robot shaped as a human arm, if commanded to move the tip of its hand a distance of one meter, it will do this by rotating the shoulder and elbow joints by a certain angle. If the control system is not aware of the actual dimensions of the arm, it will not achieve the commanded objective. The tip will travel the wrong distance and end up in the wrong position.

Similarly, if the arm is commanded to move at a certain angular speed, the controller must apply a certain amount of torque to achieve this. The centres of mass and inertia of the various arm components must be known to calculate the required torques. If these were higher than expected, then in consequence the applied torques will have to be higher as well. In terms of achieving the desired speed, feedback systems would usually take care of a mismatch between expected and real mass properties if still within the design margins. However, the power consumption that the engineer would have calculated to achieve these speeds would be incorrect since an increase in torque is associated with an increase in power consumption. An increase in torque, or power, would require more powerful and probably heavier motors. An increase of the motor's mass increases the robot mass, which in turn would require even bigger motors, etc. The electrical circuitry will have to be rated for higher currents, requiring thicker cables.

The purpose of these examples is to emphasise that in mechatronics many engineering disciplines (electrical, mechanical, electronics, etc.) can be intimately related and if design time is to stay low, the effort required to propagate shared information between all the subsystem models must be minimised.

## 1.4 Mechatronics and Concurrent Engineering

In this section, we concentrate on the effects of bidirectional integration or lack thereof on the efficiency of the development process for Mechatronics work.

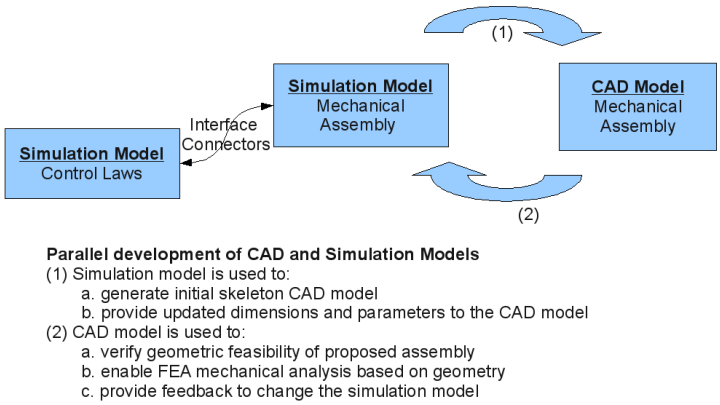



Figure 1.2: CAD and Simulation Co-evolution

In Mechatronics a similar problem exists between, on the one hand, the models used to describe and simulate the dynamic behaviour of a mechanical assembly, and on the other hand its geometric representation described using Computer Assisted Design (CAD) tools. The simulation tool helps define the mechanical system and its behaviour together with the environment model, actuators, sensors and controllers. The CAD tool helps define the geometry of each part, position the joints in each part and assemble the complete mechanical structure. With the material densities provided, the geometry is then used to determine the Mass properties (total mass, centre of gravity and inertia matrix) of a part and consequently that of the whole assembly structure. In general, a mechanical assembly is first conceived without consideration of detailed geometry. A corresponding simulation model is then developed to evaluate and optimise parameters such as mechanical dimensions, mass and inertia properties, topology, location and type of assembly joints, etc.

Figure 1.2 shows some of these relations and dependencies between the various models throughout the design process. On the left-hand-side, we find two components in the simulation model – the controller and the simulation mechanical assembly. The simulation mechanical assembly component and the controller share a clearly defined interface.

Once the simulation model mechanical assembly has matured, it can be used to establish a skeleton structure for the CAD model, seen on the right-hand-side. The CAD model is then developed further defining the geometry of each part. Once the geometry is available, various analyses are made possible including geometric interference detection, finite-element stress analysis, thermal simulation, etc. The results of these analyses in turn will usually lead to changes to the mechanical structure thus requiring updates to the simulation model mechanical assembly.

In other words, the two models are mutually dependent and complementary. The simulation model is better suited for:

- rapid model manipulation such as adding, removing and modifying parts and joints.
- the addition of complementary components such as the mechanical assembly controller, other subsystem models that interact with the mechanisms, etc.
- providing an appropriate environment for carrying out simulation result analyses.

Whereas the CAD model is better suited for carrying out geometry related/derived analyses such as:

- detecting geometric interference between the parts
- visualising and understanding the geometry and its potential effects
- help in geometrically placing the parts relative to each other
- provide better estimates of mechanical properties such as centre of mass, mass, inertia matrix, rigidity, etc.
- provide the geometry required for creating various FEA models: thermal, mechanical, ...

There is then a need to have both the simulation and CAD models evolve in parallel and semi-independently with consistency recovery operations interspersed in the process. The mechanical assembly simulation model can be modified quickly, whereas modifying a CAD model, even with parametrised dimensions is usually slower although current CAD tools have become quite efficient.

**Transition from dynamics to geometry** Before defining or refining the geometry of mechanical parts, for example in a car suspension system or a robot arm, extensive dynamic simulation is usually carried out using at first models that focus on the mechanical assembly structure without too much focus on the exact geometry. This includes part dimensions, joint locations, control systems and environment models. As the system matures, more attention is paid to the geometry of parts in part to make sure that the design is feasible, carry out various finite elements analyses (stress, etc), determine whether the design is geometrically feasible, and provide answers to any questions requiring knowledge of the geometry.

**Relation between Dynamics and Geometry** MCAD Tools used to define part geometry (Solid Works, Solid Edge, etc.) and the tools used to simulate the mechanical dynamics share a description of the mechanical assembly, mass properties of the parts, joint characteristics and locations. This information must remain consistent regardless of which tool, CAD or simulation, defines or modifies it. The simulation tool is better suited for defining the mechanical assembly dimensions and joint locations, except its geometry. This needs to feed into the CAD model as a starting point for defining the geometry further.

# 2

# MODELLING AND SIMULATION OF THE DYNAMICS OF SYSTEMS

*Simulation is the imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system* ***(Unknown author)***.

Computer simulation is widely used in engineering to carry out experiments. Engineers use computers to construct simulations of chemical, electrical, mechanical and all systems imaginable. In electronics, there are many specialised tools for the modelling and simulation of circuits to carry out analyses before any physical implementation. In civil engineering, the suitability of a future bridge for resisting high winds and hurricanes can be established using computer models that simulate the flow dynamics. Conditions difficult or expensive to realise using physical systems can be modelled in a computer simulation. The United States Department of Defense (DoD) uses super-computers to design and simulate nuclear bombs. In the automotive industry, engineers use computers to model and simulate every aspect of a car. Mechanical engineers develop models of the suspension, transmission system and engine of a car and subject it to various road conditions in order to optimise it. Motor performance, gas consumption, efficiency of the suspension system to absorb shocks and provide the required driving user-experience and many other performance metrics can be determined and improved.

## 2.1 Overview

This chapter introduces the subject of modelling and simulation of the dynamics of systems with particular focus on MBS. Section 2.2 provides a general mathematical overview of the major categories of dynamics simulation, namely continuous, discrete and hybrid. DE and particularly Differential Algebraic Equations (DAE) are introduced. The importance of DAE for the representations of physical systems and in particular MBS is emphasised. Section 2.3 describes the mathematical background for simulating physical systems. Subsection 2.3.1 discusses block-diagram causal modelling where the blocks calculate their outputs using their inputs, and the user must mainly determine the proper sequence of execution to correctly calculate the outputs. In contrast, subsection 2.3.2 introduces block-diagram a-causal modelling where the user provides the system equations (which are not the same as input/output blocks) and the simulation software must first solve the equations and then generate the required algorithm to calculate the solution. Section 2.4 introduces a multitude of languages and associated tools that support causal and/or a-causal modelling.

## 2.2 Continuous, Discrete and Hybrid Systems

In a Boeing 747 commercial passenger jet simulator (figure 2.1) the computer is creating a simulated replica of a very complex aircraft. At its core are computer models that simulate the behaviour of

Figure 2.1: Flight simulator's 6-degree of freedom hydraulics[Courtesy United Virtual]

the aircraft and many of the subsystems such as the aerodynamics, propulsion, electronics, etc. The user interface provides a simulated cockpit with all dials, controls and a simulated view out of the airplane cockpit. For all the complexity involved, we find two major categories of models: continuous and discrete.

### 2.2.1 Continuous Systems Simulation

To determine the dynamics of the aircraft body (position, velocity, acceleration) we need to determine the various forces that apply. The engineering field of aerodynamics enables us to determine these based on various aerodynamic factors such as wing shape, rudder and flap positions, engine thrust, airspeed vector, air density and a variety of other elements. We can translate these effects into the mathematical language of DE. We call these continuous since their states (position, velocity, ...) vary continuously with time.

Like the aircraft example, many systems can be represented mathematically using DE, Ordinary Differential Equations (ODE) or DAE. In particular, DAE play an important role in the description and representation of physical systems where algebraic constraints are present. We will discuss these briefly in the following sections.

#### 2.2.1.1 Differential and Ordinary Differential Equations

The behaviour, or equations of motion, of many physical systems can be specified using DE:

> *An equation containing the derivatives of one or more dependent variables, with respect to one or more independent variables, is said to be a differential equation (DE). [4]*

When the unknown function in a DE is a function of a single independent variable, then the DE can be called an ODE:

> *If $y$ is an unknown function $y : \mathbb{R} \to \mathbb{R}$ in $x$ with y(n) the nth derivative of $y$, then an equation of the form*
> $F(x, y, y', \ \ldots, \ y^{(n-1)}) = y^{(n)}$
> *is called an ODE of order $n$. For vector valued functions, $y : \mathbb{R} \to \mathbb{R}^m$, it is called a system of ordinary DE of dimension $m$.*
> *When a differential equation of order $n$ has the form*

$F\left(x, y, y', y'', \ \ldots, \ y^{(n)}\right) = 0$
*it is called an* ***implicit differential*** *equation whereas the form*
$F\left(x, y, y', y'', \ \ldots, \ y^{(n-1)}\right) = y^{(n)}$
*is called an* ***explicit differential*** *equation. [Wikipedia]*

Only a small portion of DE have explicit solutions. For this reason, numerical methods are widely used for calculating the behaviour of DE [5].

Derivatives are the basic components of DE defining the rate of change of a variable as a function of another and itself. For example, the speed variable is the derivative of the position variable as a function of time. Then, given the DE defining a system and its initial conditions, we can calculate the progression of the system by using the derivatives.

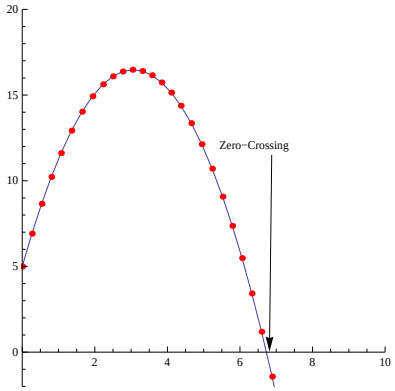



Figure 2.2: Continuous Ball Model

Figure 2.2 models the behaviour of the differential equation of a ball as a function of time. Notice that the ball did not bounce when it touched or crossed the floor (height zero) since we didn't include the bouncing behaviour of the full model. A discrete model will define what happens when the ball hits the floor: i.e. bounce. We will discuss this below. Note, that the bouncing could also have been implemented as a continuous motion but for a simple model a discrete model is satisfactory.

#### 2.2.1.2 Differential Algebraic Equations

*The systems of equations that govern certain phenomena (in electrical circuits, chemical kinetics, etc.) contain a combination of DE and algebraic equations. The DE are responsible for the dynamical evolution of the system, while the algebraic equations serve to constrain the solutions to certain manifolds. It is therefore of some interest to study the solutions of such differential-algebraic equations (DAEs).[6]*

DAE show up frequently in engineering problems. They occur whenever we combine DE and differential-free algebraic equations or constraints. There are many special formulations for DAE. The most general is in implicit form and is defined as follows:

$$x_j(t) \in \mathbb{R}^n \ where \ j = 1, \cdots n \tag{2.2.1}$$

$$f_i(t, x, x^{(1)}, x^{(2)}, \cdots) = 0, \ where \ i = 1, \cdots n \tag{2.2.2}$$

The $x_j$ in (2.2.1) are the time dependent state variables. The $f_i = 0$ in (2.2.2) are the set of DE and algebraic constraints, and the $x^{(1)}$, $x^{(2)}$ ... are the $1^{st}$, $2^{nd}$ and higher order derivatives of the vector $x$.

**Pendulum** Consider the motion of a pendulum in the x-y plane as seen in figure 2.3. The equation of motion can be written in DAE form using the DE (2.2.3) in addition to the algebraic constraints (2.2.4) as listed here:

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I_z \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} 0 \\ -mg \\ M \end{bmatrix} + \begin{bmatrix} F_x \\ F_y \\ -F_x l \cos\alpha - F_y l \sin\alpha \end{bmatrix} \tag{2.2.3}$$

$$\mathbf{g} = \begin{bmatrix} x - l\sin\alpha \\ y + l\cos\alpha \end{bmatrix} = \mathbf{0} \ . \tag{2.2.4}$$

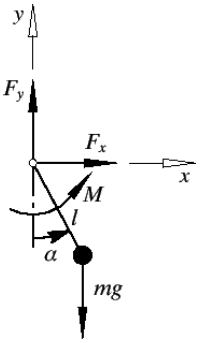



Figure 2.3: Pendulum

The pendulum consists of a mass $m$ attached on a rod of length $l$ where a torque $M$ is applied. Gravity $g$ applies a force along the $-y$ direction and the rod exerts a constraint force on the point mass decomposed into $F_x$ and $F_y$ components. $I_z$ is the inertia and $\alpha$ is the angle of the pendulum. Notice the following:

1. Equation (2.2.3) by itself is under-determined; not enough information is provided to calculate the time behaviour of the state variables $x$ and $y$.

2. Equation (2.2.4) is not a differential equation as there are no derivatives involved. This is what makes the system of equations a DAE.

Solving a DAE can then be considered as the process of solving an under-determined ODE problem, where the solution space at each point is reduced or projected onto the constraint manifold given by the algebraic constraints. This projection should produce the unique solution we are looking for if the DAE is well-posed.

**Applications to Robotics** Mechanical assemblies including robotic systems lend themselves well to modelling using DAE since the motion of the assembly under applied torques and forces is constrained by the joints linking the various parts. These mechanical constraints get translated into algebraic variables in the equations of motion and produce a DAE.

**Solution methods** Solving DAE is difficult and many open problems remain. Unlike an ODE where valid initial conditions are easier to find. In the case of DAE the algebraic variables are constrained and so finding valid initial conditions requires solving the constraints. Since the constraints can be arbitrarily complex, a generalised solution is not available. As such, simply finding a valid set of initial conditions may be difficult and computer-based tools currently in use are not always able to resolve them.

### 2.2.2 Discrete Events Simulation

Discrete event simulation deals with systems where the state variable follows a piecewise constant motion over time with its value changing discontinuously only at events. The states follow a piecewise discontinuous pattern where each state variable is constant between events and at event points the state values will change discontinuously if they change. The events therefore split the state variables into constant value segments.

### 2.2.3 Hybrid systems simulation

Hybrid systems' simulation integrate the behaviours of both continuous and discrete-time systems. A complete simulation of an aircraft must combine both the continuous behaviour of the aerodynamics as well as the discrete behaviour of the various modes of operation of its jet engine (off-idle-full thrust). The two models must be combined into one integrated simulation.

The bouncing ball example illustrates how discrete and continuous models come together. The ball in flight is described by a continuous model using classical mechanics. The bouncing event is described with a discrete model. The two combine to form a hybrid system simulation.

**Continuous model** The continuous part of the model is seen in figure 2.2.

**Discrete model** In order to simulate the ball bouncing off the floor, we first need to detect the ball **crossing** the zero-level (i.e. crossing through the floor) and then determine the time this occurred. This is required in order to avoid unrealistic behaviours such as bouncing before touching the floor or after going through the floor. The bounce must be executed at the moment where the ball touches the floor.

Figure 2.4 depicts a close-up of the ball's trajectory as it crosses through the floor. At the $N^{th}$ step of integration the ball is above the zero-point and then in the following step (N+1) it is below. The crossing signals the occurrence of the **hit the floor** event. The time is determined numerically and the continuous part of the simulation is interrupted. The discrete model is executed which models the ball bounce and this changes the velocity vector.

**Hybrid model** When we combine the previous continuous and discrete models and associated simulation engines, we get a single simulation where the ball exhibits both continuous (the parabolic trajectory) and discrete (the bounce) behaviours as seen in figure 2.5. The two simulations are integrated as follows. The continuous simulation verifies at each step whether the **hit the floor** event has occurred at which point it discards the current solution point, calculates the exact event time as well as the system state at that event then passes this information to the discrete model. The discrete model generates a new set of state variables (in this case, the velocity vector is modified) and passes the information to the continuous model which uses this as its starting point. The result is a plot of the ball incoming to hit the floor and bouncing off.

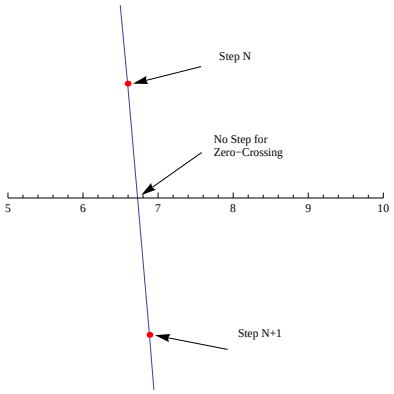



Figure 2.4: Ball Zero-Crossing

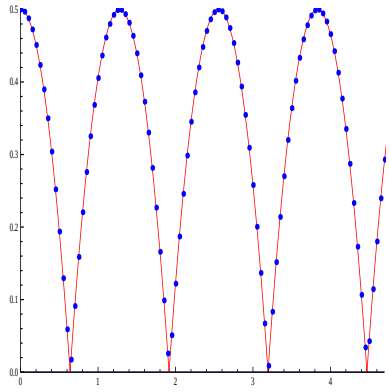


Figure 2.5: Hybrid system simulation: Simulation of a bouncing ball

## 2.3 Causal and A-causal modelling

In [7] the benefits of causal versus a-causal modelling are discussed.

### 2.3.1 Causal modelling

Causal modelling refers to models where the individual components behave in a causal manner. These components have inputs, outputs, state variables and parameters. The inputs together with the state variables and parameters determine the values of the outputs. These components can be chained together to form a directed graph and closed chains are allowed as long as a legal execution order exists.

Such models are suitable for modelling dynamic systems. Examples of causal modelling include Causal Block diagrams [8], Petri Nets [9] and Systems Dynamics [10]. Systems can be modelled as nodes representing system variables and connecting lines representing causal effects. The changing value of one variable can cause another to increase or decrease as described by equations. Causal Loop Diagrams (CLDs) are used to model dynamic systems. The simple diagram notation of nodes and lines, identifies the important variables in a system and how they interact.

A causal loop diagram (CLD) is a diagram that aids in visualising how interrelated variables affect one another. The diagram consists of a set of nodes representing the variables connected together. The

relationships between these variables, represented by arrows, can be labelled as positive or negative. Causal models frequently contain loops, describe feedback and have a static topology. An example of a system dynamics model is given in figure 2.6.

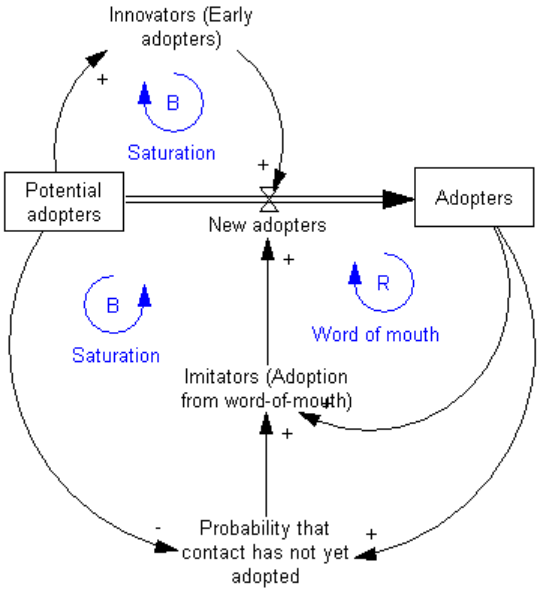



Figure 2.6: Causal Loop Diagram

Causal modelling is suitable for the modelling of mathematical algorithms. A mathematical algorithm is a sequence of mathematical operations executed in some sequence in order to solve a particular problem. Usually, the initial problem is not always cast in a procedural manner. Instead, the problem must be recast by the user into such a form. For example, if the mathematical problem is to solve the quadratic equation $ax^2 + bx + c = 0$ for a particular set of coefficients, then we can convert this to a procedural algorithm where the general solution is given by $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. This can be represented as a block diagram causal model as in figure 2.7.

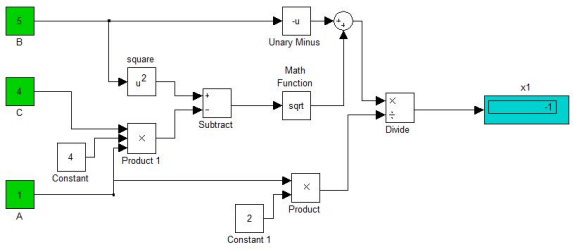



Figure 2.7: Causal Block Diagram Quadratic solution

### 2.3.2 A-Causal modelling

Unlike causal modelling where each model had a definite computational flow using its input to calculate its outputs, a-causal or non-causal modelling essentially means that models are defined in neutral form

without consideration of computational order. This enables models to be defined in a more general way simplifying the model development and maintenance tasks. A tool that supports a-causal modelling methods must therefore also carry out symbolic manipulation so the model equations can be re-ordered automatically when the model is run.

In causal modelling, the component models always have a direction associated with each of their interface ports: input or output. The input ports receive information and the output ports transmit them.

In a-causal modelling, ports can also be direction-less. Instead of receiving or transmitting information, they specify a shared constraint between the components connected to that port.

Proceeding through various representations of a simple electrical circuit is a useful method to understand the meaning of an a-causal connector (or port) and the differences between causal and a-causal modelling.

Figure 2.8 represents the a-causal model of an electric circuit. The solver of the a-causal modelling tool would take this block-diagram representation and assign to each component a set of equations and automatically generate the set of equations (2.3.1). From this representation the algorithmic solution (2.3.2) is automatically derived.

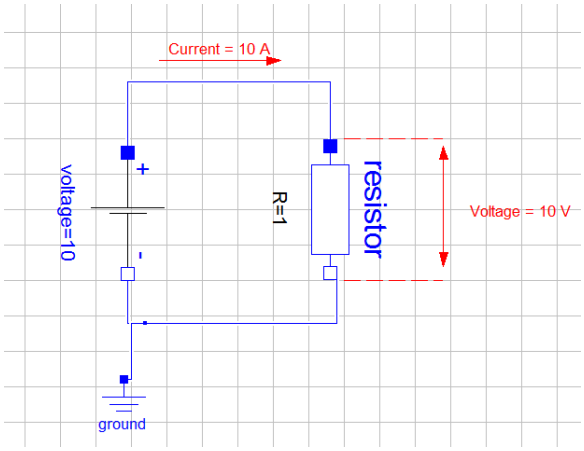



Figure 2.8: A-causal electric circuit model

$$V_{Battery} = 10A \tag{2.3.1a}$$
$$R = 1\Omega \tag{2.3.1b}$$
$$I_{Battery} + I_{Resistor} = 0 \tag{2.3.1c}$$
$$V_{Battery} - V_{Resistor} = 0 \tag{2.3.1d}$$
$$V_{Resistor} = R\ I_{Resistor} \tag{2.3.1e}$$

The equations do not explicitly provide the solution and instead they must be solved or a sequence of operations must be determined. The result is shown in the algorithm (2.3.2):

$$V_{Resistor} = V_{Battery} = 10V \tag{2.3.2a}$$

$$I_{Resistor} = \frac{V_{Resistor}}{R} = 10A \tag{2.3.2b}$$

$$I_{Battery} = -I_{Resistor} = -10A \tag{2.3.2c}$$

$$\tag{2.3.2d}$$

In a block-diagram causal modelling environment, we would implement this algorithm as seen in figure 2.9.

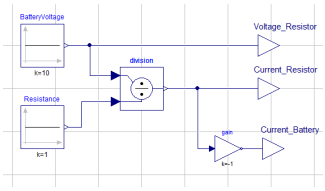



Figure 2.9: Causal model of electric circuit

On the left-hand we have the two relevant parameters of the circuit: battery voltage and resistor value. All other values are dependent on these two and their values are provided on the right-hand side output signals.

Notice that figure 2.8 uses the electrical representation used by electrical engineers and comparatively how unrelated figure 2.9 appears. For many physical systems, a-causal models provide the capacity to create models that have a clear resemblance with the physical system being modelled. This is less common with causal modelling.

In the following sections we provide an overview of various causal and a-causal modelling languages and simulation tools.

## 2.4 Simulation Languages and Tools

DAE are the foundation for formulating many engineering problems. Once the problem is formulated, there are a great many ways to proceed in implementing the solution. The DE can be manually derived resulting in a procedural solution which is then implemented in an appropriate programming language or tool. The DE in implicit form can also be passed directly to an application which then determines/generates the procedural solution. The former is the causal approach and the latter is the a-causal one. This latter solution can in turn be implemented in many different ways.

A-causal modelling and simulation was discussed by Hilding Elmqvist in his 1978 PhD dissertation [11] where he also developed the first version of Dymola. A great variety of tools that provide a-causal modelling capabilities have been developed since.

In the following sections we will provide a brief summary of selected causal and a-causal modelling tools in order of increasing capabilities and usability. We discuss the causal modelling tools first as they are the precursors to the a-causal ones.

Table 2.1 provides an overview of simulation tools and some of their important features.

Table 2.1: Simulation tools and features

| Features versus Tools | ACSLx | GAUSS | O-Matrix | MATLAB | Simscape | ODE | 20-Sim | AMESim | Dymola | Maple | MapleSim | SimulationX | Easy5 | Ecosimpro | Mathematica | MathModelica | Objectica | ObjectMath |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Modelling** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Procedural | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Equation based | Y | N | N | N | Y | N | Y | Y | Y | Y | Y | Y | N | Y | Y | Y | Y | Y |
| Block-Diagram Causal | Y | N | N | N | Y | N | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Block-Diagram A-causal | N | N | N | N | Y | N | Y | Y | Y | N | Y | Y | N | Y | N | Y | N | N |
| **Language** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Proprietary | Y | Y | Y | Y | Y | Y | Y | Y | N | Y | Y | Y | Y | Y | Y | N | Y | Y |
| Modelica | N | N | N | N | N | N | N | Y | Y | N | Y | Y | N | N | N | Y | N | N |
| **Libraries** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Control | Y | N | N | N | Y | N | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Electrical | N | N | N | N | Y | N | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Thermal | N | N | N | N | Y | N | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Fluid | N | N | N | N | Y | N | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Mechanics(1D) | N | N | N | N | Y | N | Y | Y | Y | N | Y | Y | N | Y | N | Y | N | N |
| Multi-body(3D) | N | N | N | N | Y | Y | Y | Y | Y | N | Y | Y | Y | Y | N | Y | N | N |
| Automotive systems | Y | N | N | N | Y | N | N | Y | Y | N | Y | Y | N | N | N | Y | N | N |
| **Multi-body** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Built-in | N | N | N | N | Y | Y | Y | N | Y | N | Y | Y | N | N | N | Y | N | N |
| Modelica Mechanics | N | N | N | N | N | N | N | N | Y | N | N | N | N | N | N | N | N | N |
| ADAMS (external) | N | N | N | N | N | N | N | Y | N | N | N | N | Y | N | N | N | N | N |
| Multi-body 3D Visualisation | N | N | N | N | Y | Y | Y | Y | Y | N | Y | Y | Y | N | N | Y | N | N |
| 3D Design User Interface | N | N | N | N | N | N | Y | Y | N | N | Y | Y | Y | N | N | N | N | N |
| **Interface with** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Code export | Y | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |

*continued on next page*

**Table 2.1 – continued from previous page**

| Features versus Tools | ACSLx | GAUSS | O-Matrix | MATLAB | Simscape | ODE | 20-Sim | AMESim | Dymola | Maple | MapleSim | SimulationX | Easy5 | Ecosimpro | Mathematica | MathModelica | Objectica | ObjectMath |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| import from Simulink | N | N | N | - | - | N | N | Y | N | Y | Y | Y | Y | N | N | N | N | N |
| export to Simulink | Y | N | N | - | - | N | Y | Y | Y | Y | Y | Y | Y | N | N | N | N | N |
| Simulink co-simulation | N | N | N | - | - | N | Y | Y | N | N | N | Y | Y | N | N | N | N | N |
| ADAMS export | N | N | N | N | Y | N | Y | Y | N | N | N | N | Y | N | N | N | N | N |
| Microsoft .NET | N | N | N | Y | Y | N | Y | Y | N | Y | Y | Y | Y | N | Y | Y | Y | Y |
| **Scripting** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Limited | | | | | | | | | Y | | | | | Y | | | | |
| Intermediate | | | | | | | Y | Y | | | | Y | Y | | | | | |
| Advanced | Y | Y | Y | Y | Y | | | Y | | Y | Y | | | | Y | Y | Y | Y |
| Pre-Processing Support | Y | Y | Y | Y | Y | N | Y | Y | N | Y | Y | Y | N | N | Y | Y | Y | Y |
| Post-Processing Support | Y | Y | Y | Y | Y | N | Y | Y | Y | Y | Y | Y | N | N | Y | Y | Y | Y |
| **Misc** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Interactive Web Publishing | N | N | N | Y | Y | N | N | N | N | Y | Y | N | N | N | Y | N | N | N |

### 2.4.1 Numerical Simulation Tools

Before the advent of specialised mathematical modelling and simulation tools, scientists, mathematicians and engineers created and made use of the FORTRAN language which simplified the process of expressing mathematical operations. FORTRAN offers a programming language geared towards mathematical operations.

To solve DAE systems, general solvers were written in the form of functions. The user usually provides the differential system to solve in the form of a function and passes it as an argument to the solver function as well as various other parameters including a valid set of initial conditions. Numerical methods are then used to calculate derivatives and integrate the solution. When initial conditions are not provided specialised library routines are available as well. An extensive set of such solvers is provided by the NetLib project (in C and FORTRAN) in its collection of mathematical functions [12] together with documentation.

Although FORTRAN as a mathematical language alleviated the difficulty of formulating mathematical problems, it is not the most convenient to use. More suitable languages and tools were needed to improve the situation.

#### 2.4.1.1 ACSL

The Advanced Continuous Simulation Language (ACSL) [13] (pronounced "axle") is a computer language designed for modelling and evaluating the performance of continuous systems described by time-dependent, nonlinear differential equations. ACSL is an equation-oriented language and support causal block diagram modelling using the ACSLx extension. An important feature of ACSL is its sorting of the continuous model equations, in contrast to general purpose programming languages such as FORTRAN where program execution depends critically on statement order.

#### 2.4.1.2 GAUSS

The GAUSS [14] Mathematical and Statistical System is a fast matrix programming language widely used by scientists, engineers, statisticians, biometricians, econometricians, and financial analysts. A wide variety of statistical, mathematical and matrix handling routines are available. Comprehensive plotting functions are provided.

#### 2.4.1.3 O-Matrix

O-MATRIX [15] is an easy-to-use technical computing environment and matrix-based scripting language. The high-performance integrated O-Matrix environment includes an extensive collection of mathematical, statistical, engineering and visualisation functions. The robust and diverse set of analysis functions enable the rapid development of complex, computationally intensive scientific, engineering, and technical computing solutions.

#### 2.4.1.4 MATLAB

MATLAB [16] is an interactive numeric programming tool with command shell based interaction. It is an easy-to-use integrated technical computing environment and array-based scripting language. Similar in function to O-Matrix, it is fundamentally a matrix-oriented numerical analysis tool. It provides a dynamically typed interpreted and imperative language with very powerful plotting capabilities, an extensive set of mathematical functions and excellent documentation.

### 2.4.2 Computer Algebra Systems

Whereas numerical simulation tools only deal with numbers, Computer Algebra Systems (CAS) provide an interactive environment to manipulate formulae symbolically and are increasingly used as **mathematical assistants**. This increases the range of problems that can be tackled.

MACSYMA [17] was the first comprehensive symbolic mathematics system and one of the earliest knowledge based systems; many of its ideas were later adopted by Mathematica [18], Maple [19] and other systems such as MuPad [20] and REDUCE [21]. MuPad, initially an independent product, is currently provided as a toolbox to MATLAB. Maple and Mathematica are some of the most popular CAS tools today. These tools all provide dynamically typed and interpreted programming languages. Imperative, functional and object-oriented programming styles are also available in some cases. Program packages for linear algebra, differential equations, number theory, statistics, and functional programming are available as a minimum.

MuPad, Maple and Mathematica come with an interactive graphic system that supports animations and transparent areas in 3D plotting. Maple and Mathematica provide the best capabilities. Plotting

support with MACSYMA and REDUCE is more basic and relies on the Linux gnuplot package for plotting support.

MuPAD, Mathematica and Maple provide support for object-oriented programming. This means that each object "carries with itself" the methods allowed to use on it. Objects hold their own data and methods. In the case of MuPad Overloading and inheritance support is also provided. Maple provides Object-oriented programming capabilities and parametric polymorphism. An example of parametric polymorphism with Maple is given in [22]. Mathematica provides Object Orientation. However, few if any examples exist of its use and expert knowledge of the language is required in order to benefit from these capabilities. [23] and [24] provide two such examples. Two third-party packages Objectica [25] and ObjectMath [26] exist that simplify or package this Object-Oriented functionality in a more convenient programming interface for Mathematica.

The object-oriented properties of these tools/languages would allow the development of **component-based models** (objects that hide as much as possible the intricacies of the code and expose an interface appropriate for engineering) but unfortunately there are only a few examples of products that make use of this and few libraries useful for engineering exist that could be found. This testifies to the inappropriate abstraction level provided by these languages for the easy creation of physical simulation libraries. In the next section, we will show how various initiatives try to improve this.

### 2.4.3 Component-Based A-causal Simulation Tools

Compared to imperative languages such as FORTRAN, C and C++, equation-based languages have an advantage in terms of the added flexibility and capabilities in the manipulation of equations. Imperative languages do not provide direct symbolic manipulation of equations and are incapable of operations such as exact differentiation, integration, equating variables or combining multiple equations. Support for symbolic equation manipulation is a first step towards the creation of component-based equation models and we covered such tools in the previous section.

Engineering problems that are based on DE can be directly described in equation-based tools. In order to simplify the problem formulation, the tools should provide a level of abstraction that goes beyond direct use of equations. Several tools and libraries that address this have been developed. 20-Sim, Easy5, AMESim, Ecosimpro, Simscape, Dymola, MathModelica, MapleSim, SimulationX are all such tools.

They all share the following features. They offer both a procedural and equation based language, block-diagram acausal component modelling support and model libraries in multiple engineering domains.

All these tools with the exception of AMESim provide support for 3D mechanical models and animation visualisation. AMESim makes use of ADAMS as a third-party integrated product for the same purpose.

These tools supports the use of components. This allows to enter models as in an engineering sketch: by choosing components from the library and connecting them, the engineering scheme is actually rebuilt

The building blocks are packaged in easily accessible application libraries. Users can also create custom libraries for reuse and sharing across the enterprise.

These are schematic (iconic blocks)-based virtual product development software used to model, simulate, and analyse multi domain dynamic systems characterised by differential, difference, and algebraic equations.

The systems that can be analysed include mechanical, electrical, hydraulic, pneumatic, thermal, gas dynamics, power train, vehicle dynamics, digital/analog control systems and much more. Models may be assembled graphically from special pre-built, ready-to-use multi-domain system-level blocks such

as valves, actuators, heat exchangers, gears, clutches, engines, pneumatics, flight dynamics, and many more, or from primitive functional blocks, such as summers, dividers, lead-lag filters, and integrators. The building blocks are packaged in easily accessible application libraries. Users can also create custom libraries for reuse and sharing across the enterprise.

The components of a system are described by analytical models representing the hydraulic, pneumatic, electric or mechanical behaviour of the system components.

Easy to use schematic based system-level modelling, simulation and analysis

#### 2.4.3.1 Simscape

Simscape [27] from Mathworks integrates with Simulink [28], which is a causal block-diagram graphical modelling and simulation environment integrated with the MATLAB tool, and provides physical-based modelling. Mechanical, hydraulic and electrical libraries are provided among others and its programming language allows the creation of new acausal block-diagram models.

#### 2.4.3.2 20-Sim

20-sim [29] supports a-causal modelling and uses a Maple-like language. Its libraries are mainly geared towards automotive simulation of all major subsystems including electric, hydraulic, thermal, Mechanical (1D, 2D, 3D Mechanics), Signal and Control.

#### 2.4.3.3 AMESim 1D-Lab

AMESim [30] is a simulation software for the modelling and analysis of one-dimensional (1D) systems. The software package offers a 1D simulation suite to model and analyse multi-domain, intelligent systems and to predict their multi-disciplinary performance. AMESim is a complete 1D virtual system analysis platform that allows users to design multi-domain systems

#### 2.4.3.4 Dymola

Dymola [31] is a simulation tool based on the Modelica language. The Modelica language is an object-oriented and hierarchical mathematical language appropriate for creating multi-domain components of physical systems. The Modelica Standard Library provides simulation models in subsystems including electric, hydraulic, thermal, Mechanical (1D, 2D, 3D Mechanics), Signal and Control. Dymola and SimulationX are one of two tools that currently support directly the Modelica.Mechanics.MultiBody library. Further details on the Modelica language can be found in section 2.4.5 and the Multi-body library in section 3.2.4.

#### 2.4.3.5 MapleSim

MapleSim [32] is based on the Maple mathematical package. Although Maple provides Object-Oriented capabilities such as those provided by ObjectMath and Objectica (since Maple 9.5 in 2004), they have recently come into the Modelica market with MapleSim and are making fast progress in catching up with competitors.

#### 2.4.3.6 SimulationX

SimulationX [33] is an advanced Modelica based modelling and simulation tool with extensive industrial user base. It also provides its own flavor of the Modelica language with completing features. The user can develop new models using either pure Modelica or with the SimulationX flavor. SimulationX also supports the Modelica.Mechanics.Multibody library as well as providing its own version of a multi-body library. Compared to Dymola, SimulationX boasts a better user interface, better support for

3D animation and a much faster compile/simulate process which makes debugging the model more convenient. It also possesses a greater variety of export and import capabilities of the simulation code. VB-Script programming language allows the user to automate any procedures that can be executed manually via the user interface.

#### 2.4.3.7 Easy5

Easy5 [34] was initially developed by Boeing for modelling aircraft systems. It is part of a suite of tools covering many aspects of engineering analysis.

#### 2.4.3.8 Ecosimpro

Ecosimpro by EA Internacional [35] is an equation-based modelling and simulation tool. It provides many advanced modelling libraries and is unique among similar tools to provide specific libraries for the aerospace and space industries.

#### 2.4.3.9 Mathematica-based tools

Although the Mathematica language provides object-like constructs based on lists (derived from the Lisp language), they are not easily understood except by expert programmers and this makes Object Oriented Programming (OOP) with Mathematica a difficult proposal.

Stephan Leibbrandt [25, 36] concisely describes the problem and solution of Mathematica when dealing with complex world problems:

> Mathematica has tremendous capabilities to solve problems by means of functional and imperative programming. But the bigger such a model gets, the more difficult is it to keep track of the program flow. This is caused by the fact that concise objects of the real world cannot be directly represented in the Mathematica language. Having object orientation at hand, you can intuitively map real-world problems onto the mathematical models. Additionally, modularisation and hierarchisation are byproducts that allow you to keep model units small and clear enough such that the overview does not get lost.

ObjectMath and Objectica are Mathematica based products that provide an object-oriented abstraction above the equations.

**MathModelica** MathModelica [37] is an implementation of Modelica based on the Mathematica CAS. It provides the user full support to modelling with the Modelica language. the Mathematica engine is used as the Modelica solver. Similar to MapleSim and Maple, the advantage of having an extremely sophisticated CAS behind MathModelica, it provides the Modelica user full access to Mathematica's capabilities. This is one area where Dymola lags behind as the scripting language is comparatively minimal.

**Objectica** Objectica [25] is a commercial third-party application that adds to Mathematica the paradigm of object orientation (since Mathematica 5.0 in 2005). It is seamlessly integrated into Mathematica without using any external package or programming language and yields full access to all capabilities of Mathematica without posing restrictions on the user. This includes the fact that all internal Mathematica symbols keep their meaning outside an object context. The four main paradigms of object orientation: abstract data types, inheritance, encapsulation, and polymorphism are supported. Unfortunately, there are no advertised commercially available libraries built on Objectica yet.

**ObjectMath** The ObjectMath [26, 38] language is an object oriented extension to the Mathematica computer algebra language, which provides mathematical notation and symbolic transformations. Built

as both a language and tool, it permits the user to define objects with clear interfaces (variables), parameters and equations that could be composed together to form more complex objects.

With ObjectMath, Mathematica functions and equations can be grouped into classes, in order to structure the mathematical model. Equations and functions can be inherited from general classes into more specific classes, which allows reuse within the mathematical model. Multiple inheritance is supported. This object oriented way of modelling is a natural way to describe physical systems. ObjectMath was designed by Peter Fritzson and his team at the PELAB (Programming Environment Lab), Linköping University, Sweden. Some of the new developments of ObjectMath are part of the Modelica modelling language design effort.

### 2.4.4 Tools and Features

Table 2.1 provides a resume of many of the tools we discussed and their features. The rows contains the features and the columns the tools. The tools are ordered by category. The first set are the numerical simulation tools, followed by the CAS and component-based simulation tools including multi-body tools.

The features are sorted by groups which include **Modelling**, **Language**, **Libraries**, **Multi-body**, **Interfaces**, **Scripting** and **Miscellaneous** features.

Here we will go into more details as to the meaning of each of these features:

**Modelling** Under the **Modelling** group we cover the types of modelling provided. The simplest is the **Procedural** textual programming capability which defines the sequence of actions to be executed. Equation-based support signifies that the language has the capacity to describe the problem using equations. The equations do not specify a sequence of execution and therefore the tool would have the capability to convert the equations into an algorithmic procedure to execute. Block-Diagram causal modelling covers tools that provide a block-diagram modelling capability where each block behaves as a function with inputs and outputs clearly defined. Block-Diagram a-causal modelling covers all the features of block-diagram causal modelling and provides in addition blocks that may have directionless connectors.

**Language** Under the **Language** group we specify whether the modelling language is proprietary to the tool or uses the Modelica language. Some tools provide both such as the case with AMESim and MapleSim.

**Libraries** Under the **Libraries** group we specify the types of libraries that come with the tool. This is not an extensive list but tries to cover some of the physical domains. There are two types of Mechanics libraries, including 1D which covers elements such as gears and pistons, and 3D which covers mechanical joints that have degrees of freedom that extend to the 3 dimensions. Under automotive library we cover libraries for a specific subsystem required to model vehicles. For example AMESim provides a very extensive set of libraries covering most subsystems whereas Modelica based tools have at the least the vehicle dynamics library which is part of the Modelica Standard library [39].

**Multi-body** The **Multi-body** group specifies how the multi-body simulation capability is implemented and some of its features. The first is the **Built-in** method where the tool does not use neither ADAMS or the Modelica language to implement it. This is the case for MapleSim and SimulationX where although both use the Modelica programming language, the Mechanics library itself is built with proprietary code. The **Modelica Mechanics** library which is part of the Modelica Standard Library is supported only by Dymola and not compatible with any of the other Modelica tools. This is because the library and the tool were developed for each other. Dymola offers some features that

are not part of the Modelica standard and the Modelica Mechanics library uses these. There are also some tools that are not self-sufficient to model the mechanics and rely on the excellent **ADAMS** tool which provides excellent modelling and simulation capabilities. When modelling multi-body systems, it is important to consider **3D visualisation** support. All tools that support multi-body modelling provide the means for visualising the mechanisms. Another important feature is the capacity to visualise the mechanism during modelling. This is listed under **3D Design User interface**. For example, among Modelica supporting tools Dymola and MathModelica support multi-body modelling but offer no user interface to visualise the model during the modelling phase. The user needs to complete the model, compile, simulate and use the animation display to see what has been constructed. Compared to this, MapleSim, SimulationX and ADAMS provide a user-interface where the mechanical assembly is carried out in a 3D environment. The user can add Parts, insert joints and even dynamically move the Parts in order to resolve the constraints. This is a crucial capability.

**Interfaces** The **Interfaces** group covers the interfaces that exist between the tool and external applications. The **Simulink** interfaces specify whether import from Simulink, export to Simulink or co-simulation with Simulink is provided. The **ADAMS** interface specifies whether the tool in question requires ADAMS for its operation. The **Microsoft .NET** interface specifies whether Microsoft's .NET programming languages can be used to interface with the tool from external programs.

**Scripting** The **Scripting** group covers the capabilities of the tool in terms of scripting before and/or after running the simulation model. This scripting capability would allow the creation and execution of models as well as manipulation the data that feeds into or results from the simulation. For example, Simscape, MapleSim and MathModelica are built on top of MATLAB, Maple and Mathematica respectively. These tools provide extensive scripting support. Comparatively, SimulationX and Easy5 come with some scripting capability but not nearly comparable in terms of capabilities. Therefore, we categorise the scripting capability of a tool as either **Limited**, **Intermediate** or **Advanced**. We also refer to **pre-processing** and **post-processing** support features if the tool either provides marked pre or post-processing capabilities either through its scripting language or additional analysis capabilities built in the tool.

**Miscellaneous** The **Miscellaneous** group covers various categorised features. The **Interactive web publishing** refers to the capability of publishing a particular model on the web while allowing the user to interact with it. For example Simulink models may be compiled as a web-application and published on the web. The user can then execute the model as well as modify simulation parameters.

### 2.4.5 Modelica Language

Modelica [40] is an equation-based hierarchical object-oriented physical modelling language that supports ODEs and DAEs designed to allow convenient, component-oriented modelling of complex systems, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented sub-components. The language is developed through an international effort [40]. It unifies and generalises previous object-oriented modelling languages (e.g. Dymola [41, 31], ObjectMath [38], Omola [42], etc.). The current Dymola application supports the new Modelica language. [43, 44, 45] provide a historical background in the development of the language together with a comprehensive coverage of the language components and structure. A comparison with the C++ and Java Object-Oriented languages are also provided.[46] provides an extensive reference manual for Modelica programming.

One of the driving forces in the creation of the Modelica language was the wish to integrate in one language models from multiple engineering disciplines. Modelica is currently one of the better known

object-oriented or component-oriented mathematical programming languages. There are numerous Modelica-based or Modelica-supporting tools including: Dymola, SimulationX [33], MathModelica [37], MapleSim [32], OpenModelica [47], LMS Imagine.Lab AMESim [30], SCICOS [48] and others. Also, CATIA Systems from Dassault Systèmes uses the Dymola kernel for simulation.

The language has been designed to allow tools to generate efficient simulation code automatically with the main objective to facilitate exchange of models, model libraries and simulation specifications. It allows defining simulation models modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The capability of physical modelling of multiple domains of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic etc, model components within the same application model. Further details on the language is provided in appendix A where the article *Modelica  A Unified Object-Oriented Language for System Modelling and Simulation* [45] by Peter Fritzson and Vadim Engelson is reproduced.

# 3

# MECHANICAL MODELLING AND SIMULATION

This chapter discusses the modelling and simulation of mechanical systems. The modelling aspects cover two domains: the geometry and the dynamics.

Section 3.1 provides a review of the state of the art.

Section 3.2 covers the use of equation-based dynamics modelling and simulation tools described in more detail in chapter 2 as well as some other tools for the modelling and simulation of mechanical systems. We call these MBS modelling and simulation languages and/or tools.

Section 3.3 covers some of the most common MCAD or geometric tools and their constraints.

Section 3.5 makes the connection between geometric assemblies and MBS modelling. In particular we elaborate on the relations that exist between the components of a geometric assembly and those in an MBS model.

Section 3.6 covers the creation of the geometry of mechanical objects or **Parts**, the composition of Parts (or **Assemblies** for short) and the use of geometric constraints between Parts. The common features of many MCAD tools are described by analogy by looking at one representative tool (Solid Edge) which is at the same time the tool we have selected for the thesis.

Section 3.7 describes the Modelica Mechanics library which we use as a starting point for constructing the MCAD to MBS associations.

## 3.1 Current State of the Art

MCAD tools that provide dynamics simulation capabilities come generally in two flavors: either the user specifies the parameters required to carry out the dynamics simulation from within the MCAD tool or the model of dynamics is exported as a black-box to an external tool such as Simulink [28] which can then carry out the simulation. In both cases, the model of the dynamics is generated by the tool and is non meant to be manipulated. The tool ADAMS [49] is a relevant example. In ADAMS, the user can drive the dynamics directly by locally specifying the forces and torques or first export the dynamics model to Simulink [28] and then specify forces and torques there.

The second approach is a considerable improvement over the first since we find the model in a general purpose modelling and simulation environment that provides both the capacity to integrate the mechanical system with other models as well as flexibility in defining and controlling forces and torques. For instance, in [50], ADAMS is used to draw the geometry and define the dynamics model, SABER [51] is used to define the electrical systems and Simulink provides the control systems and integrates as well as coordinates the execution of all these models.

An improvement over the previous approach is demonstrated in [52] where the dynamics model is no longer generated in the MCAD tool. Instead, a tool extracts the relevant mechanical assembly information from the MCAD model and using a mechanical library from the multi-domain simulation

language Modelica [53] defines the dynamics model. In a similar manner, Solid Works [54] MCAD models can be exported to Mathworks' SimMechanics [55]. In this approach, the user is free to fine-tune the individual components of the dynamics model free from the limitations of the MCAD tool. However, the process remains unidirectional such that changes in the Modelica model cannot be propagated back to the MCAD model.

[56] introduces the concept of combining both geometry (MCAD models) and behaviour (simulation models) of mechatronic system components into component objects. By composing these component objects, designers automatically create a virtual prototype of the system they are designing. Since the MCAD and behavioural models are now combined, their parameters need to be combined as well. For example, if the MCAD model represents an electric generator, then its associated electrical behavioural model should hold some relation with the mechanical size in determining the power it produces. This means the MCAD and behavioural models need to be parametrically linked providing some form of bidirectional interaction. Although an improvement over the previous approach, defining the mechanical topology is still completely done using the MCAD model.

[57] provides an overview of the relation between MCAD assembly constraints and the equivalent joints used in dynamics simulation. The contact geometry is analysed instantaneously and the resulting degrees of freedom extracted which in turn can be used in a dynamics model.

In [58] Sinha et al have shown that when rigid bodies are in contact, the kinematic degrees of freedom can be automatically derived from the nature of the contact. In current MCAD tools the user can specify the degrees of freedom between any two parts and these are encoded within the MCAD model. For the purposes of this thesis, we assume that is the case and we rely on the existence of this information to extract it directly.

In [59] a port-based modelling paradigm is introducing providing a mapping between a simple MCAD assembly with two parts connected by a joint and an equivalent block diagram model based on algebraic equations. The method deals only with the conversion process of the assembly information from MCAD to Dynamics.

This last approach is adopted in this thesis. The MCAD model and joints are converted to equivalent Block diagram models. Revolute, Prismatic and Spherical joints are supported. Given the available joints in the Modelica.Mechanics library, it is not difficult to map most assembly constraints available in current MCAD tools to the Modelica.Mechanics joint components.

With the current MCAD tools, deriving the behavioural model from the geometry requires querying the MCAD model for the list of geometric constraints between any two parts and then converting them to an equivalent model in the dynamics model. Similarly, the mass properties are extracted by querying the MCAD tool. There is no need to analyse the geometry to extract this information.

[60] provides a description of the various geometric constraints in MCAD and the equivalence between assembly constraints and joints. [61] presents the assembly constraint hierarchy and the degrees of freedom. Some tools describe the sharing of parameters that affect individual parts but the assembly and its joints are always inherited from the MCAD model.

In [62] Engelson provides an overview of a multitude of multi-body simulation tools and CAD tools. In the category multi-body simulation tools, ADAMS, Working Model 3D, 3D Studio Max, Simulink/SystemBuild and Modelica.Mechanics.Multibody are discussed. For the CAD tools, Solid Works, Working Model 3D, 3D Studio Max, Mechanical Desktop and PRO/ENGINEER are discussed. Notice that Working Model 3D and 3D Studio Max appear both under Simulation and CAD tools. This is because they support both functions simultaneously.

## 3.2 Multi-body Simulation Tools

There are a few tools that are primarily built for constructing and simulating multi-body systems. We list some of them here.

### 3.2.1 ADAMS

ADAMS [49] is a full mechanical systems simulation package. Machine parts are initially created using a 3D graphical user interface (called ADAMS/View) and assembled. The parts are connected by joints, and motion generators (motors) are attached. The items causing forces, such as springs, dampers, friction and impact, can be applied to certain points on the machine parts. The simulation engine (package ADAMS/Solver) is hidden behind the user interface and can be invoked when the user requires.

Integration of ADAMS with CAD tools from Autodesk, Unigraphics, CATIA, Pro-Engineer, Solid Works, Solid Edge and many other products is available. When ADAMS is integrated with CAD tools, the user works in his or her native CAD environment where parts and assemblies are normally designed. The Mechanical Designer simulates the model and the model movement is displayed online, during the simulation. When dynamic simulation is required, the parts and assemblies are translated internally to ADAMS program code, the model is simulated and feedback in the form of animation within the same CAD environment is returned. Additionally many parameters of simulations (forces, torques, speed etc.) can be measured and displayed in form of 2D graphs. Another possibility of ADAMS is the connection of its model to Easy5 and MATLAB/Simulink to control the mechanisms.

### 3.2.2 LINKAGEDESIGNER

LinkageDesigner [63] is a Mathematica application package, to prototype and analyse linkages and mechanisms. The package is designed for use with Mathematica 5.0, 5.1 or 5.2 version. In LinkageDesigner, kinematic structures (linkages, mechanisms, $\cdots$ ) are represented by graphs, where the links are the vertices and the joints are the edges. This graph is called the kinematic graph of the linkage.

Usually the graph based kinematic modelling becomes difficult if a linkage with a loop (or multiple loops) have to be modelled. In LinkageDesigner this is not a problem, because the package automatically generates the non-redundant loop closing equations. Linkage definitions in a parametrised way is also supported. Because of the kinematic graph based modelling, 2D and 3D mechanism are treated identically.

### 3.2.3 LMS Virtual.Lab Motion

LMS Virtual.Lab Motion [64] (see figure 3.1) offers a highly efficient, completely integrated solution to build multi-body models that simulate the full-motion behaviour of complex mechanical system designs. Users can easily create a complete and accurate system model from scratch or import geometry models from any industry-standard MCAD system. LMS Virtual.Lab Motion applies forces and motion to simulate the actual operational behaviour of the new design. The resulting simulation is excellent input to optimise the designs dynamic performance. The resulting loads can also be used for structural analysis, durability, and noise and vibration studies. Working with other simulation programs is a snap thanks to LMS Imagine.Lab AMESims enhanced interoperability with LMS Virtual.Lab Motion. From within LMS Imagine.Lab AMESim, users can simulate both system and three-dimensional designs using both LMS Virtual.Lab Motion and LMS Imagine.Lab AMESim solvers.

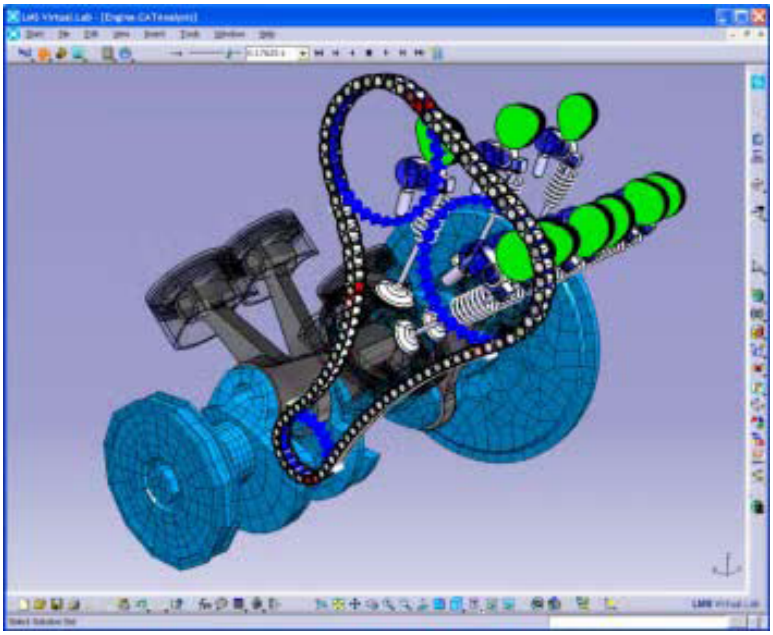


Figure 3.1: LMS Virtual Lab Motion

### 3.2.4 Modelica.Mechanics.MultiBody

The Modelica.Mechanics.MultiBody [65] library is a multi-body System Simulation library developed using the Modelica language. It contains bodies, joints, coordinate system transformations, forces, torques, and a class representing the inertial system. More detail on the Modelica.Mechanics library is provided in section 4.6.2.

Various tools support the Modelica language including Dymola, SimulationX, MathModelica, AMESim and MapleSim. However, among these only SimulationX and Dymola support the Modelica.Mechanics.MultiBody library. MathModelica does not support this multi-body library whereas MapleSim provides its own version of the multi-body library to compensate for this fact. Similarly, SimulationX provides its own multi-body implementation in addition to the Modelica version we mentioned. There are also a number of free Modelica tools including the OpenModelica[IDA lab], $\mu$-Modelica compiler [66] and JModelica but none of them support the multi-body library.

### 3.2.5 ODE (Open Dynamics Engine)

The Open Dynamics Engine (ODE) [67] is an open source, industrial quality and high performance library for simulating articulated rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools. For example, it is good for simulating ground vehicles, legged creatures, and moving objects in VR environments. It is fast, flexible and robust, and it has built-in collision detection.

### 3.2.6 SimMechanics

SimMechanics [55] is a product by MathWorks.SimMechanics provides the custom Joint that has 3 revolute, 3 prismatic and one spherical joints. The user uses a combination of these joints to create

the custom joint. Not all combinations are allowed and there are rules to follow. Before running the simulation, the application verifies that rules pertaining to the composition of SimMechanics are obeyed. Clear messages are generated that point to the problem. For example, connecting a Ground to a body is not allowed. A clear message will be generated warning the user of the problem. In Modelica, the only level at which errors are generated are at the equation level. There is no capacity to write rules that depend on the topology.

In SimMechanics, a body has no DOFs until you connect joints to it. Each joint is a combination of these joint primitives:

- P: prismatic (one translational DOF)
- R: revolute (one rotational DOF)
- S: spherical (three rotational DOFs)
- W: weld (no DOFs)

### 3.2.7 SIMPACK

SIMPACK [68] is a multi-body simulation software. SIMPACK Kinematics and Dynamics is the base module of the SIMPACK software, made up of the pre-processor, post-processor and SIMPACKs solvers. The basic concept of SIMPACK is to first create a CAD style MBS model for a mechanical or mechatronic system. The static and dynamic behaviour of the system can then be automatically solved by SIMPACK. With the SIMPACK post-processor the systems motion and forces (external and internal) can be animated and plotted.

Extensive modelling libraries support the user to rapidly create a model. Any type of joint, marker or force element (standard or user-defined), can be easily incorporated within the model. Use of a mouse enables the user to work interactively with the 3D representation. This not only saves time in setting up and modifying models but also reduces modelling errors.

The 3D graphical representation of a model can, as well as being completely set up within SIMPACK, be created within CAD packages and then imported into SIMPACK. 3D animations of the results can also be easily created. Any multi-body model, once created within SIMPACK, can be exported as FORTRAN or C code. This allows SIMPACK models to be used in other simulation environments, making SIMPACK models ideal for hardware-in-the-loop and real-time applications.

## 3.3 CAD tools, joints and mates

There are a multitude of tools for constructing geometric models of mechanical components and assemblies. We can distinguish two broad classes of modelling tools; the first is targeted towards the generation of 3d models for animation purposes such as for games, movies etc. The second is targeted towards the generation of 3d models for manufacturing purposes; This distinction comes about mainly because of the difference in requirements and the associated functionality that needs to be available. For animation, it is the visual appearance that is of primary importance. The tools will provide the user the means to define in detail visual attributes such as colour and texture. For manufacturing it is the mechanical precision with which the parts are defined and the suitability of the generated model for use in machining the part. It will be possible to annotate the parts with dimensions, assign material type and density to the parts, calculate total volume, mass, centre of gravity and inertia. These are called Mechanical CAD tools (or MCAD).

We have selected to use MCAD tools since they will store all the mechanical information needed for analysing the mechanical dynamics. The main purpose of using a MCAD tool is to create a mechanical

assembly composed of rigid parts assembled using a set of assembly constraints or joints. The 3D user-interface helps visualise the Parts, their relative positions and the insertion of the constraints, something that is not practical with standard 2D interfaces of Block Diagram (BD) tools except in simple situations.

The general mapping from a mechanical CAD assembly to a model in a multi-body simulation tool is simple overall. The individual Parts are mapped to equivalent components in the simulation tool. The assembly constraints are converted either individually or in groups to their equivalent dynamics equations. In some circumstances multiple redundant mechanical constraints may be defined in the CAD model and only one needs to be converted and is sufficient to model the dynamics.

Mates create geometric relationships between assembly components. As you add mates, you define the allowable directions of linear or rotational motion of the components. You can move a component within its degrees of freedom, visualising the assembly's behaviour. Some examples include:

- A coincident mate forces two planar faces to become coplanar. The faces can move along one another, but cannot be pulled apart
- A concentric mate forces two cylindrical faces to become concentric. The faces can move along the common axis, but cannot be moved away from this axis

We will cover the constraints in following CAD tools as a representative set: Solid Works 2010, CATIA v6, Solid Edge v100 and Autodesk Inventor 2010.

### 3.3.1 Generalised Geometric features

To define the constraints, we must first define the meaning of the geometric features involved in constraints. The following terminology is used with assembly constraints [69]. These definitions apply more generally than for the particular CAD tool (Autodesk) the book defines them for.

**Line/Linear Element** this can be the centre-line of an arc, a circular edge, a cylindrical face, a selected edge, a work axis or a sketched line.

**Normal** this is a vector that is perpendicular to a planar face.

**Plane** this can be defined by the selection of a plane or face to include the following: two non-co-linear but coplanar lines or axes, three points, or one line or axis and a point that does not lie on the line or axis. When you use edges and points to select a plane, this creates a work plane, and it is referred to as a construction plane.

**Point/Keypoint** this can be an endpoint or midpoint of a line, the centre or end of an arc or circular edge, or a vertex created by the intersection of an axis and a plane or face.

**Offset** this is the distance between two selected lines, planes, or points or any combination of the three.

### 3.3.2 Constraints in Solid Works 2010

We have three classes of mates in Solid Works and these are the simple mates, the advanced mates and the mechanism mates. We detail these mates in the tables 3.1, 3.2 and 3.3 respectively.

#### 3.3.2.1 Simple Mates

The Simple Mates in Solid Works are the most basic constraints. A constraint is most commonly applied on two geometric elements and table 3.1 provides the name, a short descriptive text and finally

a concise listing of all possible combinations of geometries expressed using set formalism.$\{a, b, c\} \times \{x, y, z\}$ means that any of LHS elements can be combined with any of the RHS elements. In case where LHS and RHS lists are the same, we use the shorthand $\{a, b, c\}^2$ and if we only have one element on either side, we use $a - b$.

Table 3.1: Solid Works Simple Mates

| **Simple Mate Types** | **Description** | **Applicable to** |
|---|---|---|
| Angle Mate | Keeps a fixed angle | {Cone, Cylinder, Extrusion-face, Line} × {Cone, Cylinder, Line}, {Cylinder, Extrusion-face, Line} × {Extrusion-face}, Plane−Plane |
| Coincident Mate | Makes two geometric features coincident. | {Point } × {Circular/Arc Edge, Cone, Curve, Cylinder, Extrusion, Line, Plane, Point, Sphere, Surface } , {Plane} × { Circular/Arc Edge, Line, Plane, Point }, {Origin} × { Coordinate System, Origin} , {Line} × { Cylinder, Line} , Cylinder − Circular/Arc Edge , {Coordinate System}$^2$ , {Circular/Arc Edge, Cone}$^2$ |
| Concentric Mate | Keeps the axis elements concentric | {Circular/Arc Edge, Cone, Cylinder, Line} × {Circular/Arc Edge, Cone, Cylinder}, {Cone, Cylinder} × {Point}, {Cylinder, Line, Point, Sphere} × {Sphere} |
| Distance Mate | Keeps the minimum distance fixed | Cone−Cone, Point−Curve, {Point, Plane, Line[i], Cylinder}$^2$, {Line, Plane, Point, Sphere} × {Sphere} |
| Lock Mate | Keeps the relative position and orientation fixed | Product − Product |
| Perpendicular Mates | Keeps the elements perpendicular | {Cone, Cylinder, Extrusion, Line}$^2$, {Line, Plane}$^2$ |
| Parallel Mates | Keeps the elements parallel | {Cone, Cylinder, Extrusion, Line}$^2$, {Line, Plane}$^2$ |
| Tangent Mate | Keeps the elements tangent | {Cylinder } × { Cam, Cylinder, Extrusion, Line, Plane, Sphere, Surface}, {Plane} × {Cam, Cone, Cylinder, Extrusion, Sphere, Surface}, {Sphere} × {Cone, Line, Sphere}, {Cone} × {Extrusion, Sphere} |

[i] Line can also refer to an axis in this instance

#### 3.3.2.2 Advanced Mates

The advanced mates are listed in table 3.2. They could be broadly described as logical constraints. The exception would be the Linear/Linear coupler mate which can be considered a mechanism.

#### 3.3.2.3 Mechanism Mates

The mechanism mates are listed in table 3.3 and correspond to (as the name implies) what are considered mechanism in engineering jargon.

Table 3.2: Solid Works Advanced Mates

| Mate Type | Description |
|---|---|
| Limit Mates | Limit mates allow components to move within a range of values for distance and angle mates. A starting distance or angle as well as a maximum and minimum value can be specified |
| Linear/Linear Coupler Mate | Establishes a relationship between the translation of one component and the translation of another component |
| Path Mate | Constrains a selected point to a path or curve. The path is defined by selecting one or more entities in the assembly. The pitch, yaw, and roll of the component as it travels along the path can be specified |
| Symmetry Mate | Forces two similar entities to be symmetric about either a plane, a planar face of a component or a plane of the assembly |
| Width Mates | A width mate centres a tab within the width of a groove |

### 3.3.3 Constraints in CATIA v6

CATIA provides geometric constraints as in other CAD tools. Each constraint when used must refer to either one, two or three elements/geometries based on the constraint type. By element we refer to either a mechanical Part (Product) or a axis system. By geometry we refer to the points, lines and other geometric features that can be identified in the 3D shape of a Product.

There are seven constraint types and they are listed together with the elements/geometries they apply on in figure 3.2. Figure 3.3 shows which combinations of elements and geometries can be used for each constraint type. One constraint alone uses a single element. Symmetry uses a third element which defines the mirroring plane. All other constraints besides symmetry use two elements/geometries.

These constraints are all geometric in nature. CATIA provides a layer on top of these constraints called **Engineering connection types** as seen in figure 3.4 which can be interpreted as defining specific mechanical behaviours. The engineering connection is made with a set of constraints between products (usually two) where a typed-relation is defined which takes into account kinematics relation. In addition some connection types like **Prismatic** and **Revolute** have alternative definitions while producing identical mechanical behaviour. User-defined engineering connections can also be defined by specifying the set of constraints composing it and the types of geometries they apply on.

### 3.3.4 Constraints in Solid Edge v100

Solid Edge assembly constraints are briefly described in table 3.4 and a more complete description is provided in section 3.6.

### 3.3.5 Constraints in Autodesk Inventor 2010

Autodesk Inventor uses:

- four types of **assembly constraints** (mate, angle, tangent and insert),
- two types of **motion constraints** (rotation and rotation-translation),
- a **transitional constraint**, and
- a **constraint set**.

Table 3.3: Solid Works Mechanisms Mates

| Mate Type | Description |
|---|---|
| Cam Follower Mates | A type of tangent or coincident mate. It allows to mate a cylinder, plane, or point to a series of tangent extruded faces, such as found on a cam. The cam profile can be made from lines, arcs and splines, as long as they are tangent and form a closed loop |
| Gear Mates | Force two components to rotate relative to one another about selected axes. Valid selections for the axis of rotation for gear mates include cylindrical and conical faces, axes, and linear edges |
| Hinge Mates | Limits the movement between two components to one rotational degree of freedom. It has the same effect as adding a concentric mate plus a coincident mate. The angular movement between the two components can also be limited |
| Rack and Pinion Mates | The linear translation of one component (the rack) causes circular rotation in another component (the pinion), and vice versa. Any two components can be mated to have this type of movement relative to each other. The components do not need to have gear teeth |
| Screw Mate | Constrains two components to be concentric, and also adds a pitch relationship between the rotation of one component and the translation of the other. Translation of one component along the axis causes rotation of the other component according to the pitch relationship. Likewise, rotation of one component causes translation of the other component |
| Universal Joint Mate | The rotation of one component (the output shaft) about its axis is driven by the rotation of another component (the input shaft) about its axis |

These constraints are detailed in the following sections.

#### 3.3.5.1 Assembly Constraints

In this section we cover all the assembly constraints including the mates, angle, tangent and insert. The mate constraint itself comes in four types: mate plane, mate line, mate point and mate flush. These are listed in table 3.3.5.1.

#### 3.3.5.2 Motion Constraints

In the section we cover all the motion constraints described in table 3.3.5.2. There are two types of motion constraints: rotation and rotation-translation. Motion constraints allow you to simulate the motion relationship of gears, pulleys, rack and pinions, and other devices. Both types of motion constraints are secondary constraints, which means that they define motion but do not maintain positional relationships between components. Before Motion constraints can be applied, the components must be fully constrained.

#### 3.3.5.3 Transitional Constraint

A Transitional Constraint will maintain contact between two selected faces. You can use a transitional constraints between a cylindrical face and a set of tangent faces on another part. The transitional constraint specifies the intended relationship between, typically, a cylindrical part face and a contiguous

Table 3.4: Solid Edge assembly constraints

| **Mate Type** | **Description** |
|---|---|
| Match Coordinate Systems | The Match Coordinate Systems command positions a part in an assembly by matching the x, y, and z axes of a coordinate system on the part you are placing with the x, y, and z axes of a coordinate system on a part already in the assembly |
| Planar Align | Applies a Planar align Assembly relation between two Parts in an assembly. A Planar align relation forces the planar element on one Part to remains parallel to and facing the same direction as the planar element of another Part |
| Mate | Applies a mate relation between two parts in an assembly. A mate relation is a replica of the Planar Align relation except that the face normals are in opposite directions |
| Angle | Applies a fixed angle relation between two planar elements or two edges of two distinct parts in an assembly |
| Axial Align | An Axial align aligns two cylindrical axes, a cylindrical axis and a linear element or two linear elements |
| Insert | The Insert command is equivalent to applying in sequence a Mate relation with a fixed offset and an Axial align relation with a fixed or floating rotation angle |
| Connect | The Connect relation comes in three variants. It is used to position a keypoint on one part with a keypoint, line, or face on another part. The Connect relation fixes the distance between the two endpoints |
| Tangent | A Tangent relation comes in two variants. It is used to position a cylindrical face with either another cylindrical face or a planar element. It ensures that the cylindrical face of one part in an assembly remains tangent to a cylindrical face or planar element of another part |
| Parallel | Makes a Part's axis or Edge parallel to another Part's Axis or Edge |
| Gear | A gear relation comes in three variants. It defines the ratio of relative movement between two rotating Parts, one rotating Part and another translating Part or two translating Parts. This is useful when working with assemblies that contain gears, pulleys, parts that travel in grooves or slots, and hydraulic or pneumatic actuators |
| CAM | A cam changes the input motion, which is usually rotary motion (a rotating motion), to a reciprocating motion of the follower. They are found in many machines and toys. A cam has two parts, the **follower** and the **profile**. The cam profile is composed of a sequence of continuous curves forming a close loop all in the same plane. The follower requires a single point that will remain tangent to the cam profile |

Table 3.5: Autodesk Assembly Constraints

| **Name** | **Description** |
|---|---|
| Mate Plane | faces on the selected planes will be planar and opposing each other |
| Mate Line | lines are made to be co-linear. you can also use mate line constraint to centre axis of a cylinder with a matching hole feature or axis |
| Mate point | constraint assembles two points, such as centres of arcs and circular edges, endpoints, and midpoints to be coincident |
| Math Flush | two planar faces face the same direction or have their surface normals point in the same direction. Applies only to planar faces |
| Angle Constraint | specifies the degrees between selected planes or faces or axes |
| Tangent | the tangent constraint defines a tangent relation between planes, cylinders, spheres, cones and ruled splines. At least one of the faces selected needs to be curved |
| Insert | applies to components with circular edges. the centerlines of the selected circles of arcs will be aligned and a mate constraint will be applied to the planes defined by the circular edges |

Table 3.6: Autodesk Motion Constraints

| **Name** | **Description** |
|---|---|
| Motion Rotation constraint | defines how one component will rotate in relation to another by specifying the ratio for the rotation between the two components |
| Motion rotation-translation | defines the rotation relative to translation between components |

| Constraint Name | Icon |
|---|---|
| Fix | |
| Fix Together | |
| Coincidence | |
| Contact | |
| Distance | |
| Angle | |
| Symmetry | |

| Element/Geometry | Icon |
|---|---|
| Product | |
| Axis system | |
| Point | |
| Line | |
| Curve | |
| Circle | |
| Plane | |
| Surface | |
| Sphere | |
| Cylinder | |
| Cone | |

Figure 3.2: CATIA constraint and geometry icons

| Constraint | First Element | Second Element | Third Element |
|---|---|---|---|
| | or | NA | NA |
| | or | or | NA |
| | | or or or or | NA |
| | or | or or | NA |
| | | | NA |
| | | or or | NA |
| | | or or | NA |
| | | or | NA |
| | | or | NA |
| | or or | or or | NA |
| | or | | NA |
| | | | |
| | | | |
| | | | |
| | | | |

NA: Not applicable.

| Constraint Name | Icon | Element/Geometry | Icon |
|---|---|---|---|

Figure 3.3: CATIA v66 Constraints and Geometry association

set of faces on another part, such as a cam follower in a cam sot.The transitional constraint maintains contact between the faces as you slide the components along open degrees of freedom.

#### 3.3.5.4 Constraint Set

If User-Coordinate Systems (UCS) were defined in individual part or assembly files, these UCS can be constrained together. This would form a constraint set.

### 3.3.6 Constraints Comparison

Table 3.7 is a compendium of the joint types we have encountered and the particular names they are given in the tools we covered. This comparison is partial as covering all the possible allowed combinations of joints and geometries would be too long and would detract from the purpose of the table which is to show the degree of similarity between the tools.

Table 3.7: CAD Tools and Constraints Comparison

| **Connect Elem.#1** | **Connect Elem.#2** | **Solid Works** | **CATIA** | **Solid Edge** | **Autodesk** |
|---|---|---|---|---|---|
| Product / Axis System | | -NA- | Rigid / Fix | Ground | -NA- |
| Line | Line | Angle | Angle | -NA- | Angle |
| Line | Plane | -NA- | Angle | Angle | Angle |
| Plane | Plane | Angle | Angle | Angle | Angle |

*continued on next page*

*continued from previous page*

| **Connect Elem.#1** | **Connect Elem.#2** | **Solid Works** | **CATIA** | **Solid Edge** | **Autodesk** |
|---|---|---|---|---|---|
| Plane | Plane | Coincident mate | Planar: Coincidence / Distance / Contact | Mate / Planar Align | Mate Plane / Mate Flush |
| Cone | Circle | Coincident mate | Contact | -NA- | -NA- |
| Cone | Cone | Coincident mate | Contact | -NA- | -NA- |
| Line | Line | Concentric Mate (colinear axes) | Cylindrical: Coincidence | Axial Align | Mate Line |
| Point / Line / Plane / Cylinder | Point / Line / Plane / Cylinder | Distance | Distance | -NA- | -NA- |
| Point | Point | Distance | Distance | Connect | Mate Point (Distance!=0) |
| Point | Line | Distance | Distance | Connect | -NA- |
| Point | Plane | Distance | Distance | Connect | -NA- |
| Line | Line | Distance | Distance | -NA- | -NA- |
| Line | Plane | Distance | Distance | -NA- | -NA- |
| Point | Point | Distance (Distance=0) | Spherical: Coincidence | Connect (Distance=0) | Mate Point (Distance=0) |
| Point | Line | Distance (Distance=0) | Coincidence | Connect (Distance=0) | -NA- |
| Point | Plane | Distance (Distance=0) | Coincidence | Connect (Distance=0) | -NA- |
| Point | Curve | Distance (Distance=0) | Coincidence | -NA- | -NA- |
| Point | Surface | Distance (Distance=0) | Coincidence | -NA- | -NA- |
| Line | Plane | Distance (Distance=0) | Coincidence | -NA- | -NA- |
| Axis System | Axis System | Lock | Fix together / Coincidence | Match Coordinates | UCS (Universal Coordinate System) |
| Line | Line, Plane | Parallel | ? | Parallel | ? |
| Line, Plane | Line, Plane | Perpendicular | -NA- | -NA- | -NA- |
| Plane | Cylinder | Tangent | Contact | Tangent | Tangent |
| Cylinder | Cylinder | -NA- | Contact | Tangent | -NA- |

*continued on next page*

*continued from previous page*

| **Connect Elem.#1** | **Connect Elem.#2** | **Solid Works** | **CATIA** | **Solid Edge** | **Autodesk** |
|---|---|---|---|---|---|
| Sphere | Sphere | Tangent | Contact | -NA- | -NA- |
| Sphere | Cylinder | Tangent | Contact | -NA- | -NA- |
| Plane | Sphere | Tangent | Contact | -NA- | -NA- |
| Sphere | Circle | -NA- | Contact | -NA- | -NA- |
| Point | Curve | Path Mate | Coincidence | -NA- | -NA- |
| Point | Point | Symmetry | Symmetry | Symmetry | -NA- |
| Line | Line | Symmetry | Symmetry | Symmetry | -NA- |
| Plane | Plane | Symmetry | Symmetry | Symmetry | -NA- |
| Axis System | Axis System | Symmetry | Symmetry | Symmetry | -NA- |
| Sphere | Sphere | Symmetry | -NA- | -NA- | -NA- |
| Cylinder | Cylinder | Symmetry | -NA- | -NA- | -NA- |
| -NA- | -NA- | Width Mate | ? | -NA- | -NA- |
| -NA- | -NA- | Limit Mate | -NA- | -NA- | -NA- |
| -NA- | -NA- | Linear-Linear Coupler | ? | Gear Linear-Linear | ? |
| Point | Surface | Cam follower | Coincidence | CAM | Transitional constraint |
| Cylinder | Cylinder | Gear Mate: rotation-rotation | ? | Gear rotation-rotation | Motion rotation-rotation |
| | | Hinge Mate | -NA- | -NA- | -NA- |
| Cylinder | Line | Rack and Pinion: linear-rotation | ? | Gear rotation-translation | Motion rotation-translation |
| Cylinder | Cylinder | Screw Mate: rotation-rotation + co-linear | ? | Gear rotation-rotation + Axial Align | Motion rotation-rotation + Mate Line |
| -NA- | -NA- | -NA- | Revolute: Coincidence (Line-Line) + Contact / Distance (Plane-Plane) | Insert: Axial Align + Planar Align | Insert: Mate Line + Mate Plane |
| -NA- | -NA- | -NA- | Revolute: Coincidence (Line-Line) + Distance / Coincidence (Point-Point) | Axial Align + Connect (Point-Point) | -NA- |

*continued on next page*

*continued from previous page*

| Connect Elem.#1 | Connect Elem.#2 | Solid Works | CATIA | Solid Edge | Autodesk |
|---|---|---|---|---|---|
| -NA- | -NA- | -NA- | Prismatic: Coincidence (Line-Line) + Coincidence (Plane-Plane) | axial Align + Planar Align | Mate Line + Mate Flush |
| -NA- | -NA- | -NA- | Prismatic: Coincidence (Line-Line) + Coincidence (Line-Line) | Axial Align + Axial align | Mate Line + Mate Line |
| -NA- | -NA- | -NA- | User defined | -NA- | -NA- |

## 3.4 Mechanical Interference and Collision detection

An important aspect in mechanical modelling and simulation is mechanical interference detection and collision dynamics. Interference depends on geometry, whereas collision dynamics depends both on geometry, material and surface properties, as well as the structural properties of the colliding parts and the dynamics of the mechanical assembly.

### 3.4.1 Interference detection

Interference detection is concerned with detecting whether various mechanical parts have some overlap in geometry which signifies that the configuration is not physically achievable. This is very useful when assembling systems with a large number of parts or even a small number of parts where the parts are very close to each other as it could be very time-consuming for the user to determine interference visually. Most commercial CAD tools provide interference detection by default. These include Solid Edge, Solid Works, CATIA.

The detection of interference between any two or more parts requires that a central application be aware of the geometry, positions and orientations of all parts. The process must determine for any pair of parts whether interference occurs or not. If there were N parts, there would have to be $\frac{N\times(N-1)}{2}$ tests in the worst case. Fortunately, there are techniques that can limit both the number of tests and the complexity of the tests. For example, to limit the number of tests, each part can be associated with an indexed volume in space (or zone). A part does not have to be tested against any part that is in a different zone. This is similar to saying that a Part in North America does not need to be tested for interference with a Part in Europe. Further, if we have to test two parts for interference, we can do so in several steps. The first step is usually to test whether the bounding boxes of each part have any interference. A bounding box is a rectangular shape containing the part and due to its geometric simplicity, interference calculation is not complicated. If the test is positive (interference detected) then is required to carry out the interference tests using the details geometries of each part.

### 3.4.2 Collision detection and contact dynamics

Collision detection makes use of interference detection as a first step to determine when various mechanical parts in motion have collided with each other. When interference is detected, similar to

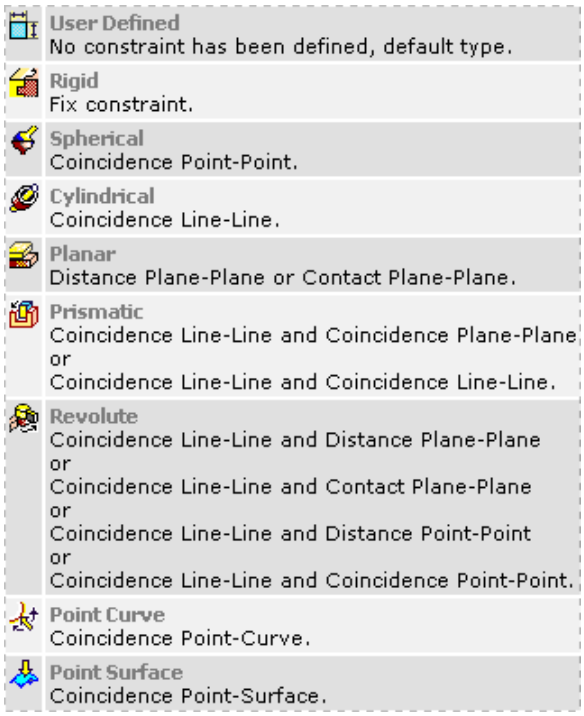



Figure 3.4: CATIA Engineering Connection Types

zero-crossing, the parts are reset to the point in time before interference occurred and a collision detection flag is raised. When this happens, if contact dynamics is implemented, then forces and torques to apply on the colliding parts must be calculated to simulate proper behaviour such as the bouncing of a ball when hitting the ground or the friction between two sliding surfaces that have come into contact. In general, contact dynamics can be very complicated depending on the material and surface properties of the colliding parts. ADAMS is one of the tools that implements contact dynamics to some extent.

In multi-body dynamics tools such as SimMechanics and Modelica.Mechanics, contact dynamics is not implemented. Joints allow connecting two parts to each other but there are no mechanism to connect all parts to each other in order to model collision detection and contact dynamics. If contact dynamics were to be implemented in such tools, it would not be practical to require a connection between each pair of interacting parts as this would lead to a great amount of clutter in the model. A better approach is to implement a central **collision detection and contact dynamics** function, require that each part provide it with geometry, position, velocity and orientation without requiring a visible connection, and have this new function calculate in return the forces and torques to apply on colliding parts.

## 3.5 Connecting Geometry With Simulation

We have covered multi-body simulation tools in section 3.2 and 3D mechanical modelling tools and constraints in section 3.3. Collision and contact dynamics was also discussed in section 3.4 but we will not develop this subject further in this thesis. There are many implementations relating or mapping the models in these CAD and multi-body simulation tools. We will describe one implementation in some detail and provide a brief review of the others. Note that all such implementation convert CAD models to multi-body simulation but never in the reverse direction.

Current generation of simulation models from CAD models can be categorized under the following headings:

1. No External interface: some tools require that the rules controlling the mechanisms be specified within the CAD tool. The user specifies forces and torques that apply on the joints and mechanisms directly in the CAD tool. The extensibility of such approaches is usually limited.

2. Black-box interface: other tools generate a black box simulation model that can be integrated in an external simulation tool where then the controller can be defined using all the capabilities of the simulation environment.

3. Topology interface: some implementations generate a simulation model of the mechanical assembly that offers a close association between the components and mechanical topology in the two models. CAD parts and joints have matching and distinct components in the simulation model.

The Working Model 3D (section 3.5.2.1) self-contained 3D modelling and dynamics simulation tool falls under the $1^{st}$ category. The MSC ADAMS-Easy5 (section 3.5.2.2) code export falls under the $2^{nd}$ category. The Solid Works to Modelica (section 3.5.1) or SimMechanics (section 3.5.2.3), CATIA to Modelica (section 3.5.2.4) translators fall under the $3^{rd}$ category.

### 3.5.1 Solid Works To Modelica

In [52], the authors have developed a tool to convert a Solid Works CAD model to Modelica code using the Modelica.Mechanics.MultiBody library components. Figure 3.5 shows the architecture of the CAD to Modelica converter. the converter extracts the 3D geometry from the Solid Works Part models and generates stereo lithography (STL) format files. It also extracts the mass properties of each Part. The Solid Works assembly itself is analysed to determine all the mating constraints. The mass properties are used to generate Modelica.Mechanics.MultiBody Part models and the mating constraints are converted to Modelica Joints and a corresponding Modelica Assembly is created. The figure also shows that **non-mechanical model components** are combined with the mechanical aspects. These would be all the forces and torques acting on the mechanism in addition to any other models we wish to include. The results of the Modelica simulation are then used to drive a 3D animation.

### 3.5.2 Other references

#### 3.5.2.1 Working Model 3D

Working Model 3D is an advanced tool providing dynamic analysis within an integrated modelling and simulation environment. In the Working Model 3D tool users create set of bodies (mechanical parts) and describe how these are pairwise connected by joints. Only primitive shapes can be constructed within Working Model 3D. However, it can import arbitrarily complex shapes from various CAD tools. The tool performs dynamic simulation of systems of rigid bodies. When a system is constructed, the revolute, prismatic, spherical and many other kinds of joints can be specified. The tool is able to detect collisions and produce response impulses. The results of analysis can be displayed as 3D scenes as well as 2D graphs of any variables computed during simulation. The major drawback of Working Model 3D is the absence of communication with the outer world. The information in Working Model 3D is available mainly inside the tool. Sometimes useful information is displayed by the tool but cannot be automatically extracted for use by external programs. The data export capabilities of Working Model 3D are limited: simulation results can be exported, but joint information cannot. Mechanical joints that can be specified in Working Model 3D directly correspond to joints in the Modelica MBS library.

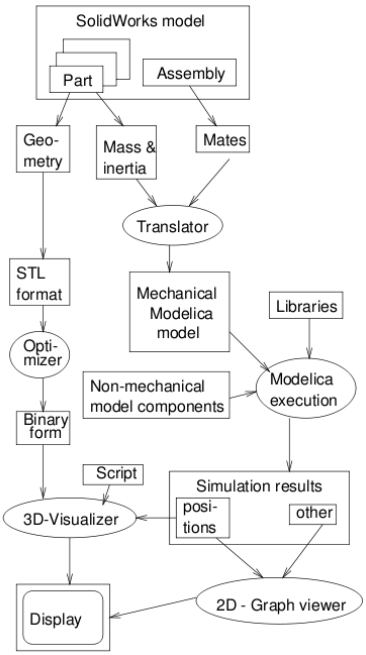



Figure 3.5: The path from Solid Works model to dynamic system visualisation

The mass, the centre of mass and the inertia tensor are automatically computed from user-specified density. However, all these can be overridden by the user.

#### 3.5.2.2 ADAMS to EASY5

ADAMS [49] and EASY5 [34] are two tools that provide Mechanical assembly simulation and analysis functionality. Whereas ADAMS is a custom application made specifically for this purpose it suffers from having limited capability for defining the forces and torques that may drive the mechanisms. Easy5 on the other hand is a general block-diagram modelling and simulation tool with custom libraries provided for a multitude of domains (mainly targeted for simulation airplane subsystems).

The ADAMS to Easy5 translator generates a single Easy5 block (driven with C-Code) that defines the complete mechanical assembly with interfaces to measure the dynamic parameters and input for forces and torques to apply on the mechanical assembly. The user cannot change the mechanical model in Easy5 or has any visibility of the topology of the mechanical assembly.

#### 3.5.2.3 Solid Works to SimMechanics

In Solid Works, an unconstrained part has six degrees of freedom (DOFs). You reduce these DOFs by inserting mates (constraints) between bodies. Only a subset of the Solid Works Mates are supported. These are: Angle, Parallel, Coincident, Perpendicular, Concentric, Tangent and Distance. Mate Entities are the geometric features used in the establishing of a constraint. The Solid Works mate entities supported for this translator are:

- Point : Vertex/sketch point/reference point
- Line : Linear edge/sketch segment/reference axis
- Plane : Reference plane or plane face

- Circle/Arc : Circular edge/arc segment
- Cone : Conical face
- Cylinder : Cylindrical face

In SimMechanics, a body has no DOFs until you connect joints to it. Unlike Solid Works, there are no elements corresponding to the mate entities above. There is only a mechanical connection type used by all joints. Each joint is a combination of joint primitives:

- P: prismatic (one translational DOF)
- R: revolute (one rotational DOF)
- S: spherical (three rotational DOFs)
- W: weld (no DOFs)

The translator maps the Solid Works mates (constraints) between parts to SimMechanics joint primitives between bodies. In general, the mapping of mates to joints is not one-to-one. When a SimMechanics model is generated from a MCAD assembly, the primitives are combined into the appropriate Joints.

#### 3.5.2.4 CATIA to Modelica Multi-body

In [70] an interface from CATIA to Modelica is demonstrated. The abstract of the paper provides a concise resumé:

> Traditionally, multi-body systems have been defined in Modelica by connecting bodies and joints in a model diagram. Additionally the user must enter values for parameters defining masses, inertias and three dimensional vectors of positions and orientations. More convenient definition of ‘ systems can be made using a 3D editor available, for example, in CATIA from Dassault Systèmes with immediate 3D viewing. A tool has been developed that translates a CATIA model to Modelica by traversing the internal CATIA structure to get information about parts and joints and how they are related. This information is then used to generate a corresponding Modelica model. The traversal provides information about the reference coordinate system, the centre of mass in the local coordinate system, the mass, the inertia, the shape and colour of the body exported in VRML format for animation purposes and the icon exported as a PNG file to be used in the Modelica diagrams. The Modelica diagram layout is automatically generated and is based on the spanning tree structure of the mechanism. Models obtained in this way often contain redundant constraints. A new method has been developed for Dymola to facilitate simulation of such models, i.e. the model reduction is performed automatically. An important property of the translated model is the possibility to use Modelica extends (inheritance) for adding controllers and other features of the model for dynamic simulation. For instance, the engine model can be extended by introducing models of the gas forces of the combustion acting on the cylindrical joints of the pistons. In that way, the translated model is separated and can be changed independently of the added models **(M. Otter, H. Elmqvist, and S. E. Mattsson)**

## 3.6 Selected MCAD Tool: Solid Edge

We focus our attention on Solid Edge as it is the tool we had available and that was used for this thesis. We will examine how 3-dimensional (3D) parts are constructed and how assembly constraints

apply to parts in order to form an assembly.

### 3.6.1 Part Model

A MCAD **Part** represents a rigid mechanical object including its physical properties, dimensioning parameters and geometry. A Part has an associated **base reference frame** relative to which all the geometric features and mass properties are defined.

#### 3.6.1.1 Part Physical Properties

MCAD tools are required for performing an FEA in the thermal, mechanical, electrical and other physical domains. As such, tool vendors have increased the information stored in Parts. Users can now associate physical properties in order to simplify the FEA modelling process which inherits much of the information it requires from the MCAD model directly. Solid Edge, Solid Works, Autodesk Inventor and many others provide this capability.

**General Properties** The Part is assumed to be made of an isotropic material and as such the physical properties are not assigned to any specific geometric feature or a geometric location. Examples of physical properties are material density and thermal conductivity. The first two lines in figure 3.8 show two user-defined properties: User_Defined_Force and HeatOutput.

**Mass Properties** Some physical properties have a special status in MCAD tools. Mass properties are in this category and include:

- Total mass of Part
- Centre of mass coordinates
- Mass moments of inertia

Unlike general properties, the mass properties can be either set manually or calculated automatically. If set manually, these values remain unchanged irrespective of how the geometry of the part or its density have changed. Otherwise they are calculated from the geometry information and the assigned density value.

Figure 3.6 shows the user interface in Solid Edge for setting the mass properties manually. Figure 3.7 shows the same user interface with different settings where the mass properties are calculated automatically from the geometry and the material density. The values are different from the previous figure.

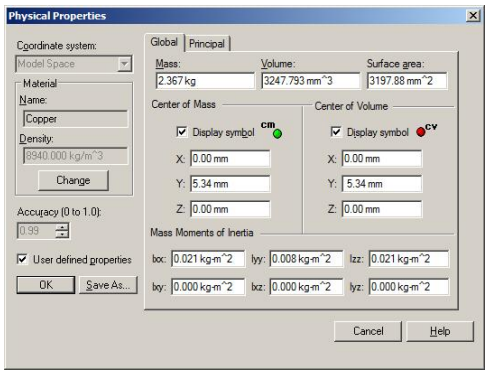



Figure 3.6: User-assigned mass properties

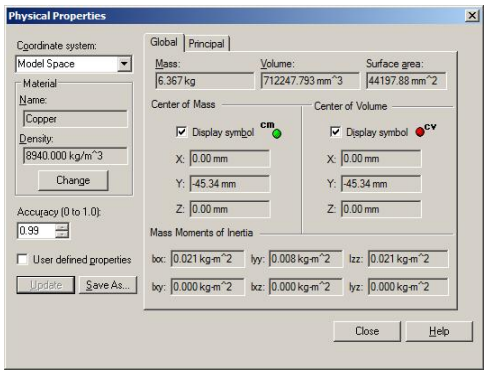
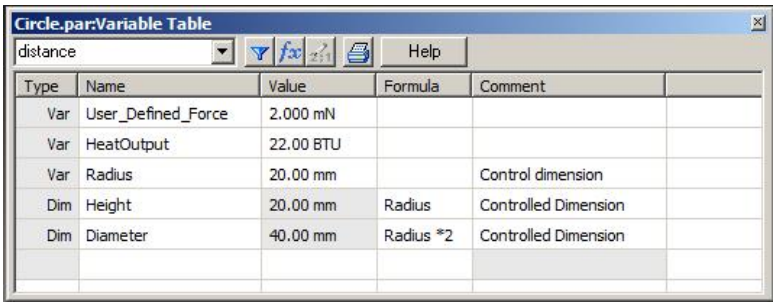



Figure 3.7: Geometry-derived mass properties

### 3.6.1.2 Part Dimensioning Parameters

Parameters can be created to control the dimensions of a Part. Figure 3.8 shows the user-interface and the parameter **Radius** which is used in the formulae in the following two lines to control the values of the dimensions **Height** and **Diameter**. The result is seen in figure 3.9.




| Type | Name | Value | Formula | Comment |
|---|---|---|---|---|
| Var | User_Defined_Force | 2.000 mN | | |
| Var | HeatOutput | 22.00 BTU | | |
| Var | Radius | 20.00 mm | | Control dimension |
| Dim | Height | 20.00 mm | Radius | Controlled Dimension |
| Dim | Diameter | 40.00 mm | Radius *2 | Controlled Dimension |

Figure 3.8: Part physical properties

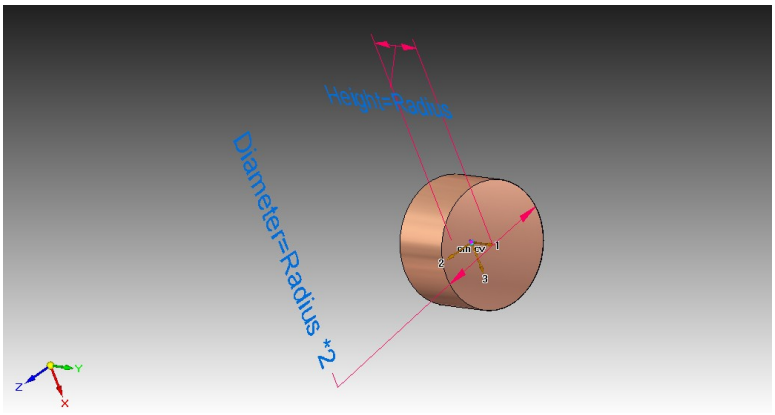



Figure 3.9: Parameter-driven geometry

### 3.6.1.3 Geometric Features

In the process of constructing a Part, a series of features emerge that we can use for placing constraints between Parts. The simplest geometric features are usually supported by most tools and include:

- Points including end points of lines or curves, midpoints of lines and arc or circle centres
- Straight lines
- Planar surfaces
- Circle or Arc axes
- Cylinder axes
- Cylinder surfaces

We present the geometric features relevant for geometric constraints as well as their mathematical abstraction as used by geometric constraints.

**Point Elements** Although geometrically any line or face has a infinite number of points, the points of relevance for geometric constraints in MCAD are usually those that have a special meaning. Points include all vertices on geometries of 1D, 2D or 3D. This includes endpoints of line elements (straight line, arc, spline, $\cdots$). These line elements may be the edges of higher dimensional geometries such as faces and volumes. Points also include geometrically meaningful points that are not visible. The centre of a circle or an arc, the midpoint of a straight-line and others fall in this category .

Mathematically, a point is defined by its 3 coordinates:

$$\mathbf{p} = (x, y, z) \text{ with } x, y, z \in \mathbb{R} \tag{3.6.1}$$

**Linear Elements** Any element of geometry that is the equivalent of a straight line falls in this category. This includes straight lines, straight edges, cylinder axes and the axes of a circle or an arc. Mathematically, a line is defined by all the points $x$ such that:

$$\mathbf{x} = \mathbf{p} + s\ \mathbf{v}\ , \textit{with}\ \mathbf{p} \in \mathbb{R}^3,\ s \in \mathbb{R}\ \textit{and}\ \mathbf{v} \in \mathbb{R}^3 \tag{3.6.2}$$

where $\mathbf{p}$ is a known point on the line, $s$ is a free variable and $\mathbf{v}$ is the line direction vector. Geometric constraints assume that all linear elements are infinite lines.

**Planar Elements** A Planar element is any planar surface with an associated positive normal direction. For a planar element that is not the surface of a 3D geometry, the positive normal depends on the creation process. Otherwise, the positive normal corresponds to the directions that points out of the volume of the geometry. The direction of the normal is relevant when specifying geometric constraints between Parts. Mathematically, a point $\mathbf{x}$ is on the planar surface if:

$$\mathbf{N} \cdot (\mathbf{p} - \mathbf{x}) = 0\ , \textit{with}\ \mathbf{N}, \mathbf{p}, \mathbf{x} \in \mathbb{R}^3 \tag{3.6.3}$$

where $\mathbf{p}$ is a known point on the plane and $\mathbf{N}$ a vector normal to the plane. Geometric constraints consider that planar elements are infinite planes.

**Cylinder Face** Cylinder faces and Partial cylindrical surfaces where the base does not form a complete circle are equivalent for geometric constraints. Mathematically, a point $x$ is on the surface of a (partial or full) cylinder if:

$$\mid (\mathbf{x} - \mathbf{p}) - (\mathbf{x} - \mathbf{p}) \cdot \hat{\mathbf{v}}\ \hat{\mathbf{v}} \mid = R\ , \textit{with}\ \mathbf{x}, \mathbf{p}, \hat{\mathbf{v}} \in \mathbb{R}^3 \tag{3.6.4}$$

where $\mathbf{p}$ is a known point on the axis of the cylinder and $\hat{\mathbf{v}}$ a normalised vector along the cylinder axis. Geometric constraints consider that the cylinders are infinite and have a full circle as a base.

### 3.6.2 Assemblies

A MCAD **Assembly** is a composition or a collection of Parts, sub-Assemblies and Assembly relations.

**SubAssembly** An Assembly inserted in another is called a **SubAssembly**.

**Assembly relation** An **Assembly relation** is a geometric constraint between two geometric features of two distinct Parts used to position and constrain them relative to each other. The application of multiple Assembly relations between two parts reduces or completely eliminates the relative degrees of freedom. MCAD tools calculate on the fly remaining degrees of freedom after each new Assembly relation is placed and verify at the same time whether a new Assembly relation can be inserted without conflicting with the previous ones. After the Assembly relations have been inserted, any remaining Degrees of Freedom (DOF) can be used to represent mechanical joints.

### 3.6.3 Assembly Relations

In the previous section, we discussed the geometric features used in geometric constraints. In this section we will describe the geometric constraints that can be imposed between two parts using these features as **connection points**. We will limit the discussion to a representative set from the Solid Edge tool. Other tools (Solid Works, CATIA, NX $\cdots$) provide slightly different constraints.

#### 3.6.3.1 Match Coordinate Systems

The **Match Coordinate Systems** command positions a part in an assembly by matching the x, y, and z axes of a coordinate system on the part you are placing with the x, y, and z axes of a coordinate system on a part already in the assembly. An offset value can be defined for each of the coordinate system axes. The command in effect introduces three Planar Align Assembly relations each with its own translational offset. Figure 3.10 shows an example where the top two Parts are assembled by matching their coordinate systems resulting in the bottom Assembly.

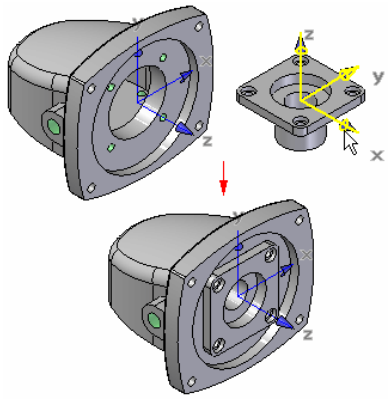



Figure 3.10: Match Coordinates Assembly example

Application is between:

1. Two Coordinate systems

#### 3.6.3.2 Planar Align

Applies a **Planar align** Assembly relation between two Parts in an assembly. A Planar align relation forces the planar element on one Part to remains parallel to and facing the same direction as the planar element of another Part. The two planar elements can be set to have a fixed offset or floating. If the offset is floating, then the planar elements are free to move away from each other along the plane normal direction. Otherwise, the distance is fixed. Figure 3.11 shows two Parts with Planar elements highlighted on the left. When the Planar Align relation is applied with a zero offset it results in the figure to the right. Figure 3.12 shows the same system but this time with a non-zero offset.

When you define a floating offset, you can apply another relation that controls the offset distance. You can also use a Planar align relation to position a part with respect to an element that is in a part, sub-assembly, or top-level assembly sketch.

Application is between:

1. Two Planar Faces

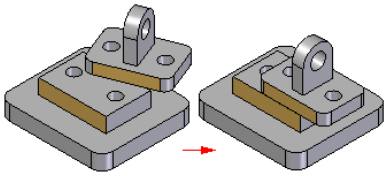


Figure 3.11: Planar Align assembly of fixed zero offset

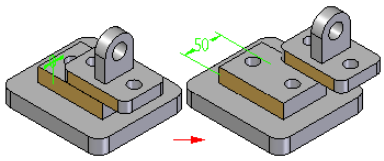



Figure 3.12: Planar Align assembly of fixed non-zero offset

Versions of Planar align:

1. Fixed offset
2. Floating offset

#### 3.6.3.3 Mate

Applies a mate relation between two parts in an assembly. A mate relation is a replica of the Planar Align relation except that the face normals are in opposite directions. Figures 3.13 and 3.14 show the creation of two assemblies where one has a zero offset and the other a non-zero offset respectively.

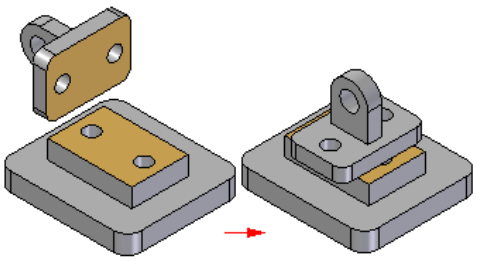


Figure 3.13: Mate assembly of fixed zero-offset

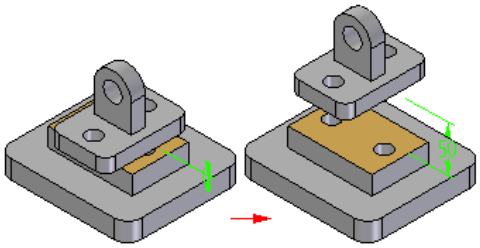



Figure 3.14: Mate assembly of fixed non-zero offset

When you define a floating offset, you can apply another relation that controls the offset distance. You can also use a Mate relation to position a Part with respect to an element in a Part, sub-assembly, or top-level assembly sketch.
Application is between:

1. Two planar faces

Versions of Mate command:

1. Fixed offset
2. Floating offset

#### 3.6.3.4 Angle

Applies a fixed angle relation between two planar elements or two edges of two distinct parts in an assembly. This relation is typically used to allow a Part to pivot about an axial align in order to connect between two edges. The angular value of the relation can be edited to rotate the Part in the

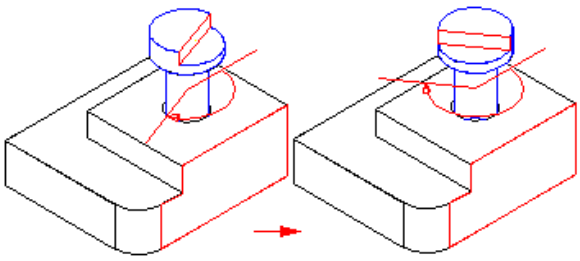


Figure 3.15: Angle assembly

assembly. Figure 3.15 shows the Angle relation applied between two Planar elements of two Parts which already have an Axial align.

Application is between:

1. two faces
2. two edges

#### 3.6.3.5 Axial Align

An Axial align relation applies between two cylindrical axes, a cylindrical axis and a linear element or two linear elements. In figure 3.16, an axial align relation is applied between a cylindrical axis on the Part being positioned (A) and a cylindrical axis on a Part already in the assembly (B) resulting in the assembly shown on the right.

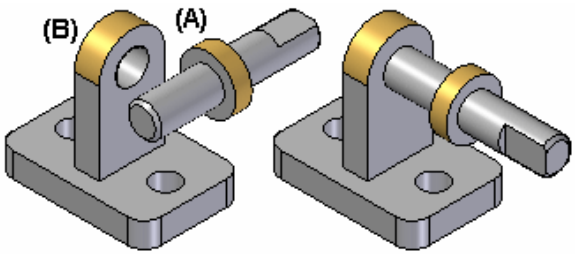



Figure 3.16: Axial Align assembly

The rotational axis can be either locked or unlocked. When the rotation is locked, the rotational orientation is fixed at a random location. This option is useful when the rotational orientation of the part is not important, such as placing a bolt into a hole. When the rotation is unlocked, another assembly relation to control the rotational orientation of the part can be applied. For example, an angle relation is seen in figure 3.17. An axial align relation can be used to position a Part with respect to an element that is in a Part, sub-assembly, or an assembly sketch.

Application is between:

1. two cylindrical axes
2. a cylindrical axis and a linear element
3. two linear elements

Versions of Axial Align:

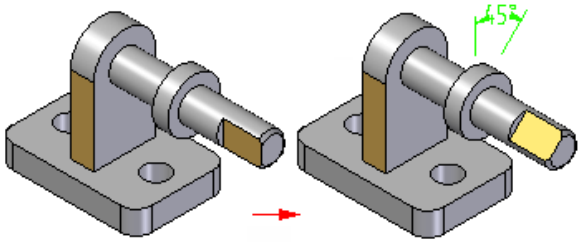



Figure 3.17: Axial Align with additional angle relation

1. rotation unlocked: the two parts are free to rotate around the axis
2. rotation locked: the two parts are rotationally locked

#### 3.6.3.6 Insert

The Insert command is equivalent to applying in sequence a Mate relation with a fixed offset and an Axial align relation with a fixed or floating rotation angle. This command is typically used to place axial-symmetric parts, such as nuts and bolts, into holes or onto cylindrical protrusions as seen in figure 3.18. The left-most Parts with two Planar elements highlighted is where the Mate relation is applied. The middle set of Part with two cylindrical surfaces highlighted is where the axial align relation is applied. The application of both produces the assembly on the far right.

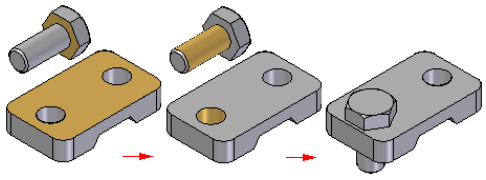


Figure 3.18: Insert assembly (zero offset)

Figure 3.19 shows the result of applying an offset to the Mate relation. Figure 3.20 shows an assembly with an insert relation where the axial align is floating and an additional angle relation is inserted to rotate the bolt. The angle relation is applied between two Planar elements which are highlighted.

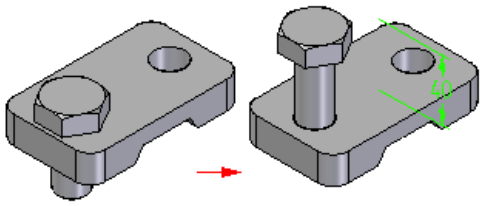



Figure 3.19: Insert assembly (fixed offset)

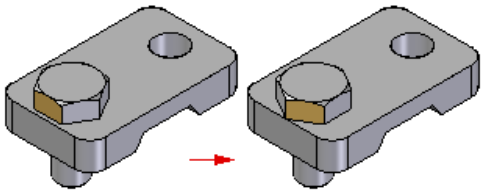


Figure 3.20: Insert assembly with additional angle relation

#### 3.6.3.7 Connect

The Connect relation come in three variants. It is used to position a keypoint on one part with a keypoint, line, or face on another part. For example, you can apply a connect relation to position the centre of a spherical face on one part with respect to a spherical face on another part as seen in figure

3.21. You can specify a positive or negative offset value with a connect relation as seen in figure 3.22. Connect relations are useful when you cannot position a part using a Mate or Planar align relation. You can also use a connect relation to position a part with respect to an element that is in a part, sub-assembly, or top-level assembly sketch.

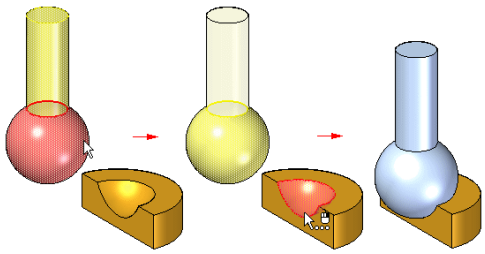


Figure 3.21: Connect assembly

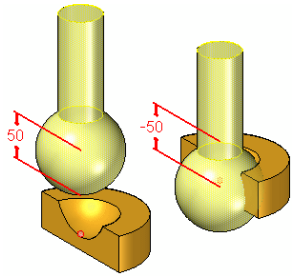



Figure 3.22: Connect assembly with fixed offset

Application is between:

1. Two Keypoints
2. Keypoint and a straight line
3. Keypoint and a planar element

#### 3.6.3.8 Tangent

A Tangent relation comes in two variants. It is used to position a cylindrical face with either another cylindrical face or a planar element. It ensures that the cylindrical face of one part in an assembly remains tangent to a cylindrical face or planar element of another part.

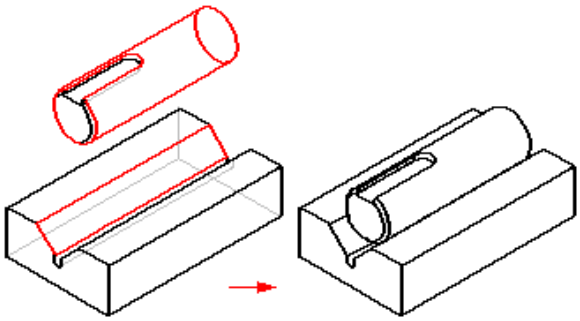


Figure 3.23: Tangent assembly relation

Tangent part faces can be in contact or be offset from each other. A fixed offset must be provided. Figure 3.23 shows the tangent relation applied between a cylinder and a planar element.

Application is between:

1. cylindrical face and cylindrical face
2. cylindrical face and a planar element

#### 3.6.3.9 Gear

A gear relation comes in three variants. It defines the ratio of relative movement between two rotating Parts, one rotating Part and another translating Part or two translating Parts. This is useful when working with assemblies that contain gears, pulleys, parts that travel in grooves or slots, and hydraulic or pneumatic actuators.

The two Parts in the gear relation must have one rotational degree of freedom and/or a translational degree of freedom remaining along the rotation axis. In the rotation-rotation gear, the two Parts must have a rotational degree of freedom and the gear defines the ratio between the rotation angles. In the rotation-translation gear, one Part must have a rotational and the other a translation degree of freedom. The gear defines the ratio between a rotation angle and a translation distance. In the translation-translation gear both Parts must have a translational degree of freedom and the gear defines a ratio of translational displacements. Figure 3.25 has examples of all three.

An example of the rotation-rotation gear relation is given between the two gears on the top of the assembly (seen in closeup in figure 3.24). Rotating the large gear will drive the smaller gear. In this case, the gear ratio is made to match the ratio between the number of teeth in the two. Note that it is perfectly correct to choose the gear ratio not to match the teeth ratio.

An example of the rotation-translation gear relation is given between the scissor-like extendible boom and the pulley at the bottom of the assembly. A displacement of the boom causes a proportional rotation of the pulley.

Finally, an example of the translation-translation gear is given between the scissor-like extendible boom and the bottom square element on the guide rail. A displacement of the boom causes an equal displacement of the square element.

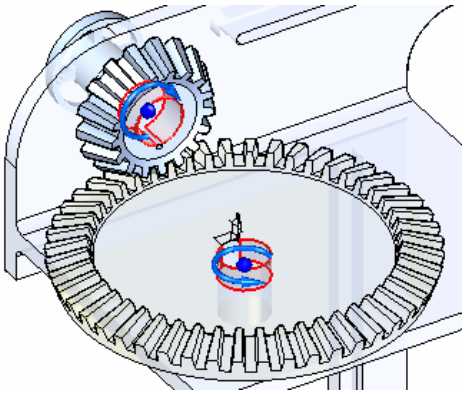


Figure 3.24: Gear relation between two gears

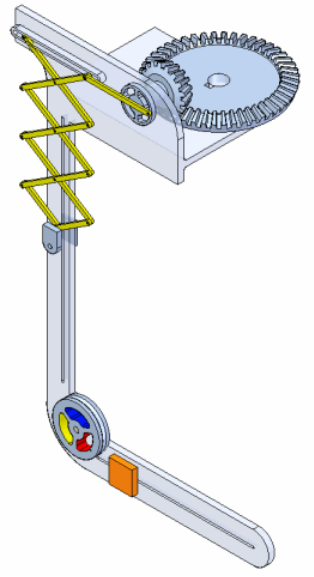


Figure 3.25: Gear relations for all three types

#### 3.6.3.10 CAM

A CAM changes the input motion, which is usually rotary motion (a rotating motion), to a reciprocating motion of the follower. They are found in many machines and toys. A CAM has two parts, the

FOLLOWER and the CAM PROFILE. Figure 3.26 shows a rotating cam pushing a follower up and then allowing it to slowly fall back down. The CAM relation applies between a closed loop of tangent faces on one part (A) and a single follower face on another part (B) as seen in the figure. The follower face can be a plane, a cylinder, a sphere, or a point.

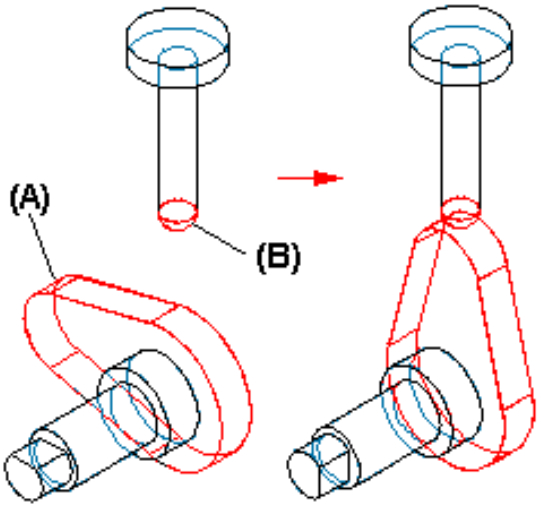



Figure 3.26: Cam assembly

When you select a planar face as the follower element, the planar face is considered to be infinite. In some cases, this may not give you the cam behaviour you want. If part geometry changes such that the closed loop of tangent faces becomes non-tangent, the relation will fail.

Application is between:

1. a closed loop of tangent faces on one part (A) and a single follower face on another part (B).

## 3.7 Selected MBS Library: Modelica.Mechanics

The library Modelica.Mechanics permits the modelling of 3-dimensional multi-body systems (MBS) with open as well as closed kinematic loops. Through the iport of MCAD data based on the STL [71] format, arbitrarily complex shaped bodies can be integrated in the model. Different interface elements ensure that MBS structures can be connected to sub-models operating in other domains (Linear and Rotary Mechanics, Hydraulics or Controls). The 3D view allows the visualisation of the model.

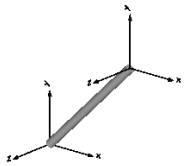
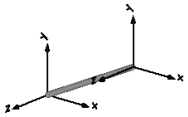
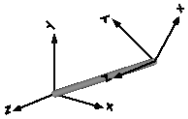
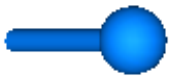
We discuss the components of the Modelica.Mechanics library including the mechanical objects (Parts package), positioning elements (Parts package) and joints (Joints Package).

### 3.7.1 Parts package

The Modelica Mechanics Parts package contains components described in table 3.8 and that we can categorise under:

- Positioning blocks: Fixed, FixedTranslation and FixedRotation
- Body blocks: Body, BodyShape, BodyBox and BodyCylinder

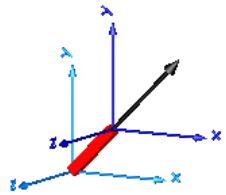
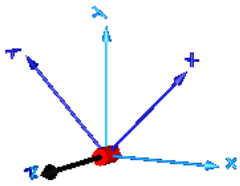
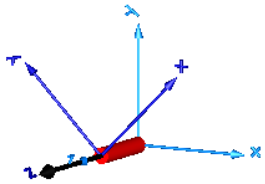
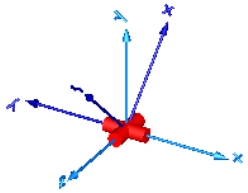
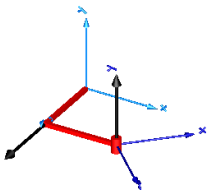
Table 3.8: Modelica Multibody Library Parts package

| ***Model*** | ***Description*** |
|---|---|
| Fixed | Frame fixed in the world frame at a given position  |
| FixedTranslation | Frame fixed in the world frame at a given position  |
| FixedRotation | Fixed translation followed by a fixed rotation of frame_b with respect to frame_a  |
| Body | Rigid body with mass, inertia tensor and one frame connector (12 potential states)  |
| Bodyshape | Rigid body with mass, inertia tensor, different shapes for animation, and two frame connectors (12 potential states)  |
| BodyBox | Rigid body with box shape and two frame connectors. Mass and animation properties are computed from box data and density (12 potential states)  |
| BodyCylinder | Rigid body with cylinder shape and two frame connectors. Mass and animation properties are computed from cylinder data and density (12 potential states)  |
| PointMass | Rigid body where body rotation and inertia tensor is neglected (6 potential states)  |

### 3.7.2 Joints package

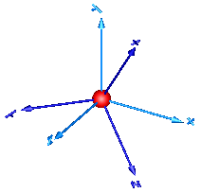
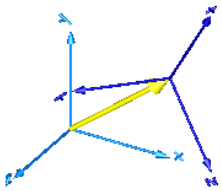
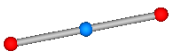
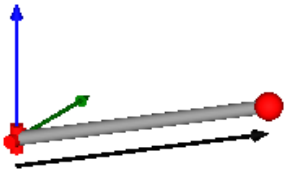
The Joints package contains models of mechanical joints. Mechanical joints define the constraints of motion between two frames of reference. These are described in table 3.9.

Table 3.9: Modelica Multibody Library Joints

| *Model* | *Description* |
|---|---|
| Prismatic<br><br>ActuatedPrismatic | Prismatic joint and actuated prismatic joint (1 translational degree-of-freedom, 2 potential states)<br>  |
| Revolute<br><br>ActuatedRevolute | Revolute and actuated revolute joint (1 rotational degree-of-freedom, 2 potential states)<br>  |
| Cylindrical | Cylindrical joint (2 degrees-of-freedom, 4 potential states)<br>  |
| Universal | Universal joint (2 degrees-of-freedom, 4 potential states)<br>  |
| Planar | Planar joint (3 degrees-of-freedom, 6 potential states)<br>  |

*continued on next page*

*continued from previous page*

| ***Model*** | ***Description*** |
|---|---|
| Spherical | Spherical joint (3 constraints and no potential states, or 3 degrees-of-freedom and 3 states)  |
| FreeMotion | Free motion joint (6 degrees-of-freedom, 12 potential states)  |
| SphericalSpherical | Spherical - spherical joint aggregation (1 constraint, no potential states) with an optional point mass in the middle  |
| UniversalSpherical | Universal - spherical joint aggregation (1 constraint, no potential states)  |
| GearConstraint | Ideal 3-dim. gearbox (arbitrary shaft directions) |

We now describe the joints in some more detail.

#### 3.7.2.1 Prismatic joint

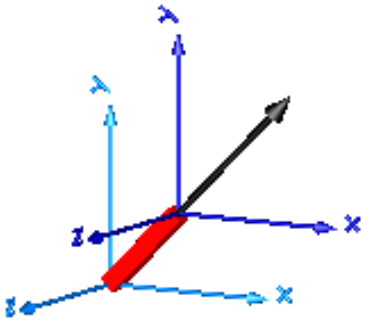


Figure 3.27: Prismatic Joint

Prismatic joint (figure 3.27) where frame_b is translated along axis n which is fixed in frame_a. The two frames coincide when the relative distance $s = 0$. The distance $s$ can be driven in the case of the *ActuatedPrismatic* joint.

#### 3.7.2.2 Revolute joint

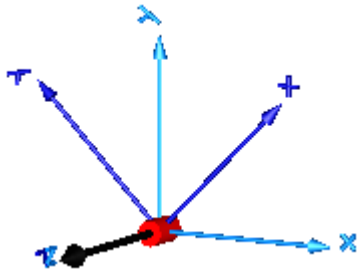


Figure 3.28: Revolute Joint

Revolute joint (figure 3.28) where frame_b rotates around axis n which is fixed in frame_a. The two frames coincide when the rotation angle "phi = 0". The angle *phi* can be driven when we use the *ActuatedRevolute* joint.

#### 3.7.2.3 Cylindrical joint

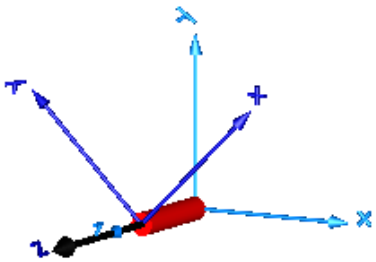


Figure 3.29: Cylindrical Joint

Cylindrical joint (figure 3.29) where frame_b rotates around and translates along axis n which is fixed in frame_a. The two frames coincide when $phi = 0$ and $s = 0$ where *phi* is the relative angle around the axis of rotation and $s$ is the distance between the two frames.

#### 3.7.2.4 Universal joint

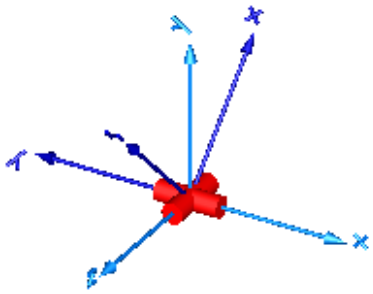


Figure 3.30: Universal Joint

Universal joint (figure 3.30) where frame_a rotates around axis n_a which is fixed in frame_a and frame_b

rotates around axis n_b which is fixed in frame_b. The two frames coincide when $revolute\_a.phi = 0$ and $revolute\_b.phi = 0$.

#### 3.7.2.5 Planar joint

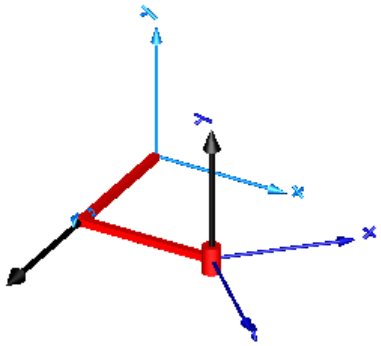


Figure 3.31: Planar Joint

Planar joint (figure 3.31) where frame_b can move in a plane and can rotate around an axis orthogonal to the plane. The plane is defined by vector $n$ which is perpendicular to the plane and by vector n_x, which points in the direction of the x-axis of the plane.

#### 3.7.2.6 Spherical joint

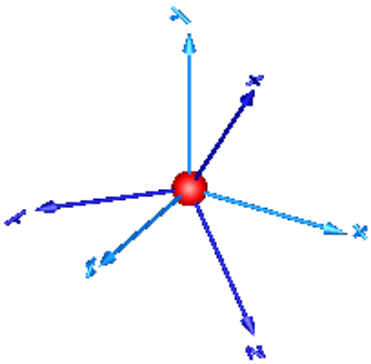


Figure 3.32: Spherical Joint

Spherical joint (figure 3.32) with 3 constraints that define that the origin of frame_a and the origin of frame_b coincide.

#### 3.7.2.7 FreeMotion joint

Free motion joint (figure 3.33) which does not constrain the motion between frame_a and frame_b.

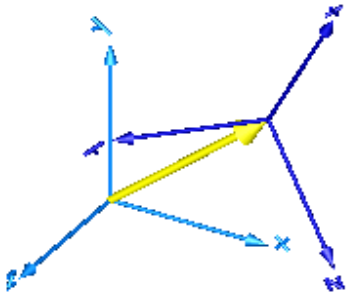


Figure 3.33: Free Motion

#### 3.7.2.8 SphericalSpherical joint

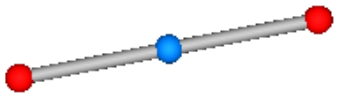


Figure 3.34: Spherical Spherical Joint

Spherical - spherical joint (figure 3.34) aggregation with an optional point mass in the middle that has a spherical joint on each of its two ends. This joint introduces one constraint defining that the distance between the origin of frame_a and the origin of frame_b is constant (= rodLength).

#### 3.7.2.9 UniversalSpherical joint

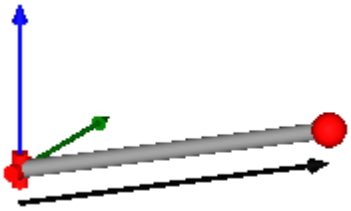


Figure 3.35: Universal Spherical Joint

Universal - spherical joint (figure 3.35) aggregation consisting of a universal joint at frame_a and a spherical joint at frame_b that are connected together with a rigid rod. This joint aggregation has no mass and no inertia and introduces the constraint that the distance between the origin of frame_a and the origin of frame_b is constant (= Frames.length(rRod_ia)). The universal joint is defined in the following way:

- The rotation axis of revolute joint 1 is along parameter vector n1_a which is fixed in frame_a.
- The rotation axis of revolute joint 2 is perpendicular to axis 1 and to the line connecting the universal and the spherical joint.

#### 3.7.2.10 Gear Constraint

Ideal 3-dimensional gearbox (arbitrary shaft directions). This ideal massless joint provides a gear constraint between frames frame_a and frame_b. The axes of rotation of frame_a and frame_b may be arbitrary.

# 4

# BIDIRECTIONAL CAD & DYNAMICS INTEGRATION

## 4.1 Introduction

In the process of designing a mechanical system, defining the full geometry is a process that usually comes in the later stages. The important elements to consider from the onset are those restricted to describing the dynamics behaviour and these include the physical dimensions of parts, approximate mass and inertia, positions of the various assembly joints and the modelling of the environment with forces and torques.

The user starts by developing a MBS model with the dynamics tool thereby specifying the overall physical dimensions, joint types and attach points all without any detailed geometry having to be provided. The geometric shapes remain a secondary concern until a certain level of maturity and stability in the physical dimensions (of the mechanical parts) is reached.

On the other hand, MCAD modelling tools provide insight into mechanical systems that block-diagram or text-based simulation and modelling languages lack. The latter do not provide detailed insight into the geometry of the objects involved nor are they very suitable in defining relations between or on geometric features. Integrating the two modelling approaches provides benefits to both.

**Current Situation Unidirectional** Review of the current state of the art found many conversion tools that can extract the necessary MBS information from the MCAD and generate a model for dynamics simulation. This is illustrated in figure 4.1 as the "Existing data transfer direction". The review also failed to find any conversion mechanisms going from a Dynamics simulation model to the MCAD or in the case of a dynamics model generated from the MCAD model, allowing changes to the dynamics model to affect in some way the source MCAD model. The process is currently unidirectional.

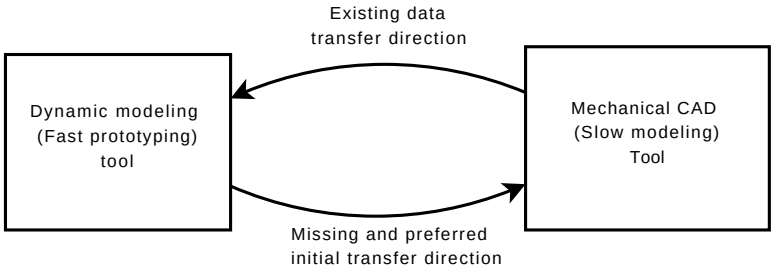



Figure 4.1: Dynamic modelling tool interface with MCAD modelling tool

**Uni-directionality Consequences** The lack of a reverse mapping, illustrated in figure 4.1 as the "Missing and preferred initial transfer direction", hinders the efficiency gains we can achieve if we could use the Block-Diagram multi-body modelling capability irrespective of the existence of a CAD model or

in parallel.

This unidirectional approach is not ideal as we are now burdened by the weight of maintaining the MCAD model and at the same time we have lost the rapid-prototyping capability provided by the multi-body dynamics modelling tool. It is much faster to carry out the preliminary design work by concentrating on the physics only. The MCAD tool on the other hand forces the user to give consideration to shape information whereas the dynamics tool only requires component (or Part) mass properties (mass, inertia, centre of mass), joint positions and link information.

**Inefficiency of not being able to convert from Dynamics to MCAD** Although the dynamics modelling tool is the tool where modelling a new MBS system, exploring alternative designs or making small changes can be made fastest and usually with the least effort, because of the lack of mechanisms to export or map this information into the MCAD model we lose some of these advantages. This is due to the fact that we are discouraged from working in the dynamics model as we would have to recreate the system in the MCAD tool from which we will export back to the dynamics model. This is one inefficiency.

**Inefficiency of not being able to maintain consistency between MCAD and Dynamics** Another inefficiency occurs once we set the CAD model as the source/baseline and use it to automatically generate the dynamics model. From that point on we are forbidden or at least discouraged from making modifications to the mechanical information in the dynamics model. Manually maintaining consistency is an option but is far from optimal. As a consequence we lose the major benefit of the dynamics tool which is its rapid-prototyping capability.

**Thesis Objectives** This is therefore the focus of this thesis: to develop and demonstrate a viable conversion mechanism from a dynamics model to CAD as well as a conversion mechanism from CAD to dynamics such that both conversion mechanisms are compatible and could pave the way for implementing a mechanism to maintain parallel evolution of the models. We provide more details in section 4.3.

## 4.2 Context

The natural design process of any system starts with low-detail models and proceeds to medium and finally high detailed models. For example, when a car is to be designed, the initial model may state how many passengers it must carry, what should the maximum speed be, etc. Some quick calculations may then determine the overall mass of the car and the engine capacity. A simulation model may be created to model the car's behaviour given these few inputs. For the initial investigations, this model may be enough. Once suitable initial design parameters are found and agreed upon, the engineers can proceed to developing more detailed models in similar iterations within the same tool or using new and more capable tools. At the final stage, detailed designs of every single component and subsystem would have been generated. There is a similar hierarchy in the modelling of mechanical structures.

Block-Diagram based multi-body simulation tools can be considered as rapid-prototyping tools and are therefore akin to low-detail (or low-information) models. The next level of detail comes with the generation of the associated CAD models where the geometry of the bodies is defined. When the geometry is available, we can proceed to the creation of Finite-Element models which we can use to carry out more detailed mechanical simulation either in static or dynamic configurations. The Finite-Element model can be considered to be the highest level of detail model (from the perspective of mechanical information only). A progression of tools then is to go from Block-Diagram (or low-level) to CAD and then to Finite Element Model (FEM) in sequence.

To put our thesis in perspective, we provide a quick overview of Block-Diagram, CAD and Finite-Element modelling tools in terms of how they are related to each other. From that overview we show the existence of a certain symmetry in the relations between these models and the missing mechanisms to convert models into each other. The missing mechanisms are required to link/associate models to each other to enable parallel development and therefore are an inefficiency to the users that must be addressed. Our thesis therefore addresses part of this problem and we will provide the details in the following sections.

### 4.2.1 Block Diagram modelling

Block-diagram based modelling tools can be used to describe the dynamic behaviour of many physical domains. Some of these tools are domain-specific such as Pspice for electronics and some are domain-agnostic such as Simulink from MathWorks or the Modelica language. Some modelling approaches permit a greater resemblance between the model topology and the physical system as is the case with Pspice or some Modelica libraries. Simulink on the other hand requires the model to be organised in functional blocks and the resemblance can be less striking or inexistent with the physical system modelled. The components of a Pspice electronic circuit with elements such as resistors, capacitors and others will in most cases be mapped to matching physical counterparts on a printed circuit board. Similarly, using Modelica libraries such as the electrical, thermal and mechanical libraries will generate models that bear a close association or mapping with the individual components of the physical implementation. We classify such tools under the heading of **physical-based modelling tools** when we see a close resemblance between model elements and the corresponding physical implementation. We have discussed these in detail in the literature review.

### 4.2.2 MCAD modelling

MCAD models combine geometric shape information as well as geometric relations between various components and can be used for many purposes. As FEA tools demonstrate, the geometry is but a gateway for defining further relations and attributes on the geometries with applications in mechanical, thermal, electrical, fluid flow, electro-magnetics and other domains. In the mechanical domain we use the geometry to define the mechanical properties as well as boundary conditions. We then construct an FEM to carry out detailed deformation and stress analyses. In the thermal domain, the FEM model is used to calculate with a fine spatial resolution the propagation of heat and temperature variations.

MCAD models can also be used to generate various other forms of simulations. An example is the use of MCAD assembly models to generate a MBS model to run in an external tool. This is demonstrated in the Solid Works to SimMechanics model translator where the mechanical behaviour of the assembly could then be simulated in Simulink.

Another example from the electromagnetic domain is the CST Studio Suite by “2010 CST Computer Simulation Technology AG” where a 3D model can be used in two opposite ways. One method will refine the geometry by defining a FEM mesh to carry out an FEA. The other would reduce the geometry by creating a lumped model to achieve the same effect but usually at the cost of reduced fidelity but increased simulation speed. This example described various uses of a MCAD model including for lumped and finite-element simulations.

### 4.2.3 Finite Element Modelling

FEM based tools can model the behaviour in the same physical domains as block-diagram tools can (FEM can cover all physical domains because we can choose to model that domain using the geometries involved and the physics that govern them, which may or may not be the best approach) and more

(block-diagram models cannot handle, or more accurately are not suited to handle, all cases that FEM models can. For example, a thermal simulation including radiative effects definitely requires the full geometry to gather all the parameters needed to carry out the simulation). This is demonstrated well by a multi-disciplinary FEM tool like ANSYS Multiphysics which can carry out the FEA in the electrical, mechanical, fluid flow, thermal, electromagnetic and other domains. Given that FEM are based on an underlying MCAD model, the relation between the CAD and the FEM is clear.

FEMs take a distinctly different approach compared to Block-Diagram methods by using extended (or distributed) instead of lumped components. For example, the MCAD model of a cylindrical tube is converted to a FEM mesh with hundreds or thousands of variables. Comparatively, in the block-diagram or lumped approach, the tube is converted to a much simpler differential equation with a few state variables. This simplification usually comes at the cost of a reduction in the fidelity of the simulation but may be very appropriate given the type of analysis that interests us.

### 4.2.4 Model Symmetry

The physical domains we can simulate using Block-Diagram models will usually have a corresponding FEM tool. The mechanisms to convert from one model to the other or for working on them in both kinds of tools in parallel or at least in a collaborative, sequential manner may not exist.

With the CAD geometry being an engineering domain-neutral representation, it has the potential to be linked to both Block-Diagram and FEM models. A symmetric relation arises then between Block-Diagram, CAD and FEM models which is seen in figure 4.2.

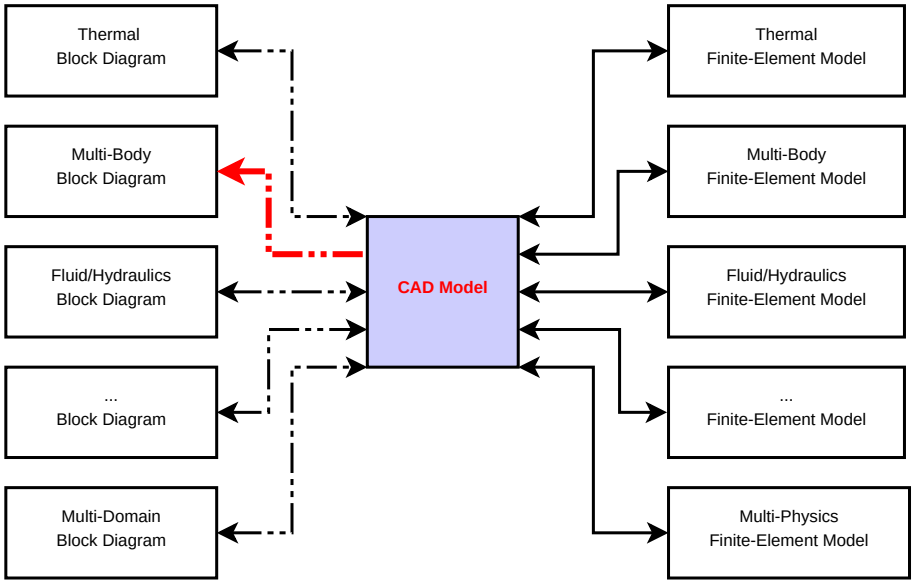



Figure 4.2: Block-Diagram and Finite-Element Model Symmetry

### 4.2.5 Bidirectional Mappings

The normal design and development process would normally proceed from left to right in figure 4.2 as the Block-Diagram modelling approach is usually the fastest to develop. Followed by the CAD model once the physical dimensions and material properties are decided or have matured. Finally, the FEM are developed for the highest fidelity simulations and analyses. This describes the overall problem. We will focus on the subset marked by a red arrow.

In general, bidirectional relations on the right hand side between CAD and FEM are well established in many commercial products (ex. ANSYS MultiPhysics). Relations on the left hand side of the figure are less mature and/or less common. These relations may exist bidirectionally, uni-directionally or not at all for a particular physical domain and is indicated by the dashed arrows connecting various block-diagram models to the CAD model.

There are many benefits to a bidirectional association between models, the main one being the capability to work on both models in parallel while staying consistent with the other. This is the reason why for example current CAD and FEA tools have well-established model association which allows for both models to develop in parallel and incrementally. The lack of such bidirectional relation is detrimental to the efficiency of the engineers in their work and we have such a situation when we consider CAD and Block-Diagram modelling tools.

In the following sections we will discuss the Block-Diagram to CAD relation in more detail by concentrating primarily, but not exclusively, on the mechanical domain.

## 4.3 Contributions

We provide a general resume of the thesis contributions followed by a more detailed breakdown.

$1^{st}$ **contribution - Combining MCAD & Multi-domain** Going beyond the mechanical domain, we discuss how the MCAD modelling tools can be linked to multi-domain block diagram simulation models by incorporating elements of BD modelling such as general block interfaces and connections into the MCAD tool.

This will be the first contribution.

$2^{nd}$ **contribution - MCAD to Modelica Mapping** In the mechanical domain, in order to implement a bidirectional mapping between the MCAD and the MBS dynamics models, we must first determine the basic elements needed in the Modelica library to achieve this. Once these elements are defined we can then further develop the details of the correspondence map as well as providing a concrete demonstration of the CAD exporter we have developed.

This will be the second contribution.

$3^{rd}$ **contribution - Bidirectional Dynamics and MCAD Mapping** The creation of the Modelica MCAD equivalent dynamics modelling library took into consideration the bidirectional mapping requirements and therefore the dynamics to MCAD mapping should prove to be a simple process. We will provide the mapping details. Although the mapping is a simple one, this part of the contribution is indeed the important one as it would be a first implementation of a reverse Dynamics to CAD mapping whereas all existing mapping are from CAD to Dynamics direction. This addition then eliminates the first inefficiency by providing the capacity to automatically convert the dynamics model into a baseline skeletal CAD model. This was illustrated in figure 4.1 as the "Missing and preferred initial transfer direction" arrow.

This will be the third contribution.

$4^{th}$ **contribution - Direct Modelica.Mechanics to MCAD mapping** With the Modelica library we created to provide a one-to-one correspondence with the MCAD model, we have done this at the cost of creating a layer on top of the existing Modelica.Mechanics library. We develop the mapping from the Modelica.Mechanics library directly to the MCAD model at the cost of complicating the mapping between the two models.

This will be the fourth contribution.

## 4.4 Options Analysis

To narrow down the avenues of exploration, we will analyse the options we have in terms of the tools we will work with and more importantly the different ways we can map MCAD and MBS dynamics models to each other. We explore some of the possibilities in linking geometric and dynamics models. We will then take a subset of these and provide an implementation.

### 4.4.1 Bi-directionality Implications

To achieve a bidirectional mapping between two models of different domains or different languages, a certain degree of similarity is required between the associated parts. The relations mapping the elements to each other must be manageable. More importantly, from the set of all legal changes allowed on either model when they are independent, the subset permissible in the context of the combined system must be clear and well-defined. Under these restrictions, it is desirable that for any evolution of either model, that there always be a corresponding evolution of the other.

#### 4.4.1.1 Consistency between dissimilar languages

The implications of bi-directionality on consistency between dissimilar languages can be illustrated by considering two initially consistent models - a C program and its corresponding Assembly code - which we will be modifying manually and independently.

How do the relations between C and Assembly codes or the association between lines of C-code and groupings of Assembly-code have an impact on the requirement of maintaining bidirectional consistency? The kinds of relations we have in this case preclude or make it very difficult to implement such a bidirectional consistency algorithm.

In order to implement bidirectional consistency we require that any (or most) legal changes on the C-code can be transformed to legal changes on the assembly and vice-versa. Further, we require that when both models (C-code and assembly-code) have been modified, recovering consistency is a manageable proposition. Of course, any legal changes to C can be translated to Assembly by compilation. However, the impact on the Assembly code of a certain change in C can vary dramatically. We can assume that certain changes to C will produce localised changes to the assembly. But, there are also small changes we can make in C that cause major changes throughout the Assembly code. For example, changing the definition of a C-structure will probably cause a change everywhere where that structure is used. Changes to a header file that is used often can also have a similar wide-ranging impact on the generated Assembly code.

However, even assuming that this is not a problem, there is the more pertinent problem of how changes to the Assembly code affect the C-code. This reverse operation has a different problem which follows from a difference in the granularity of the two languages. Making modifications to the Assembly code and expecting to modify the C-code consistently can be done only under very strict conditions. Simple operations such as changing the value of a variable and the test condition of a branch or a for-loop can be propagated back to the C-code. Anything more complicated will easily create a situation where there is no equivalent C-code. We could allow Assembly code to propagate back removing the offending C-code, replacing it by the Assembly code and producing a mixed C and Assembly code, but this would be very undesirable. Therefore, there are many legal modifications a user can make to the Assembly code but most of them will make it impossible to modify the C-code consistently and modifications to the assembly with consistency in mind are impractical and too constrained to be of any use.

Therefore, we can advance the following recommendations:

1. Differences in granularity between two models will cause many problems for the implementation

of a bidirectional consistency algorithm that can rapidly become insurmountable or make it impractical to use.

2. For each legal change on either model, a corresponding legal change, including "no change" as a legal change, must be available for the opposite model otherwise consistency cannot be maintained continuously.

#### 4.4.1.2 Dynamics-Geometry Bidirectional Consistency

In order to implement bidirectional consistency between the geometric model and its corresponding dynamics model, we have to carefully select the dynamics modelling language or meta-model. A dynamics modelling language that won't permit a quasi-isomorphic mapping of the common information and the operations on the models will make it difficult or impossible to implement bidirectional consistency.

When the geometry describes a mechanical assembly, the choice of the appropriate dynamics modelling language can be narrowed down by examining the chain of models linking the mechanical geometry to the dynamics simulator model. As described in the literature survey, there is a minimum of three models or components involved:

1. The geometric model of the mechanical assembly providing a geometric representation of the parts composing the assembly, a geometric representation of the constraints and the mechanical properties of each part such as centre of mass, mass and inertia.

2. An abstraction that discards the geometric information and uses mathematical equations (DAE) to represent the dynamics of the assembly. This model requires the mass properties of the parts, the types of constraints imposed on them and their geometric positions in order to construct a complete set of equations describing the mechanical system.

3. The DAE analysed and converted into a mathematical procedure that can calculate/simulate the mechanical behaviour.

**Geometry-Equations Relation** Converting the geometric/mechanical information (item 1) into equations (item 2) is the first step towards producing a simulation and there is a clear and simple mapping or association between elements of geometry and geometric constraint on the one hand and equations on the other. When the equations are grouped together appropriately into components, the mapping between the mechanical model components and the dynamic model becomes isomorphic component-wise (Geometry and geometric constraints are separate components in the mechanical model).

Figure 4.3 shows these relationships between the mechanical assembly and its Modelica/equations counterpart. The mechanical/geometric parts tagged 1 to 6 and Load on the robot (left) correspond to model element b0 to b6 and load respectively (right). The six axes (geometric constraints) identified on the left as **axis1** through **axis6** correspond to model components/elements **r1** through **r6** on the right.

This simple relationship allows us to match the Create/Update/Delete (CrUD) operations on any geometric elements on the left with corresponding CrUD operations on the right.

**Equation-Procedure Relation** The same simple relation does not hold between the equations (item 2) and the solution procedure (item 3). The procedure or simulation code that is generated automatically from the Modelica model or equation components in figure 4.3 is seen partially in figure 4.4. Although the algorithm that produces the procedure is known exactly and we can trace elements of the equations to elements of the procedure, these relations are complex. Small segments of code in the procedure are

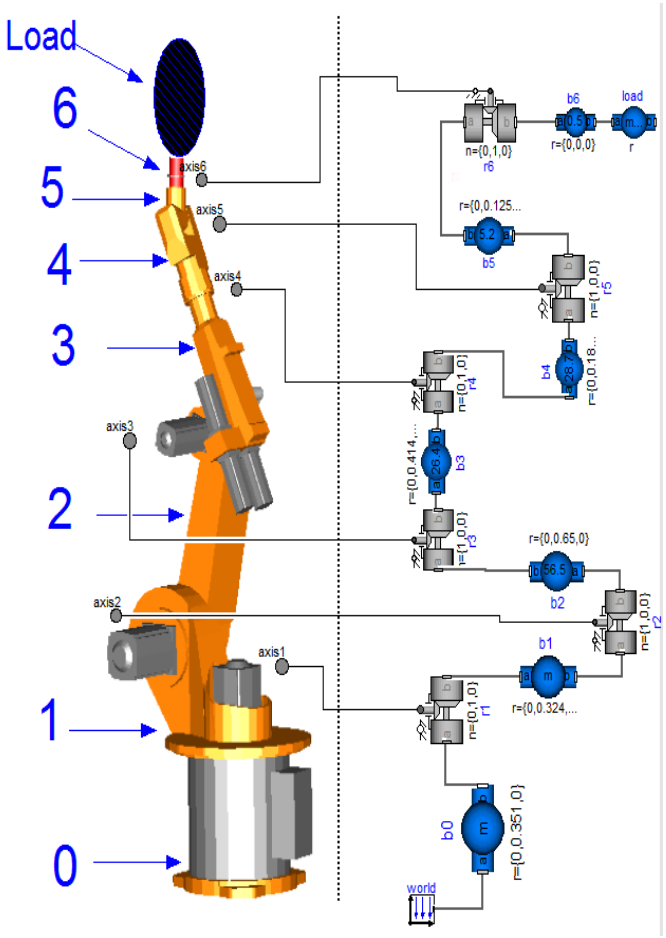



Figure 4.3: Association of Geometric and Dynamic Components

associated with multiple elements in the equations to an extent that render impractical manipulating the procedure for any useful purposes. This resembles the relation we had between C-code and its Assembly counterpart as discussed earlier.

CrUD operations on a single geometric element or its corresponding equations will usually affect many parts in the procedure in a complex manner and probably in a non-localised fashion. In the reverse direction, most changes to the procedures allowed by the language used will not correspond to any physically realisable mechanical assembly. Consistency in the reverse direction is impractical and not particularly useful. Thus, bidirectional consistency between either **geometric elements** or **equation groupings** and **procedure code** is both impractical and of limited use.

#### 4.4.1.3 Recommendations

We put forward the following recommendations. We need to construct a MBS ⇔ Solid Edge mapping where the granularity of the components used on either side are similar together with a correspondence map valid for all CrUD operations on either model. Achieving an isomorphic mapping between elements of either model would be an ideal approach if possible.

```
/* DSblock model generated by Dymola from Modelica model
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot
 Dymola Version 7.0, 2008-02-27 translated this at Wed May 27 12:27:38 2009

  Advanced.NewStateSelection = true; // Due to used package.
 */
...
...
...

   /* Linear system of equations to solve. */
  W_[2658] = RememberSimple_(W_[2658], 44);
  SolveScalarLinearMixed((IF W_[2661] THEN 1 ELSE 0)+W_[2734]*(IF W_[2661] THEN 0
     ELSE 1),"(if axis1.gear.bearingFriction.locked then 1 else 0)+axis1.motor.Jmotor.J*(if
     axis1.gear.bearingFriction.locked then 0 else 1)",
     W_[2733]-((IF W_[2661] THEN 0 ELSE IF W_[2659] THEN (PushModelContext(1,
    "Modelica.Math.tempInterpol1(axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1(W_[2671], RealTemporaryDense( &W_[2650], 2, 2, 2),
     2)) ELSE IF W_[2660] THEN  -(PushModelContext(1,"Modelica.Math.tempInterpol1( -
     axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1( -W_[2671], RealTemporaryDense( &W_[2650], 2, 2,
     2), 2)) ELSE IF PRE(W_[2667], 15) == 1 THEN (PushModelContext(1,
    "Modelica.Math.tempInterpol1(axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1(W_[2671], RealTemporaryDense( &W_[2650], 2, 2, 2),
     2)) ELSE  -(PushModelContext(1,"Modelica.Math.tempInterpol1( -axis1.gear.bearingFriction.w,
     axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1( -W_[2671], RealTemporaryDense( &W_[2650], 2, 2,
     2), 2)))-W_[2648]+W_[2734]*(IF W_[2661] THEN 0 ELSE IF W_[2659] THEN  -
    W_[2655] ELSE IF W_[2660] THEN W_[2655] ELSE IF PRE(W_[2667], 15) == 1 THEN
     -W_[2655] ELSE W_[2655])),"axis1.motor.Jmotor.flange_a.tau-((if axis1.gear.bearingFriction.locked
     then 0 else (if axis1.gear.bearingFriction.startForward then
     Modelica.Math.tempInterpol1(axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)
     else (if axis1.gear.bearingFriction.startBackward then  -Modelica.Math.tempInterpol1( -
     axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2) else (if pre...",
     W_[2658],"axis1.gear.bearingFriction.sa");
  W_[2654] = IF W_[2661] THEN W_[2658] ELSE IF W_[2659] THEN (PushModelContext(1,
    "Modelica.Math.tempInterpol1(axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1(W_[2671], RealTemporaryDense( &W_[2650], 2, 2, 2),
     2)) ELSE IF W_[2660] THEN  -(PushModelContext(1,"Modelica.Math.tempInterpol1( -
     axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1( -W_[2671], RealTemporaryDense( &W_[2650], 2, 2,
     2), 2)) ELSE IF PRE(W_[2667], 15) == 1 THEN (PushModelContext(1,
    "Modelica.Math.tempInterpol1(axis1.gear.bearingFriction.w, axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1(W_[2671], RealTemporaryDense( &W_[2650], 2, 2, 2),
     2)) ELSE  -(PushModelContext(1,"Modelica.Math.tempInterpol1( -axis1.gear.bearingFriction.w,
     axis1.gear.bearingFriction.tau_pos, 2)")
    Modelica_Math_tempInterpol1( -W_[2671], RealTemporaryDense( &W_[2650], 2, 2,
     2), 2));
...
...
W_[2544] = IF fabs(W_[2500]-W_[2506]) > 1E-014 THEN W_[2577] ELSE false;
if (NewParameters) {
helpvar[852] = fabs(W_[2501]-W_[2507]);
}
W_[2545] = IF fabs(W_[2501]-W_[2507]) > 1E-014 THEN W_[2577] ELSE false;

DefaultSection
InitializeData(1)
EndTranslatedEquations
```

Figure 4.4: Extract from Robot Simulation Code

### 4.4.2 Tool Selection

Before providing an implementation/proof-of-concept of a bidirectional link between a MCAD model and a MBS dynamics modelling tool, we first need to select the appropriate tools.

**Solid Edge** For this thesis we have selected Solid Edge for MCAD modelling. Solid Edge is a suitable tool as it compares well to other leading MCAD tools and provides a well documented Application Programming Interface (API) for interfacing with its internal model. More importantly it was the best MCAD tool available to the author.

**Modelica Mechanics** A dynamics modelling and simulation language was required that could isomorphically represent the entities in a CAD model including the Parts, Assemblies and the Assembly Relations. The MBS library for Modelica [53] provides a promising starting point. Its object-oriented nature comes close to delivering such an isomorphic and bidirectional mapping. Alternative tool combinations could have also been used as a starting point and these were discussed in the literature review

sections of this thesis.

### 4.4.3 Tool/Model Customisations And Restrictions

In order to establish a bidirectional association between the Modelica MBS and the Solid Edge MCAD tool, we first need to understand the relevant components on either side.

Both environments provide the means to construct multi-body systems using parts and joints however the joints in either environment do not map completely to joints in the other as they currently stand. Therefore, we first need to identify the differences and then work out the optimal approach in building this bidirectional association.

On the Modelica side there is the standard Mechanics (or MBS) library that provides a starting point for creating mechanical systems connected with a variety of joints. These joints were constructed using the Modelica language and can be modified or adapted further if required. Thus we have an adaptable library.

On the Solid Edge side we are given a tool where the definition and behaviour of its geometric elements (parts and assembly constraints) cannot be changed. There is no capacity to create a new type of joint or redefine existing ones. We are given the tool and we do not have the capacity to modify the geometric engine or the geometric constraints provided. At best we can create Parts and Assemblies that could map to some elements in the MBS. Therefore SE is not very adaptable for our purposes and we will try to avoid any customisations whenever possible.

Given these two facts, we are encouraged to concentrate our efforts on customising/adapting the Modelica MBS library to meet the bidirectional mapping requirements as well as compensate for the limited adaptability of Solid Edge.

### 4.4.4 Parametric Relations

Numeric parameters play an important role in today's MCAD tools by providing parametrised Part models. The parameters can either drive physical dimensions directly, act as inputs to algorithms that generate geometry or be simply annotations not used for geometric purposes but for specifying details of the Part being modelled.

These parameters can be shared or more generally some functional mapping can be specified between the parameters of both models in either direction. Many types of relations are possible, but here we list three that are shown in figure 4.5.

**Equality Relation** A simple relation is to have an equality relation between parameters in models A and B and in the case where an inconsistency has been introduced, to select one model as the reference and to update the other as the user requires.

**Functional Relation** A slightly more complex example is to have the parameters in model A dependent on the parameters in model B via mathematical functions instead of simple equalities. This defines a causality relation between the parameters involved from model B to A. The function f1 in the figure is not given with an arrow as we can have $f1 : A \rightarrow B$ or $f1 : B \rightarrow A$. This could be used to update the model B parameters when those in model A are updated. Or, allow parameters in model B to be modified, and try to determine model A parameter values that satisfy the relation $f1 : A \rightarrow B$ (or vice-versa).

**Equation Relation** Another possibility is to define an equation that links parameters from both models. Unlike a function where some of the parameters are causally dependent on others, an equation can be considered as a constraint that the combined set of parameters must satisfy.

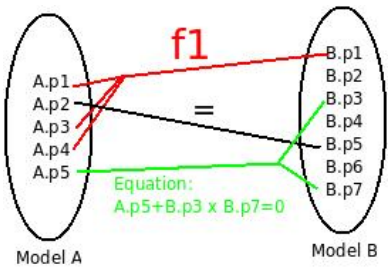



Figure 4.5: Parameters of two models connected via equalities, functions and equations

We provide a concrete example of these relations between a MCAD and Modelica models of a closed tank shown respectively in figures 4.6 and 4.7. In this example, the CAD model doesn't have a unique variable that defines its internal cavity volume. Instead, three parameters of length $L$, width $W$ and height $H$ are used. In the Modelica model a single parameter $V$ defines the volume. This provides an example of an **equation** relation between the two models with the equation being $V = L \times W \times H$.

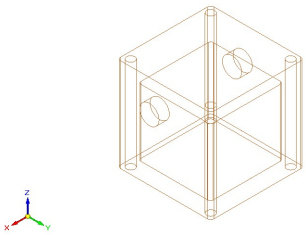


Figure 4.6: Transparent tank with inside volume visible

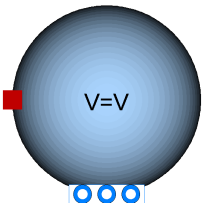



Figure 4.7: Modelica model of a closed Tank

### 4.4.5 Parameter Categories

We can classify parameters into categories.

#### 4.4.5.1 Intensive, Extensive and Path Parameters

We can differentiate between intensive and extensive and path parameters.

**Intensive parameters** Intensive parameters do not depend on the shape but only on the local properties. Density, electrical resistivity and friction coefficients are all intensive parameters.

**Extensive parameters** Extensive parameters on the other hand are parameters like volume, mass and inertia as they depend on the global properties (including shape) of the object. We can consider these to be geometry dependent or derived parameters although MCAD tools usually give the user the option to override them with user-defined values.

**Path parameters** From the perspective of trying to leverage information stored in a MCAD model for use in dynamics we can imagine a more complicated case of a dependent parameter. For example, if we are concerned about lumped behaviour of MCAD Parts, we can calculate the electrical resistance, the thermal conductance or other physical parameters between two interface points on the Part. If these interface points are defined as the contact points with other parts, then under mechanical motion these could be moving in which case the lumped values may be changing as well. This type of parameter is

harder to use but could potentially have its uses. We call it a Path parameter.

#### 4.4.5.2 Dimensional Parameters

Solid Edge and Solid Works make use of parameters to drive the physical dimensions of the geometry. As an example, the MCAD tool is used to construct a parametrised model of a gear. **Dimension parameters** that determine its shape such as inner and outer diameters, gear thickness, number of teeth and various other dimensions provide an abstraction useful in non-geometric models. The user can then directly assign a numeric value to these parameters or define them as some a function of other parameters. Figures 4.8 and 4.9 provide two views of the same model where some of the dimension parameter were modified. Note the change in the thickness and length of the notch from the centre of the gear.

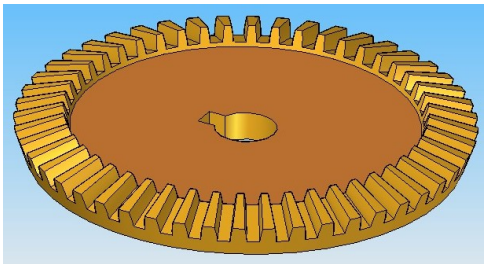


Figure 4.8: Initial Parametric model

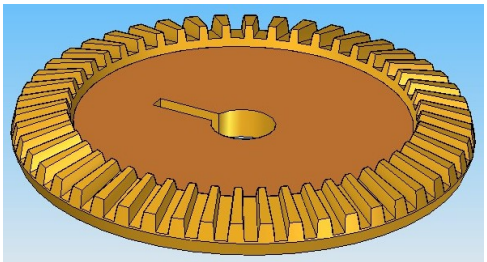


Figure 4.9: With dimensions modified

#### 4.4.5.3 Structural parameters

We loosely define as **structural geometric parameters** those parameters that change the geometry in a way that cannot be achieved by changing dimensional parameters. For example, the dynamics model of a gear may have a parameter defining the number of teeth. This parameter could be used, together with other mechanical parameters, to determine maximum torque allowed to avoid damaging the gear, estimating gear ratios, etc. Sharing this parameter with the MCAD model would mean that we require the geometry to change accordingly.

CAD tools like CATIA from Dassault and NX from Siemens provide advanced features in order to drive the construction of geometry. As the user is constructing the geometry the tool can record the commands and produce code (Microsoft VBA). This code can then be recalled to quickly generate the same structure. The user can modify the code, insert conditional statements or generate a new custom code from scratch. Functions can be used to automate any and all operations that the user could achieve by manually interacting with the graphical user interface of the MCAD tool.

Specifically for our purposes these can be turned into functions and called with parameters to drive the generation of the geometry. These can be made as complex as required. A simple use is to set a parameter defining the number of teeth in the gear and the function will generate the correct geometry. This is visualised in figures 4.10 and 4.11 where the number of teeth was changed from 24 to 48.

#### 4.4.5.4 Annotative parameters

Parameters do not necessarily need to drive the geometry. The MCAD model is often the starting point for analyses in multiple engineering disciplines including thermal, mechanical, electrical and others. As such, the designer of a MCAD Part often wishes to associate additional information to the Part that has no immediate use in defining its geometry. These can be broadly classified as annotations and can be numeric for dimensioning or not, textual, or even contain links to external objects such as

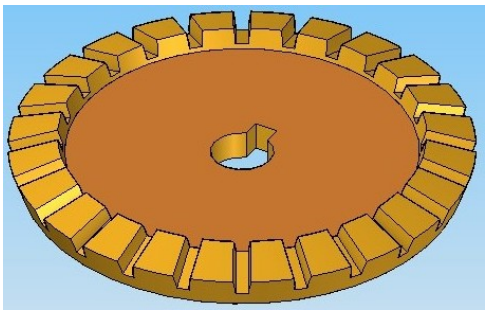


Figure 4.10: 24 tooth gear

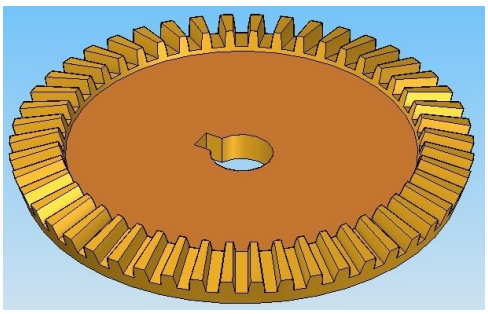


Figure 4.11: 48 tooth gear

file attachments.

For example, textual annotations can be used to record comments from various experts analysing the Part and later brought out during group discussions before finalising the geometry. The material properties with which the geometry is to be fabricated can be recorded in such parameters and be used later in mechanical stress analyses, thermal simulations and machining. For our purposes, such parameters (specially numeric ones) can be shared with the dynamics model.

### 4.4.6 Constraint Mapping Alternatives

We have several options when generating the dynamics model w.r.t. the handling of the mechanical assembly relations. When there are no assembly relations between two Parts, the relative mechanical degrees of freedom is six: three of displacement and three of rotation. As we add one assembly relation after another, the degrees of freedom are reduced till none remain. This corresponds to fixing the Parts relative to each other and is equivalent to a single connection between two reference frame connectors in Modelica.Mechanics. Two examples of **fixing** Bodies relative to each other were given in figure 4.24.

Whereas in the Solid Edge MCAD model the individual constraints we insert are remembered (and not combined into their resultant constraint), we may choose not to follow this approach when converting the constraints to the dynamics model. We therefore have a minimum of two options when mapping from MCAD to BD which we will now discuss.

Figure 4.12 with its multiple sub-diagrams will serve as a reference for this discussion. The $1^{st}$ or top-left class diagram in the figure shows the entity relations of a MCAD model with two Parts linked by two Assembly relations. **Geometry 1**, **Geometry 2** and **Geometry 3** appear in both **Part A** and **Part B**. The remaining three figures show various Modelica mappings we could select from.

#### 4.4.6.1 Mapping Constraints Individually

This option corresponds to mapping individual Assembly Relations to corresponding joint components in the dynamics models following the idea of maintaining an isomorphic mapping between the elements of MCAD and the MBS model. It can be implemented by exporting both geometric features as well as individual Assembly Relations to corresponding elements in the simulation model.

The benefit of the individual-mapping approach is its simplicity an clarity in terms of managing the parallel evolution of the models including the Parts, Assembly Relations and Geometry: creating, updating or deleting an element in either the MCAD or simulation models has a clear and unambiguous equivalent in the opposite model.

One important disadvantage lies in the fact that when more than one assembly relation exists between

two MCAD parts, mapping individual assembly relations to their equivalent in the MBS model will often lead to redundant constraints (or equations) which need to be corrected. Fortunately, the problem is well understood and a solution has been demonstrated in [70]. Although Elmqvist et al's solution for removing redundant equations is not implemented in our design, it is compatible with the **individual mapping** approach and therefore can be added as a post-processing step.

There are two interesting variants of this mapping which are equivalent from an equation perspective but have different implications for model evolution. We discuss them here:

**Geometry-Geometry joints** The first variant maintains the element partitioning found in the MCAD model. i.e. the geometry is stored with the Part information and the Assembly relation is a separate element. This is seen in the $2^{nd}$ or top-right Modelica class diagram (Option 1) in figure 4.12 showing the MBS model and that it is identical to the MCAD class diagram in the $1^{st}$ diagram. The joints (Assembly Constraint x and y) each connect two geometries. We say that these joints are of type **Geometry-to-Geometry**. From a model evolution perspective, this partitioning method results in a clean evolutionary link between MCAD and MBS models: the geometric and mass properties evolution of the MBS Part depend only on the evolution of the corresponding MCAD Part.

**Part-Part joints** The second variant, contrary to the previous approach, doesn't require that geometry be stored in the MBS Part model itself. Therefore, the MBS Part model will be updated only if the the mass properties of the MCAD Part change but not when its geometry changes. The geometry information is also mapped selectively only when it is used in an Assembly relation in the MCAD model. Therefore, in the conversion process this geometry is made part of the generated constraint as is seen in the $3^{rd}$ diagram (Option 2) of figure 4.12 at the bottom-left. This approach has the advantage of being minimalistic in terms of what geometry information is exported to the MBS model as opposed to the previous method where any and all geometry information in a MCAD Part was exported. Only geometry required by the assembly relations in use is exported. Notice that indeed some of the Geometric features (Geometry 2 on the left and Geometry 1 on the right) have not been mapped as they are not used by any Assembly constraint . The joints (Assembly Constraint x and y) connect two Parts instead of two Geometries. We say that these joints are of type **Part-to-Part**.

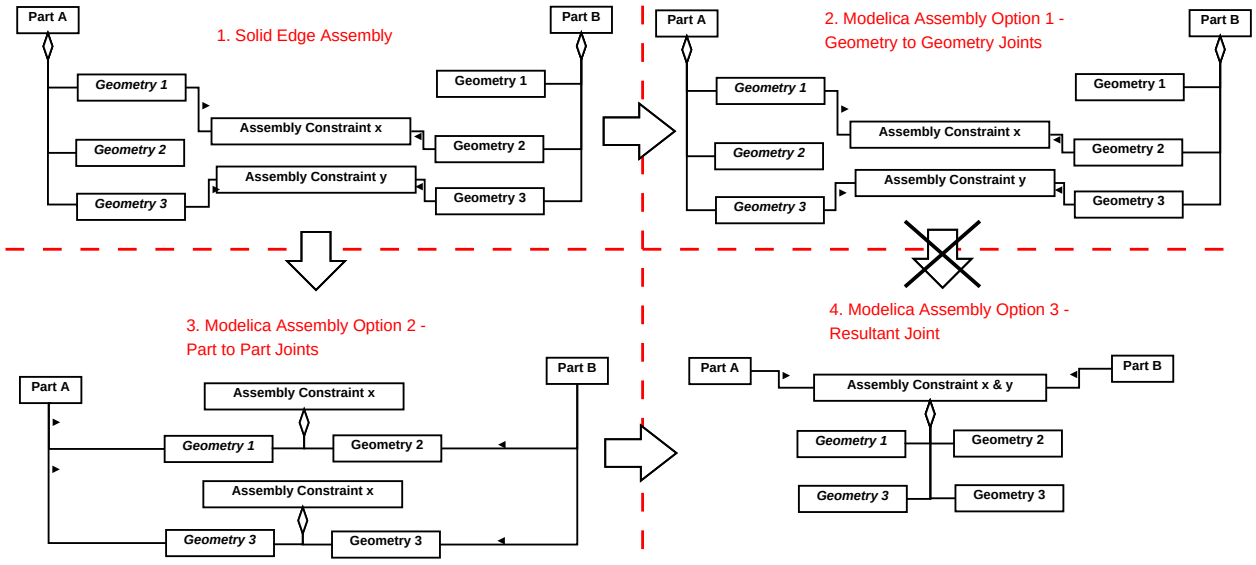



Figure 4.12: Solid Edge to Modelica Joints mapping options

#### 4.4.6.2 Mapping Resultant Constraint

Despite the advantages of the "Individual Constraint Mapping" approach, it has the drawback that the generated set of constraints may be more complicated than required for the numerical solver as well as leading to redundant constraints. Given that multiple assembly relations applied between two Parts will produce a resultant constrained dynamic behaviour and that we are concerned about the efficiency of the simulation model, then it is of interest to consider what can alleviate the equation processing and numerical simulation difficulties when generating the Modelica code.

Consider two rectangular Parts which are initially totally unconstrained relative to each other. Adding a first planar mate constraint will remove one translational and two rotational degrees of freedom. A second planar mate constraint applied perpendicular to the first removes a further translational and one rotational degrees of freedom. A final planar mate perpendicular to the previous two constraints removes the final translational constraint. Therefore the result of applying three properly oriented planar mate constraints is a fixed constraint.

In the previous section we considered mapping each assembly constraint to a corresponding Modelica joint which would therefore result in three Modelica planar mate constraints each with its own set of equations and initialisation problems (in addition to some redundant equations). However, since the resultant behaviour is that of a fixed constraint we would still get the correct behaviour if we instead produce a single fixed constraint. A fixed constraint has the benefit of being a much simpler construct, will generate a minimal set of equations and has no equation initialisation issues.

What is required is a custom constraint block which would let the user connect to two Parts of Mechanical objects and then specify a set of constraints that must be applied in parallel through a parameter driven user-interface. The logic behind the user-interface would ensure to create the "non-redundant" set of equations. Such a block exists in the MathWorks SimMechanics component "Custom Joint" which allows the user to implement what is shown in the $4^{th}$ diagram (Option 3) in figure 4.12.

Given the various types of constraints available, the multiple ways in which they can be combined and the relative orientations of the geometries they apply on, evaluating the resultant constraint is a complex process but one that is currently solved as demonstrated by any of the MCAD tools available such as Solid Edge, CATIA, Solid Works and others. We do not elaborate this approach in this thesis.

#### 4.4.6.3 Mapping Comparison

The objectives of our thesis require that we establish a bidirectional mapping between MCAD and MBS model. One option (section 4.4.6.1) was to have each MCAD Assembly Relation be associated respectively to a corresponding Modelica joint.

The other option was to map multiple MCAD Assembly Relations to a minimal set of Modelica equations. This could take the form of a generic Modelica constraint block which depending on parameter values can behave as any possible constraint by internally activating the minimal set of equations needed to reflect the resultant constraint. Such a generic block approach seems a difficult proposition to implement using the Modelica language given the various constraints we must adhere to. The complex logic required to generate the resultant equations and being restricted to what is essentially a language for doing mathematics may not be adequate for the purpose. Moreover, only very simple user interfaces can be created with Modelica. No functions can be executed during the user interaction. Modelica seems to be lacking the required functionalities for this purpose.

Another complication is determining the correct logic. As this would detract from the main purpose of this thesis and since we can accomplish the stated goals using the "Individual constraint" mapping approach, we have not explored this new avenue further as we consider it beyond the scope of the

thesis.

A final relevant consideration is that the "Resultant" approach can be considered as a continuation of the "Individual" approach and therefore to implement the "Resultant" approach we would need to solve all the problems associated with the "Individual" approach first. This is simply because the "Individual" approach is a subset of all the possibilities in the "Resultant" approach.

#### 4.4.6.4 Mapping Selection

We have elected to use and develop the "Individual Constraint Mapping" approach in this thesis because of its suitability for bidirectional mapping as well as its simplicity.

## 4.5 Combining MCAD & Multi-domain

The geometric tool/model, beside being appropriate for manipulating geometries, geometric properties and relations can serve as a valuable source of information for multi-domain dynamic models. The relations and properties defined within the model can be selectively extracted, transformed and used to drive the generation of multi-disciplinary dynamics models.

In previous sections we discussed how the parameters and properties of an individual geometric Part can be shared with other models. We didn't expand this idea to the relations that may exist **between** geometric Parts. The examples in this section show how relations between geometric Parts can be mapped or associated to relations between the components of a BD model.

A mechanical object such as that shown in figure 4.8 has many geometric features that can be uniquely and clearly identified. These include for example the different vertices, edges, surfaces, etc. If geometric features can be tagged with custom information then we can use this information to trigger the generation of, or establish an association with, components in the dynamics model.

Solid Edge provides such methods for identifying particular geometric features through textual annotations, wire-connection identifiers and other methods. In the simplest approach, the intent is to associate tagged geometric features with connector ports in the dynamics model:

$MCAD : Tagged\ Geometry \Longleftrightarrow Dynamics :\ Typed\ Connector$

A further step would be to define tagged connectors in the MCAD Parts directly without referencing their geometric features. This would allow establishing typed connections between MCAD Parts prior to any geometries being defined. These MCAD connectors can later be associated to geometric features but can have a life-cycle independent of them (i.e.. deleting the geometry will not necessarily delete the connector):

$MCAD : Connector \Longleftrightarrow Dynamics : Connector$

Therefore, with the MCAD connection points defined (tagged geometric features or geometry-less connectors), the various types of relations that MCAD tools can set between Part geometries can be used to generate a corresponding relation in the dynamics model. Solid Edge provides three mechanisms for establishing such relations between geometries: wire, pipe and mechanical assembly relations. This then gives us the following general mapping:

$MCAD : Assembly,\ Wire,\ Pipe\ Relations \Longleftrightarrow Dynamics : Custom\ relation$

This custom relation could mean a simple connection between two connector ports, something slightly more complex where a model is inserted between the two target ports or even more complex mappings if necessary.

### 4.5.1 Textual Geometry Annotations

The first geometry identification method uses MCAD annotations. Annotations are meant to add informative text to a certain drawing in order to record instructions, advice and recommendations for machining and manufacturing purposes. For example, an annotation attached to a surface might specify the machining smoothness or precision required.

We intend to use the annotation function in order to identify geometric features that will have a meaning in the dynamics model although currently there is no MCAD built-in mechanism to distinguish annotations from each other. A special annotation text format could be adopted by the user so that those meaningful for the dynamics model could be differentiated from the rest. Ideally it is best if the MCAD tool provided a mechanism to select the annotation **type** as well as allow the user to fill-in custom or prescribed parameter values.

Figure 4.13 shows an example of a Part model where two cylindrical holes are identified as **ConnectorA** and **ConnectorB**. If we were to auto-generate the corresponding Modelica model, the presence of the two tags could be used to generate corresponding connectors in the dynamics model and this would look like figure 4.14. The specific conversion will depend on the mapping rules as well as the interpretation we selected for the annotation text.

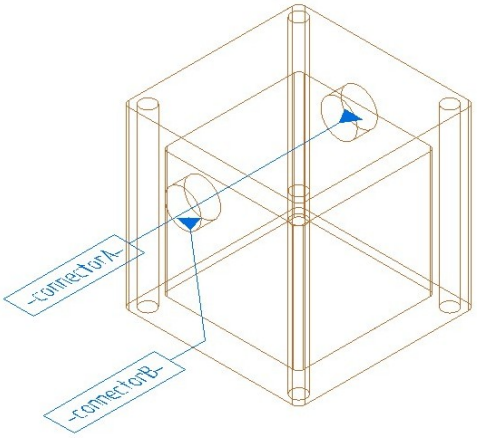



Figure 4.13: Tagged CAD model

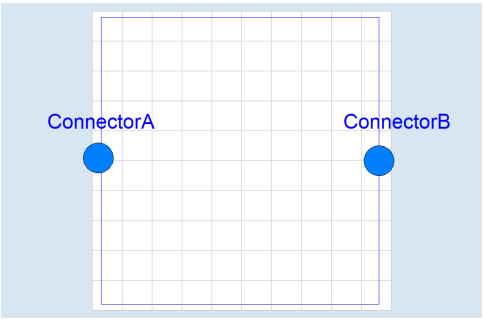



Figure 4.14: Generated Modelica Model

In other terms, assuming we start with a tagged geometric feature where the tag defines the type and name of connector, then we should have the appropriate connectors appear in the dynamics model as seen in figure 4.14. If a pipe connects to this geometric feature then we will create a connection in the dynamics model with the other end of the pipe and the pipe will then correspond to a simple connection line. If an assembly relationship connects two such tagged geometric features, again we establish a connection between the connectors in the dynamics model. We could also cut the connection line and insert a model in the middle.

### 4.5.2 Wire Harness

The second geometry identification method is a domain specific addition to the MCAD tool and is more in line with the type of capability required to meet the needs described here as it is intended to identify a certain type of geometric feature for use in connections.

Solid Edge provides the capability to specify the placement and routing of electrical wires, cables and bundles across the CAD model with the ends of the wires connected to specific geometric features on

Parts. An example without and with wires is shown in figures 4.15 and 4.16 respectively.

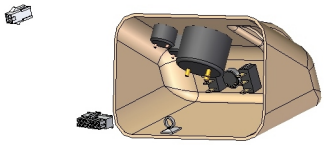


Figure 4.15: CAD assembly with no wires

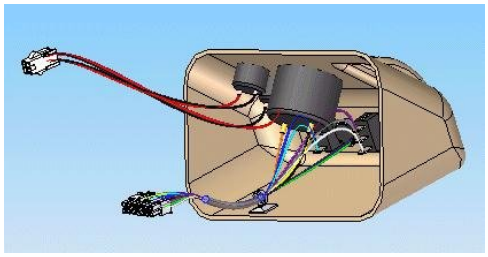


Figure 4.16: CAD assembly with wires added

Connecting wires to MCAD Parts in Solid Edge requires the presence of specific geometric features: circular shapes. These shapes do not have to be previously marked as connection terminals but could be. Figure 4.17 shows an example of a MCAD Part with two geometric features identified as wire terminals. Figure 4.18 shows the same Part as in figure 4.13 but this time the geometric features are identified as wire terminals named **ConnectorX1** and **ConnectorX2**.

The user-interface shown at the left of figure 4.17 allows the user to give a component name to the part and to create named terminals attached to **circular shape** geometric features only.

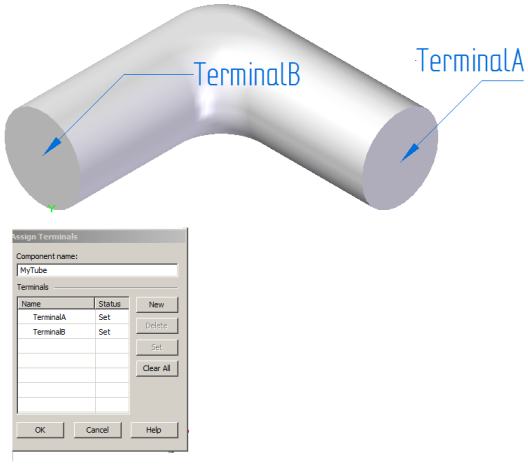



Figure 4.17: Bent-pipe with wire terminals and GUI for creating terminals

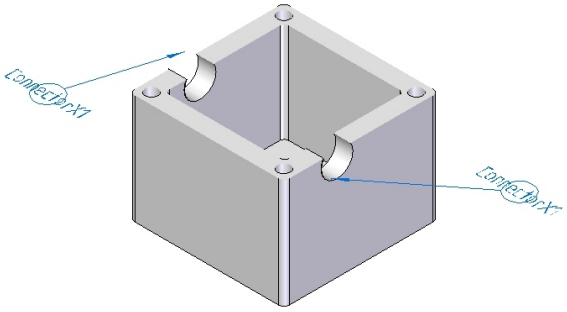



Figure 4.18: Tank with two wire terminals identified

When wires are used to establish connections between various wire terminals, the result would look something like that in figure 4.19 which shows how the tank and the tubes are connected. The equivalent dynamics model would look similar to figure 4.20 and the correspondence map is clear.

### 4.5.3 XpressRoute

XpressRoute is a mechanism for building pipes between Parts. It is the $2^{nd}$ built-in mechanism in Solid Edge for domain specific modelling of connections between components and the $3^{rd}$ approach we are describing and wish to associate to dynamic simulation models.

Unlike the previous two methods where the Part's connection points could be tagged directly in the Part model and consequently could be associated to connectors in the dynamics model, XpressRoute

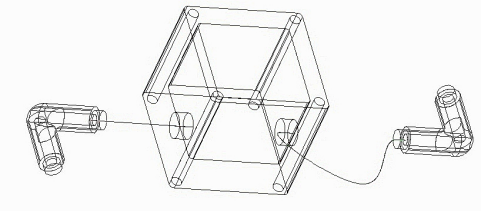


Figure 4.19: Tank and tubes connected at terminals with wires

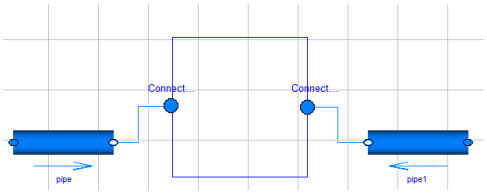



Figure 4.20: Modelica model of Tank with two pipes

doesn't provide a direct tagging mechanism. It is only after a pipe has been connected to a Part in the assembly that the creation of a connector in the dynamics model becomes necessary. This is convenient in terms of MCAD modelling as any point can be used without additional effort but inconvenient and problematic when we are trying to associate the pipe connection point to its dynamics counterpart before a pipe is inserted.

The connection points are indirectly identified when a pipe is inserted at which point the two endpoints it connects to as well as properties such as material, outer and inner pipe radii, bend radius, etc. must be specified. In many respects, XpressRoute constructed MCAD models contain most of the information required for the creation of a dynamics model for hydraulic systems. The parameters we listed in addition to the pipe length, the number of bends and a few others allow us to calculate flow resistance, pressure drop and other dynamic variables.

In a mapping from MCAD to dynamics model, the individual pipe segments can be converted to equivalent dynamic model components interconnected together and to the various tank, pump, heater models. The following two figures emphasise the similarities. Figure 4.21 shows a CAD model of a hydraulic system with two tanks, radiator, various pipes, valves and other components. Figure 4.22 shows a similar (not exactly matching) system with tanks, pipes, radiators, and other components.

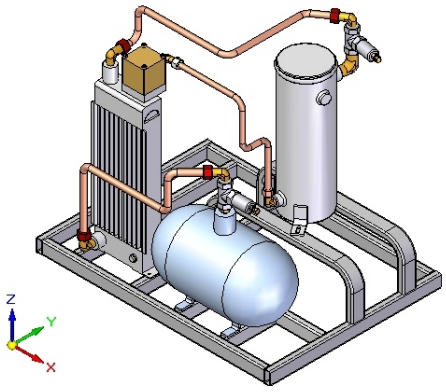



Figure 4.21: hydraulic system CAD model

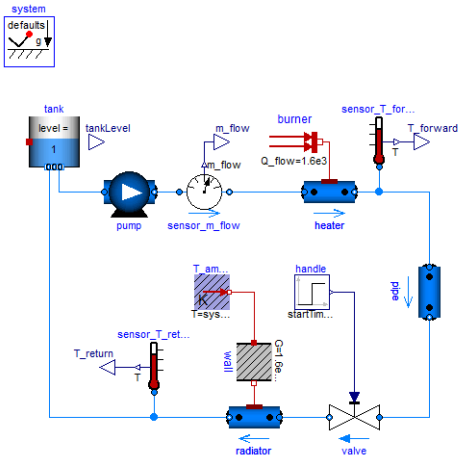



Figure 4.22: Hydraulic system with Modelica

### 4.5.4 Mechanical Assembly Relations

Mechanical **Assembly Relations** are the default Assembly construction mechanisms in Solid Edge. A single MCAD mechanical Assembly Relation applies between two geometric features on two distinct

Parts. It is primarily a mechanical constraint and a corresponding MBS dynamics model can reflect this. This is well illustrated in figure 4.3 where the MCAD model seen in the left was used to generate the equivalent MBS dynamics model seen on the right making use of the Modelica Mechanics library.

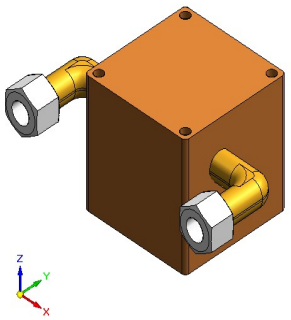



Figure 4.23: Geometric assembly of a vessel and two pipes model

As with the previous methods using Wires and XpressRoute pipes for establishing relations or connections between geometric features, Assembly relations can be considered as having established a connection or relation between geometric features which can be used in non-mechanical dynamics simulation models as well.

For example, figure 4.23 shows the geometric model of a hydraulic system composed of a constant volume tank (the box) and two bent pipes connected at its sides. A person can use this geometric representation and interpret it to generate both mechanical and hydraulic dynamics simulation models.

#### 4.5.4.1 Mechanical Interpretation

The mechanical aspect of the simulation model can be generated from the geometric model without requiring additional information: Parts are mapped to Modelica Body elements, connection points on each Part are created with the use of Coordinate transform blocks in Modelica and the assembly relations are converted to Modelica Mechanics library joints or combinations thereof. This interpretation would then produce figure 4.24 which is a model for mechanical simulation of three bodies rigidly connected together using the Modelica Mechanics library. The left and right Body objects represent the left and right connected tubes and the central body represents the vessel or tank. Since the tubes are connected at two particular points on the vessel, we have attached two frame transformation blocks to the vessel's body thus defining reference frames at the tube connection points. Similar coordinate frame transformation blocks could have also been attached to the tube models.

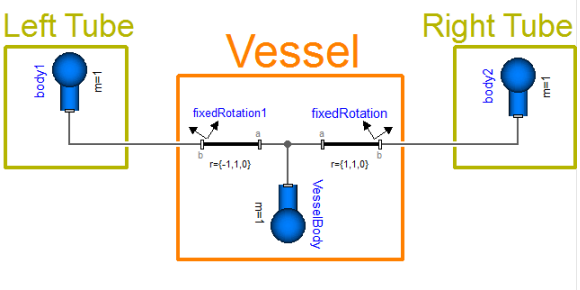



Figure 4.24: Mechanical view of vessel and two pipes model using Modelica

Assuming $GF$ is a MCAD Geometric Feature, $AsmRel$ a MCAD mechanical Assembly Relation and $MechPort$ mechanical ports in the dynamics model, we get the following MCAD to MBS dynamics mapping rule:

$GF \times AsmRel \times GF \Longleftrightarrow MechPort \times connection \times MechPort$

Note that the *connection* in the dynamics model might be replaced more generally by $connection + block + connection$ where a further block is inserted in between the two ports.

#### 4.5.4.2 Hydraulic interpretation

In order to determine whether a hydraulic connection should be made in the dynamics model, we can make use of annotation tags as explained in the previous sections. The annotation tag could identify a geometric feature on each Part as a dynamics hydraulic connection port and its name. For example, the tag text could be "DYN:HYDPORT:PORT1". The first part "DYN" identifies this as a tag to be used in the dynamics model. The second part "HYDPORT" is the type of port to create and the third part "PORT1" is the name of the port in the dynamics model.

The process would first determine where all the mechanical Assembly Relations are. For each of these connections it would verify the existence of tags on either end of a connection and if the tags are compatible or have an appropriate rule to map them to the dynamics model, a connection is established between the corresponding ports in the dynamic model.

This would then produce figure 4.25 which is a model for hydraulic simulation with a tank model in the middle and two pipes interfacing with the tank. This was developed with the Modelica hydraulics library.

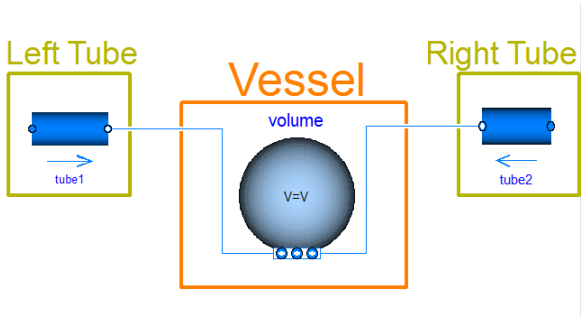



Figure 4.25: Hydraulic view of vessel and two pipes model using Modelica

Assuming $HydPort$ is a hydraulics ports in the dynamics model, we get the following MCAD to hydraulic dynamics mapping rules:

$(GF \ + \ hydraulic\ tag) \times AsmRel \times (GF \ + \ hydraulic\ tag) \Longleftrightarrow HydPort \times connection \times HydPort$

### 4.5.5 Generalising Connectors and Connections

The hydraulic interpretation can be used as a basis to generalise to other domains.

From figures 4.24 and 4.25, we can see that the mechanical connection between the vessel and the two tubes seen in figure 4.23 could be interpreted quite readily in two different ways (at least) and at the same time be combined together into a single model as shown in figure 4.26. The connections between the pipes and the tank could have as easily been made with wires or pipes and we would have interpreted all of these relations equivalently.

From the above discussion, we have the following general mapping. A MCAD model is composed of

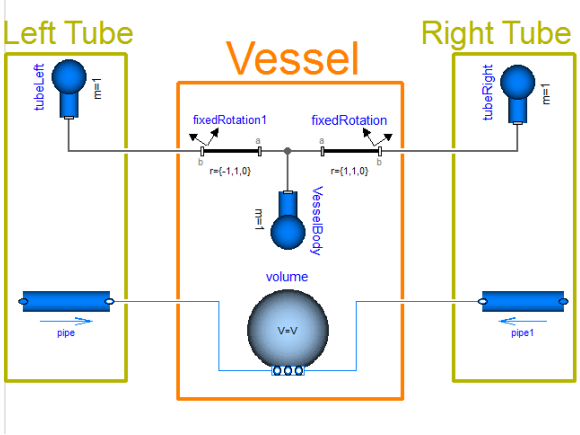



Figure 4.26: Mechano-hydraulic view of vessel and two pipes using Modelica

**Parts** containing tagged or simple geometric features, connected together with **Assembly relations**, **Wires** and **Pipes** to form **Assemblies**. the geometric features together with their relations associate to **connectors** and **Connections** in the dynamics model:

$$MCAD : [\textbf{Tagged}]\ geometry \times \{Assembly|Wire|Pipe\ relations\} \times [\textbf{Tagged}]\ geometry$$
$$\Longleftrightarrow$$
$$Dynamics : Typed\ Connector \times Connection \times Typed\ Connector$$

### 4.5.6 Combining MCAD and Modelica Tools

None of the methods presented were created by the Solid Edge software designers with the intention of being used to generate a dynamics model nor is there an easy and clear conversion to a dynamics model either:

- The geometry annotation-based technique requires further specialisation of the MCAD tool to allow us to easily select from various connector types and possibly derive this from the list of connectors available in the dynamics simulation library.
- The wire-based technique provides connectors and connections but doesn't enforce connector naming nor provides additional parameter slots.
- The XpressRoute mechanism allows the easy creation of piping systems and the pipes could adjust their dimensions to accommodate the displacement of the parts they connect to in a fashion similar to connection lines in BD models where the lines stay connected to the end-points. However, the end-points do not need to be identified neither before or after the connection is made. Therefore we cannot associate the endpoints to connectors in the dynamics model before or after the pipe connection is made unless we name the end-points using the annotations we described before.
- The Assembly Relation mechanism makes every geometric feature a potential connection point and usually multiple assembly relations are established between two Parts. Comparatively, MBS dynamics models will normally provide a few mechanical interface points and usually as fixed reference frames.

The purpose of this analysis was to indicate that current MCAD tools provide many mechanisms that could be improved upon and allow us to achieve a multi-domain and robust integration with

dynamic models. The MCAD tools already provide most of what is needed to define electrical wiring, hydraulic piping systems and therefore extract corresponding dynamics models with the connectivity information. But the capability to define connector types, attach them to Parts or geometries and connect them together as is routine in in BD tools is not available.

MCAD and Dynamics models can be integrated potentially into **a single tool**. It would not be difficult to augment the MCAD tools (and subsequently their models) to allow us to define generalised connector types, connector attach points and connections. Individual MCAD Parts would be associated to individual Modelica models with the Modelica connectors visible and connectible within the MCAD environment.

## 4.6 MCAD to Modelica Mapping

In order to establish a bidirectional association between Modelica and Solid Edge, we first need to understand the relevant components on either side. Both environments provide the means to construct multi-body systems using mechanical parts and joints. However, the joints in either environment do not map entirely to joints in the other. Therefore we need to determine the optimal approach in building this bidirectional association.

On the Modelica side we have the Modelica.Mechanics library that provides a starting point for creating mechanical systems connected with a variety of joints. These joints were constructed using the Modelica language and can be modified or adapted further if required. We will refer to this library as Modelica.Mechanics or the MBS library.

On the Solid Edge side we are given a tool where the definition and behaviour of its geometric elements (parts and assembly relations) cannot be changed although we have the capacity to create new assemblies and use them. We will work under the assumption that Solid Edge is of limited adaptability for our purposes and whenever possible focus on adapting the MBS library to meet the constraints imposed by SE.

Our objective for achieving a bidirectional relation between an MCAD model of a mechanical assembly and an MBS model in Modelica requires that we work out the conversion rules from a MCAD model to a Modelica MBS model. We will later develop the Modelica to MCAD mapping. In order to determine what Modelica models to develop, we present a comparison of the relevant elements of both. We restrict our analysis to mechanical systems and therefore refer to the existing Modelica.Mechanics library as a starting point.

The details of Solid Edge and Modelica.Mechanics were provided in the literature review section. Here we provide a quick overview of the relevant elements for further analysis and comparison.

### 4.6.1 Solid Edge Elements

The Solid Edge tool has the following modelling elements.

#### 4.6.1.1 Part Model

An example of a MCAD Part is shown in figure 4.27 and has the following features:

- Part: rigid object with a default **coordinate frame**, **mass properties** and a **geometry**.
- Coordinate frame: the principal coordinate frame exists by default when a Part is created and can be seen at the centre of the figure. It serves as the reference relative to which all other geometric objects are placed and with respect to which the mass properties are calculated.
- Secondary coordinate frames: secondary coordinate frames can be defined relative to geometric

features or other coordinate frames.

- Mass properties: the Mass properties of a Part are fixed and describe the mechanical properties of the rigid body. They include the mass, inertia and centre of gravity. The mechanical properties can be either automatically calculated using the Part's geometry and density information or be arbitrarily set by the user.
- Geometry: the geometry is the shape of the Part. It is generated as the result of various simple geometric operations. These geometric operations will then define entities such as points, lines, planes, cylinders and many other basic geometric entities which when patched together give us the overall geometry.

#### 4.6.1.2 Assembly Relation

An Assembly relation is a mechanical constraint between geometries of two distinct Parts. Listed are some of its properties:

- Assembly relations come in different types
- Assembly relations connect two Parts by referring to a single geometric feature on each
- Multiple assembly relations can be set between the same two MCAD Parts
- Each assembly relation type is restricted to apply to a particular combination of geometric types

#### 4.6.1.3 Assembly Model

An Assembly is a composition of multiple parts, assembly relations and even other assemblies which are then called sub-assemblies.

### 4.6.2 Modelica Mechanics library

A Modelica MBS model will have the following elements:

#### 4.6.2.1 Body model

A Modelica.Mechanics Body is seen in figure 4.28 has the following mechanical features:

- Body: models the mechanical dynamics of a rigid body
- Coordinate frame: represented by the connector in the left of the figure
- Mass properties: centre of gravity coordinates, mass and inertia matrix

#### 4.6.2.2 Fixed Constraints

Fixed constraints provide either an absolute or relative coordinate transformation mechanism. There are three such elements in the Modelica.Mechanics library.

- Fixed (*Modelica.Mechanics.Parts.Fixed*): is a model that provides a fixed reference frame with an arbitrary position and orientation. A Body connected to a Fixed model would not move under the influence of forces or torques.
- FixedTranslation (*Modelica.Mechanics.Parts.FixedTranslation*): is a model that applies a rigid translation constraint between two Body models which then must maintain a fixed translation between their respective reference frames. These reference frames will be aligned with each other.
- FixedRotation (*Modelica.Mechanics.Parts.FixedRotation*): is similar to FixedTranslation except that the two reference frames do not need to be aligned as we can introduce an arbitrary rotation in addition to the arbitrary translation.

#### 4.6.2.3 Joints

Joints are a category of models that restrict the mechanical degrees of freedom between their two reference frames. These reference frames must be connected to other Body models, Fixed constraints or even possibly other joints.

#### 4.6.2.4 Composite model

The composite model is not part of the Modelica.Mechanics library but a feature of the Modelica language. A model is **composite** if it contains other models. We list it here to later establish a parallel with the MCAD Assembly models. As an example, a model containing a Body and some Joints would be a composite model.

### 4.6.3 Mapping MCAD Part to Modelica

The basic counterpart of a MCAD Part model is the Modelica.Mechanics Body model. Representative models of each are shown in figures 4.27 and 4.28 respectively. Its mechanical parameters include the mass, inertia matrix and a centre of mass defined relative to its principal coordinate frame represented by the connector at the left.

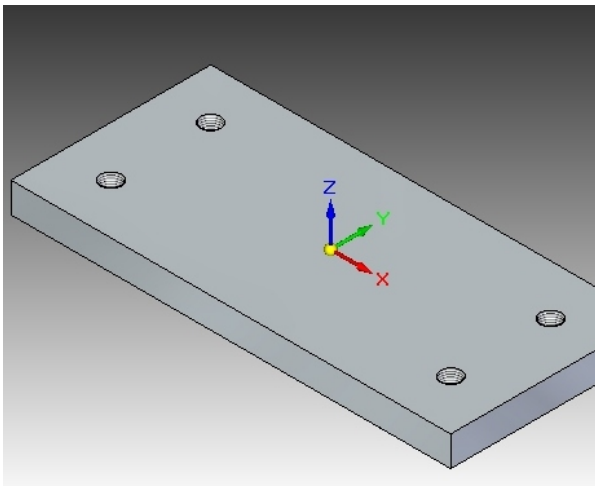



Figure 4.27: MCAD Part

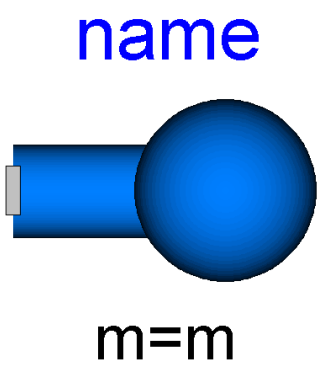



Figure 4.28: Modelica.Mechanics Body

The following features are common to both MCAD Part and Modelica.Mechanics Body:

- Reference Frame
- Mass properties

Differences exist between MCAD Part and Modelica.Mechanics Body. The first difference is in the geometric features:

- the MCAD geometric features, besides being used for visualisation purposes, can be used to define an assembly constraint with the geometric features of another Part as well as be used to detect collision between geometries in motion.
- Modelica Body does not have geometry to achieve the same function as the geometry does in MCAD modelling.

The second difference is a minor one and has to do with secondary coordinate frames we can add in the MCAD model. The default Modelica Body has only one coordinate frame. Therefore if we add coordinate frames in the MCAD model we must add corresponding coordinate frames in the Modelica model as well.

Therefore, a newly created MCAD Part with no Geometry is equivalent to a Modelica Mechanics Body. Both have reference coordinate frames and mass properties which gives the following correspondence relation: **Modelica Body $\leftrightarrow$ Empty MCAD Part**

When geometry or extra coordinate frames are added to the MCAD model, this equivalence breaks. Equivalents for the MCAD coordinate frames exist in Modelica but not form geometries. To achieve a bidirectional (or more precisely an isomorphic) association between MCAD elements and their Modelica counterparts, we can create the following:

- Modelica Geometric elements: corresponding to MCAD geometric elements
- Modelica:Part: a composite Modelica model referred as Modelica:Part which contains a single Modelica:Body model, Modelica geometric elements and secondary Coordinates frames

The Modelica:Part then becomes the equivalent of the MCAD:Part model.

### 4.6.4 Mapping MCAD Geometry to Modelica

We have seen the basic association that exists between a Modelica Body and a MCAD Part. We have also seen that the correspondence is complete when the Part has no associated geometry (when it has just been created). This correspondence is lost once geometry is added, modified or new coordinate frames are added. We already know that we can map coordinate frames to the Modelica.Mechanics fixed constraints and therefore what remains to be determined is how to map the geometric information into Modelica.

#### 4.6.4.1 Geometric Features

The Solid Edge geometric features which can be used in assembly relations are the following:

- Keypoint: all vertices in a geometry are keypoints. In addition, geometrically significant points are considered keypoints and include the centre of a circle or an arc and the middle of a straight line among others.
- Linear element: these are all the straight lines in a geometry. For the purposes of the assembly relations, the linear elements are considered infinite in size.
- Planar surface: these are all the planar surfaces in a geometry. For the purposes of the assembly relations, the Planar surfaces are considered infinite in size.
- Cylinder face: these are the surfaces of cylindrical objects. For the purposes of the assembly relations, the Cylinder face is considered to be that of a full cylinder ($360^o$) and of infinite axial size.
- Cylinder axis: this is the axis of a Cylindrical geometry. For the purposes of the assembly relations, the cylinder axis is considered infinite in length and is equivalent to a linear element.
- Coordinate frame: Parts can have multiple Coordinate frames defined relative to each other or relative to geometries.

We require an efficient method to represent and use MCAD geometric features in Modelica such that they can be used to establish mechanical constraints as in Solid Edge. Fortunately, all relevant geomet-

| Solid Edge | Modelica | Comments |
|---|---|---|
| Reference frame | Reference Frame | Exact correspondence |
| Point/Keypoint | Reference Frame | Reference Frame origin is same as Point coordinates |
| Line | Reference Frame | Reference frame's origin is on the line, z-axis correspond to line axis |
| Plane | Reference Frame | Reference frame's origin is on the Plane, z-axis corresponds to plane normal |
| Cylinder Axis | Reference Frame | Reference Frame's origin is on Cylinder Axis, z-axis corresponds to Cylinder Axis |
| Cylinder Surface | Reference Frame<br>+ Radius | Reference Frame's origin is on Cylinder Axis, z-axis corresponds to Cylinder Axis<br>Radius defines surface relative to z-axis |

Table 4.1: General representation of Geometric features

ric features (those used with Solid Edge Assembly relations) have a simple mathematical representation which is described in table 4.1:

The Solid Edge *Reference Frame* is represented by the Modelica.Mechanics *Frame* connector such that the two share the same origin and axes orientations. The Solid Edge *Keypoint* can also be represented by a Modelica.Mechanics *Frame* but the orientations of the axes would be inconsequential. The Solid Edge *Line* can be defined by two elements: a point on the line and the line direction. We can therefore use a Modelica.Mechanics *Frame* with its origin somewhere on the line and its z-axis parallel to the line. The same idea applies to the Solid Edge *Plane* where we pick a point on the plane as the origin of the Modelica.Mechanics *Frame* and its z-axis oriented normal to the Plane. The Solid Edge *Cylinder Axis* is mapped the same way as the Solid Edge line.

The Solid Edge *Cylinder Surface* remains the only exception in the way it is represented in Modelica. All other elements were represented by a single Frame. As with the previous geometries, a Modelica.Mechanics *Frame* is required but is not enough. To define an infinitely long Cylinder Surface we need to represent its axis and radius. The *Cylinder Surface* is then represented with a *Frame* whose origin is on the cylinder axis, its z-axis collinear with the axis and an additional Radius parameter.

#### 4.6.4.2 Connectors - Attach Points

Modelica.Mechanics models use Reference *Frames* as attach points in mechanical systems. The corresponding element in MCAD domain is the Coordinate frame which can be positioned anywhere relative to the Part's reference coordinate frame, relative to other coordinate frames or relative to geometries in the Part.

The same cannot be said for MCAD Geometric features as there are no corresponding elements in Modelica.Mechanics. This then requires that we create additional models to complement the Modelica.Mechanics library and corresponding to these geometric interface points or "geometric connectors".

**Connectivity Differences** We examine how Modelica.Mechanics Reference *Frame* connectors differ from the geometric connectors in Solid Edge.

A Modelica.Mechanics model connected to a *Frame* has the references to fix any of its three translational and three rotational degrees of freedom. Connections are made between *Frames* as seen in

figure 4.24. Whereas a MCAD Part connected to a Plane geometry via some assembly relation cannot possibly be constrained not to move parallel to the plane. Similarly an assembly relation between a Part and a point geometry cannot enforce a rotational constraint as there are no angles we can calculate relative to a point.

Another difference is in the connectivity restrictions. The Modelica.Mechanics *Frame* connector can be connected with any other *Frame* connector. The components in the Modelica.Mechanics library have a single type mechanical interface: the *Frame* type.

In Solid Edge, Assembly relations have multiple connector types. A particular Assembly relation can be applied only between particular combinations of geometries. For example a *Connect* Assembly Relation will apply between a *Point* and one of *Point*, *Line*, or *Plane* geometries. It cannot connect a *Line* to another *Line*. To enable the same connectivity restrictions in Modelica, we cannot simply use the Modelica Frame connectors. We require "typed" Frame connectors.

**Modelica Implementation** We start by defining a new Modelica type for the geometries:

- type typeKeypoint = Real; // Point type
- type typeLinearElement = Real; // Line type
- type typePlanarFace = Real; // Plane type
- type typeCylindricalAxis = Real; // Cylinder Axis type
- type typeCylindricalFace = Real; // Cylinder Surface type

We did not create a new type for the Solid Edge Coordinate System connector. This is because we will not differentiate it from the standard Modelica.Mechanics Frame connector as the Frame connector represents perfectly the Solid Edge Coordinate System connector.

For all other geometries, the next step is to include the newly defined typed variables in the Geometry Frame connectors. We would then have defined typed Frame connectors which can be only be connected to connectors of the same type. This will permit us to define Modelica equivalents of Solid Edge Assembly Relations.

In the case of the Cylinder geometry, the Surface Connector requires special treatment. The value of the cylinder radius needs to be shared with objects that connect to the surface. As an example, the "Tangent" Assembly Relation needs to know the distance from the Cylinder Axis to model the tangency relation appropriately. To implement this behaviour, there are multiple approaches and we consider some of them here:

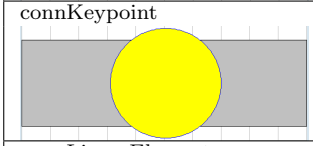
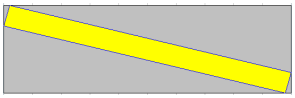
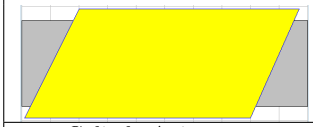
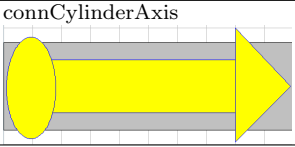
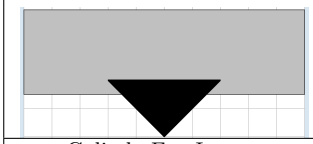
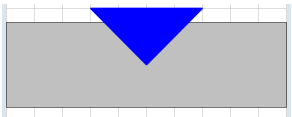
1. The Modelica Assembly Relation connecting to the Cylinder Surface connector has a parameter value for the Cylinder radius
2. The Modelica Assembly Relation connecting to the Cylinder Surface connector has a parameter reference to the cylinder geometry. This reference indirectly provides the cylinder radius
3. The Cylinder Surface Connector outputs the Cylinder radius through the connector

Option 1 is an error prone solution as it requires the user to enter the correct radius value manually and will not be updated as the geometry evolves (unless we implement a special process to compensate).

Option 2 is more robust as long as we have provided the correct geometry reference and we are updating the geometry data in a single location when the corresponding MCAD geometry is modified. However, getting to the correct Geometry reference in Modelica will not be simple as we do not see the Geometry

visually to select it (as we can in Solid Edge) and the Geometry References we mentioned are long and obscure identifiers which are hard to differentiate from each other for a human.

Table 4.2: Modelica Geometric Connectors

| *Model* | *Description* |
|---|---|
| connCoordinateSystem  | Coordinate System Connector<br>connector connCoordinateSystem "Coordinate System Connector"<br>extends MultiBody.Interfaces.Frame_a;<br>end connCoordinateSystem; |
| connKeypoint  | Single point Geometry connector<br>connector connKeypoint "Keypoint Connector"<br>MultiBody.Interfaces.Frame_a frame_a;<br>constant typeKeypoint keypoint= 0;<br>end connKeypoint; |
| connLinearElement  | Straight Line Geometry Connector<br>connector connLinearElement "Linear Element Connector"<br>MultiBody.Interfaces.Frame_a frame_a;<br>constant typeLinearElement connType= 0;<br>end connLinearElement; |
| connPlanarFace  | Planar Surface Geometry Connector<br>connector connPlanarFace "Planar Face Connector"<br>MultiBody.Interfaces.Frame_a frame_a;<br>constant typePlanarFace planarFace= 0;<br>end connPlanarFace; |
| connCylinderAxis  | Cylinder Axis Geometric Connector<br>connector connCylinderAxis "Cylindrical Axis Connector"<br>MultiBody.Interfaces.Frame_a frame_a;<br>constant typeCylindricalAxis cylindricalAxis= 0;<br>end connCylinderAxis; |
| connCylinderFaceOut  | Cylinder Surface Geometric Output connector.<br>This connector would be part of the Geometry<br>connector connCylinderFaceOut "Cylindrical Face Connector - Output"<br>MultiBody.Interfaces.Frame_a frame_a;<br>Modelica.Blocks.Interfaces.RealOutput radius "Radius Output";<br>constant typeCylindricalFace cylindricalFace= 0;<br>end connCylinderFaceOut; |
| connCylinderFaceIn  | Cylinder Surface Geometric Input connector. This connector would be part of a model connecting to the Geometry<br>connector connCylinderFaceIn "Cylindrical Face Connector - Input"<br>MultiBody.Interfaces.Frame_a frame_a;<br>Modelica.Blocks.Interfaces.RealInput radius "Radius Input";<br>constant typeCylindricalFace cylindricalFace= 0;<br>end connCylinderFaceIn; |

Option 3 is simple and robust against changes to the cylinder geometry. The cylinder geometry component in Modelica will provide its radius to the Cylinder Surface connector radius output and any components which use this value will get updated automatically. The user does not need, as in the previous cases, to manually adjust any parameters to fit. Moreover, the radius value is attached to its proper parent (i.e. the Geometry) from which the information is accessible to models that need to use it.

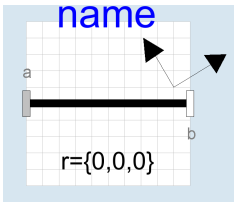
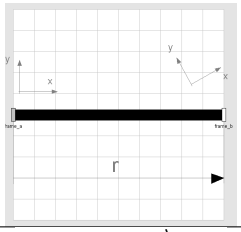
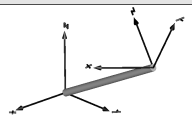
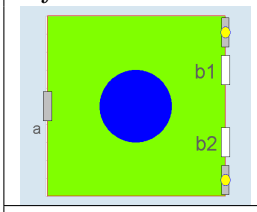
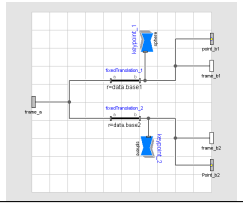
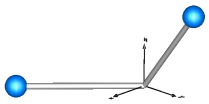
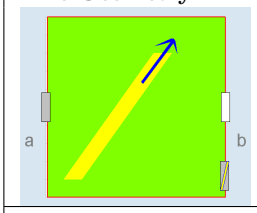
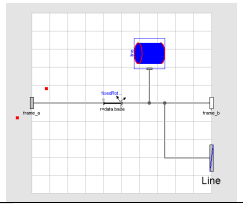
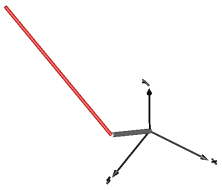
For robustness and simplicity reasons, we have chosen this last option 3 using a Modelica "variable" to pass the radius value. We will need to create two such connectors: the output as described and the corresponding input connector.

Table 4.2 provides a list of all the Geometric connector and their Modelica concrete syntax. Notice the simple Coordinate System connector and the two connectors for the cylinder surface.

#### 4.6.4.3 Combining Geometry and Connectors

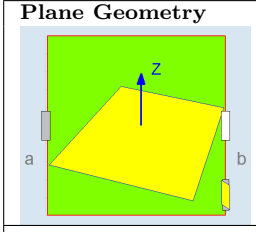
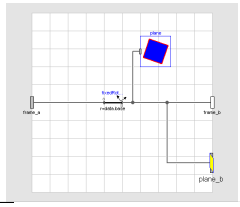
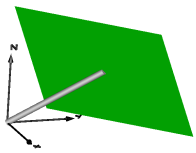
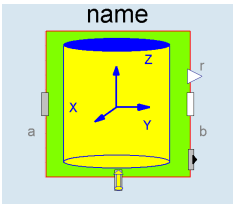
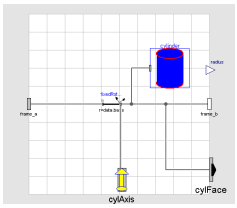
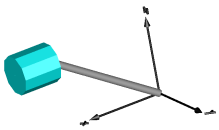
We have provided a method to represent the various geometric entities and we have also defined new connectors which if properly positioned and interpreted will allow us to mimic the characteristics of the Solid Edge geometric elements. Table 4.3 provides a description of the Modelica equivalents to the Solid Edge geometries.

Table 4.3: Modelica Geometries

| *Model Icon* | *Model Diagram and 3D* | *Description* |
|---|---|---|
| **geometry Coordinate System**   |  | The Coordinate Frame geometry is simply the MultiBody.Parts.FixedRotation model which can position a Reference Frame at a fixed Offset and Rotation from the base frame. |
| 3D representation |  | The right coordinate system is positioned and oriented relative to the base coordinate system on the left. A connector is provided (in the diagram) to connect to the right coordinate frame. |
| **Two Keypoint Geometry**   |  | The Keypoint geometry contains the connections for two Keypoints as Keypoints always come in Pairs in Solid Edge. It is composed of two MultiBody.Parts.FixedRotation models with associated visualisation models and the Keypoint connectors we have defined earlier. |
| 3D representation |  | The coordinate system represent the base point. Two keypoints are represented as blue spheres. A vector connects the base to the keypoints. Connectors are provided (in the diagram) to connect to either of the points. |
| **Line Geometry**   |   | The Line geometry is composed of a single MultiBody.Parts.FixedRotation that positions the line in space such that the z-axis of the end-frame is aligned with the Line. Visualisation models are also part of the model. |
| 3D representation |  | The red cylinder represents the line. The grey vector from the base reference provides a point on the line. Connectors are provided (in the diagram) to connect to the line. |

*continued on next page*

*continued from previous page*

| *Model Icon* | *Model Diagram and 3D* | *Description* |
|---|---|---|
| **Plane Geometry**   |  | The Plane geometry is composed of a single MultiBody.Parts.FixedRotation that positions the Plane in space such that the z-axis of the end-frame is aligned with the normal of the Plane and this end-frame's origin is on the Plane surface. Visualisation models are also part of the model. |
| 3D representation |  | The green surface is the plane geometry positioned and oriented relative to the base plane. Connectors are provided (in the diagram) to connect to its surface. |
| **Cylinder Geometry**   |   | The Cylinder geometry contains two connectors: one axis connector and one surface connector. A single Modelica.MultiBody.Parts.FixedRotation places a reference frame at the base of the cylinder with its z-axis aligned with the cylinder axis. The two connectors are then positioned relative to this frame. |
| 3D representation |  | The cylinder is positioned and oriented relative to the base frame. Connectors are provided (in the diagram) to connect to its axis or surface. |

### 4.6.5 Mapping Solid Edge Assembly Relations And Joints

We provide a detailed comparison of the Assembly relations available in Solid Edge versus the joints provided by default by the Modelica.Mechanics library in a further section. The Modelica.Mechanics library does not provide an exact equivalent constraints for all the Solid Edge constraints and therefore new constraint models will have to be created to supplement the library.

The following is the list of the Solid Edge Assembly relations available with a short description:

- Match Coordinate: fixes the relative position of the coordinate frames of two parts. Is a combination of three Planar Align relations applied between the corresponding three faces of the two coordinate frames.
- Axial Align: sets two axes or straight lines to be collinear. The rotation angle about the line/axis can be set to either a fixed value or remain floating. The direction of the axes/lines can be set to be the same or opposite.
- Mate: two planar surface are parallel and face to face (their normals are in opposite directions). Distance between the faces is either fixed or floating (unconstrained).
- Planar Align: same as Mate but the face normals are in the same direction.
- Connect (Keypoint - xx): sets a fixed distance constraint between a point and either one of a point, a line or a planar surface (xx stands for point, line or plane).
- Tangent: a Cylindrical surface is tangent to either a planar or cylindrical surface and a tangent distance is provided.

- Angle: specifies a fixed angle constraint between either two lines or two planar surfaces.
- Gear: Specifies the rotation angle ratio between selected axes of two parts.
- Cam: Specifies a constraint between the normals of a planar surface and a closed line in a plane. The planar surface must keep a fixed distance to the line.
- Insert: a combination of Mate and Axial Align relations

#### 4.6.5.1 Assembly Relations Connectivity Behaviour

**Polymorphic behaviour in Solid Edge** In Solid Edge, some Assembly Relations exhibit a polymorphic behaviour.

**Variant A** For example the "Connect" relation comes in three types in terms of the geometries it can connect to as well as the mechanical constraint it will generate. These types are:

1. Point to Point: the distance between the two points is fixed
2. Point to Line: the shortest distance between the point and the line is fixed
3. Point to Plane: the shortest distance between the point and the plane is fixed

The user cannot select a specific type at the start of the process. This is established as the connections are being made. If the first connection is to a line, then the system will pick the $2^{nd}$ type and automatically supply the associated user-interface. If instead the first connection is to a point, then all three types are still available and the user-interface will adapt based on the next geometry selected.

In resumé, the Solid Edge user-interface allowed the user to start with a group of Assembly Relations and through parameter choices or selections the user-interaction reduced it to a particular element in the group. This is an example of a polymorphic behaviour. The three types have a different mathematical representation.

**Variant B** Another variant of a polymorphic Assembly relation is the "Axial Align" relation which allows any of the following combinations of geometry connections: {Cylinder Axis, Linear element} $\times$ {Cylinder Axis, Linear element}. In this case, the mechanical behaviour is the same for all the combinations and therefore the mathematical model of the constraint is the same.

**Polymorphic Behaviour In Modelica** Duplicating the polymorphic nature of these models in Modelica (with Dymola) is problematic as polymorphism is not supported as of Modelica 3.1. An alternative to polymorphism is to have more flexible user-interfaces that can constrain the types of connections available based on the first connection made or parameter inputs. Modelica does not support the connection to affect user choices, therefore we are left with user-parameters to manipulate the available choices.

We will use model parameters to select the connectors and the specific mechanical constraint or Assembly Relation model that are to be activated. We implement this in two steps:

- create the specific Assembly relation required
- create a container model that, based on parameter selections, will activate specific assembly relation and connectors

We provide the details of this implementation as we discuss the individual Assembly Relations.

#### 4.6.5.2 Unimplemented Assembly Relations

Some of the Solid Edge Assembly relations were not converted to Modelica and will be implemented in future work. However, we will provide some details on what is required to implement the Modelica models.

**Gear** There are three types of gears in Solid Edge: Rotation-Rotation, Rotation-Linear, Linear-Linear. In all three cases, the Solid Edge constraints that limit the motion must be already set and the gear relation merely specify how the motion of both parties is dependent on the other. The relation therefore specifies the dependency between a rotation and a rotation, between a rotation and a linear motion and finally between two linear motions.

**Rotation-Rotation Gear** In order to apply a Rotation-Rotation gear relation in Solid Edge, the two Parts involved must already have an AxialRelation3d applied on them. Therefore, the equivalent Modelica model corresponds to the Parts with a Revolute joint on each. When a Rotation-Rotation gear relation is applied, the corresponding change in Modelica is to add a gearRatio model to drive the two Revolute joints through their actuation ports. The gearRatio Modelica model can apply a ratio as well as a rotation directionality similar to SE.

**Rotation-Linear Gear** In order to apply a Rotation-Linear gear relation in Solid Edge, the two Parts involved must already have an AxialRelation3d and either another AxialRelation3d or an equivalently constrained translation motion in 1-dimension. The relationship establishes a ratio between the rotation angle and the displacement value. The equivalent Modelica model is composed of two Parts one with an actuated Revolute joint and the other with either an actuated prismatic or an actuated cylindrical joint. The addition of the gear would then correspond to the addition of a (Modelica.Mechanics.Rotational.Components.) IdealGearR2T which establish a ratio between rotation and translation.

**Linear-Linear Gear** In order to apply a Linear-Linear Gear relation in Solid Edge, the two Parts must be constrained to have only one translation degree of freedom. The corresponding Modelica model would be composed of Parts and either a Prismatic or Cylindrical joints. The addition of the Gear would then correspond to a new block with a behaviour similar to IdealGearR2T which instead specifies the ratios between two translations.

**Cam** Not Implemented. The CAM Assembly relation can not be constructed from existing Modelica.Mechanics joints and would require the development of new models.

**Insert** The Insert Assembly relation in Solid Edge in fact will insert two more basic Assembly relations: a Planar Mate and an Axial Align. These would correspond to a single Modelica.Mechanics Revolute joint. Two geometries on each Solid Edge Part are required to place the two assembly relations. These are a Plane and an axis normal to the Plane. Therefore, from the Modelica perspective we need to combine these two geometries into a single connection point as trying to apply two constraints in Modelica between the same two Body models would lead to an over-constrained set of equations. The combination of a plane and a normal axis can be replaced by a single Modelica Frame (or Coordinate system) such that its origin is at the intersection point, its z axis collinear with the normal axis and consequently the x,y axes would be in the plane.

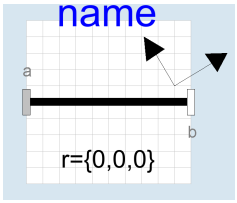
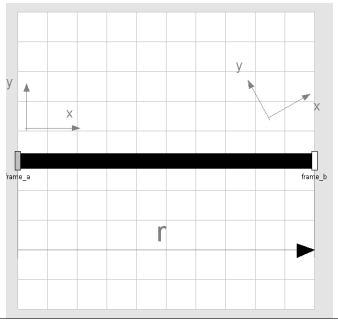
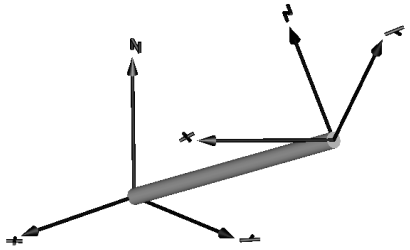
#### 4.6.5.3 Assembly Relations: Geometry to Geometry version

The **Geometry-to-Geometry** Assembly relations described in table 4.4 were created to allow the user to establish constraints between geometric features directly in Modelica. Although the presence of the

individual geometric features included in each Part permitted us to visualise them in the Dymola 3D display, determine their names and allow us to establish a connection by using the Nested Connector interface, it is clear from experiments that having so much geometric information for each Part is heavy on the processor and memory. It is also clear that referring to geometry directly in Modelica or Dymola is not practical as we have a multi-step process to go through: compile and visualise the model, find the geometric feature and remember its (cryptic) name, return to the modelling environment where we search through the extensive list of geometric connectors to find this cryptic name and then establish a connection.

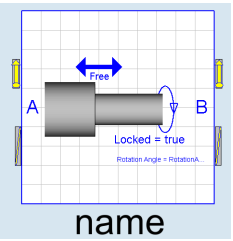
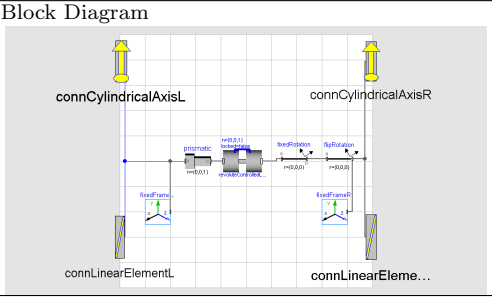
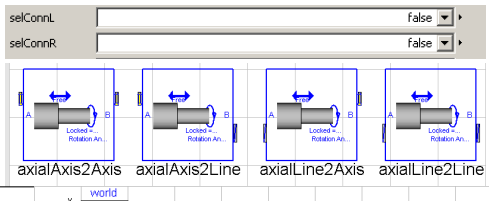
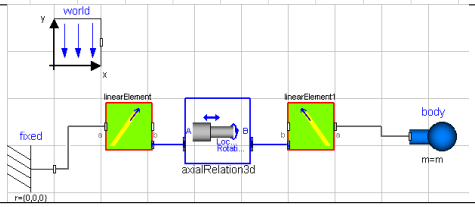
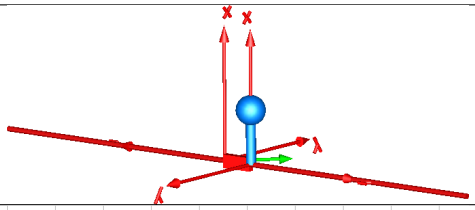
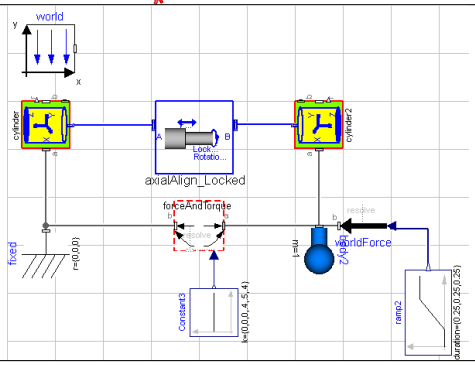
Current Modelica tools do not have the proper user-interface for this kind of interaction. A suggestion is to either integrate Modelica into a MCAD tool such as Solid Edge or to integrate some level of 3D support into the Modelica tool itself. In the latter approach, the user would be able to switch to a 3D view of the model where those elements which have 3D information will be visible. Each 3D object would have a variety of geometric connection points which can include all the geometric features as described and additional geometric interface points. These geometric connection points can be contained in layers which can be activated or deactivated to manage the clutter. One layer may include for example all hydraulic connection ports.

Table 4.4: Modelica Geometry-Geometry Assembly Relations

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Match Coordinates**   | Block Diagram   | The Match Coordinates assembly relation in Solid Edge completely constrains the relative translational degrees of freedom between two Coordinate systems. The user can change the translation parameters but no rotation can be inserted. The Modelica components we have developed implement this in addition to being able to adjust the rotation. |
| **Example**: Diagram of two Frames with a Match Coordinate relation |   | A translation and a rotation is introduced between the two reference frames. |

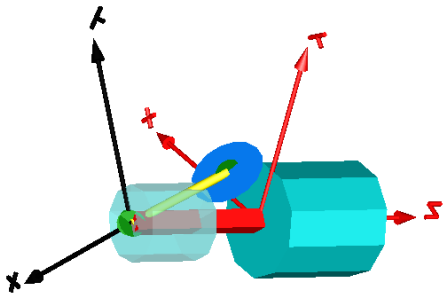
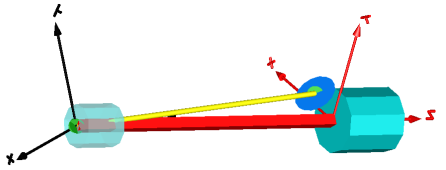
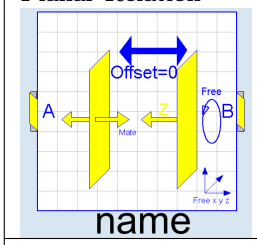
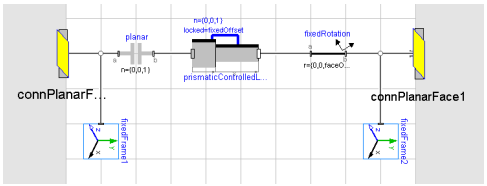
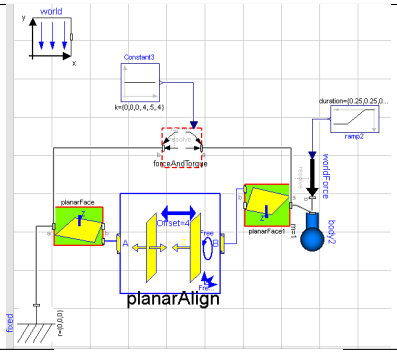
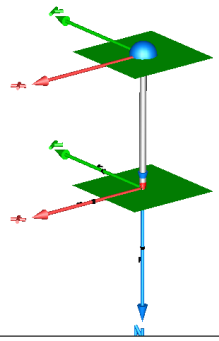
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Axial Relation**<br>  | Block Diagram<br>  | The Axial Relation block in Modelica establishes an Cylindrical constraint between to linear elements. Linear elements may be either a Cylinder axis or a straight Line. The two linear elements may be free to rotate or not in which case a relative angle must be specified. |
| Connection Parameters and<br><br>Resultant Blocks |  | The user interface of the Axial Relation allows the user to select what geometries it will connect to on either side.<br><br>This generates four possible combinations (or three if we remove one of Line2Axis or Axis2Line variants). |
| **Example 1**: Diagram of an Axial Relation linking two linear elements |  | |
| Initial Position |  | In the initial position, we see the body element (blue sphere) to the right of the figure. the xy coordinate axes on the left and right represent the linear elements with the z axes pointing along the line elements. The square red rod connecting the two represents the axial relation. In this case, the axial relation is rotationally locked which is the reason why it is represented by a prismatic rod. |
| After displacement |  | Under the influence of the force of gravity (green arrow at the right), the body element starts falling. We see that the motion is along the axial relation. The two linear elements now appear as red cylinders (superposed with the red z-axes). |
| **Example 2**: Diagram of an axial relation linking the axes of two cylindrical shapes. Various forces and torques are applied to test the constraint behaviour. |  | |

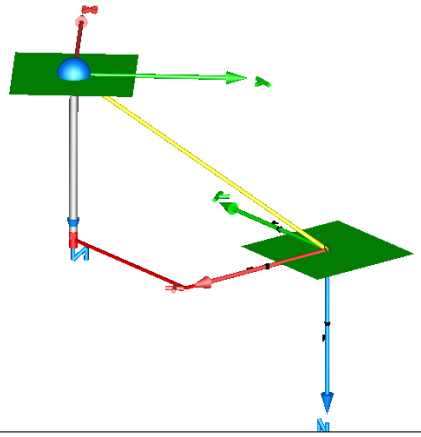
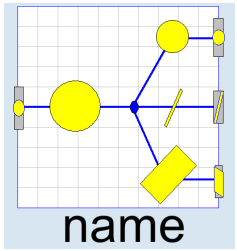
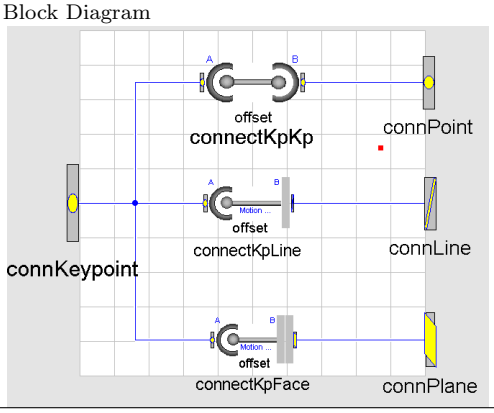
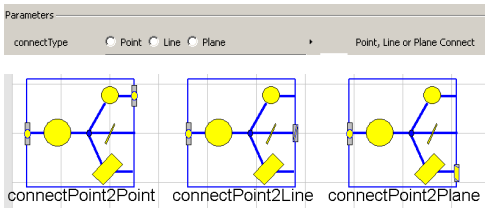
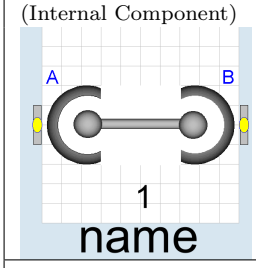
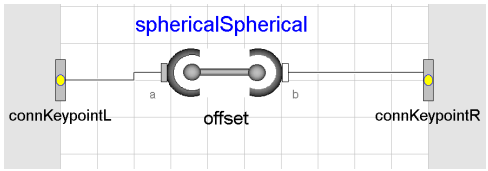
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Initial Position |  | The cylinders on the left and right represent the two cylindrical elements. The red prismatic connecting the two is the rotationally locked axial relation. The yellow rod links the base points of the cylinders and represents an applied torque which is trying to rotate the two cylinders along their common axis. A force is also applied on the free cylinder to make it move along the axial constraint. |
| After displacement |  | The right cylinder has translated under the influence of the applied force. However, the applied torque could not rotate them relative to each other as the axial relation is rotationally locked. |
| **Planar Relation**   | Block Diagram   | A Planar Relation establishes a constraint such that two planes must remain parallel. The distance between the planes may be fixed or not. The normals of the two planes may be aligned or anti-aligned. |
| **Example 1**: Diagram of a Planar Align relation constraining the movement of two Planes. |   | |
| Initial Position |  | A plane is attached to another via a planar align relation. A body is attached to one of the planes and forces and torques applied to demonstrate that motion and rotation are constrained to a plane. |

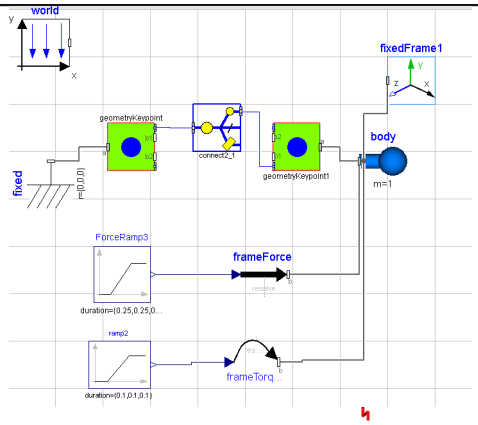
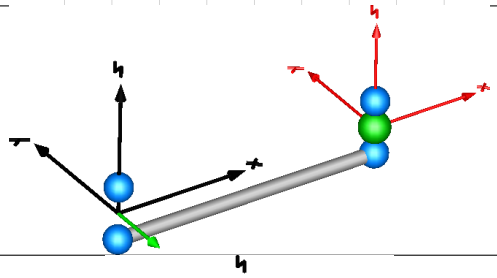
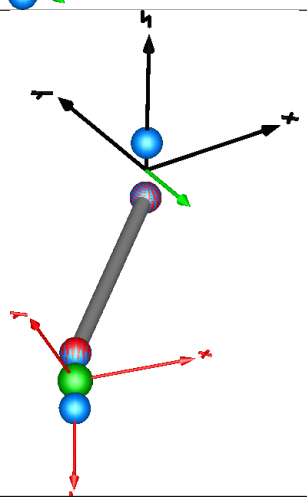
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| After displacement |  | The free body has rotated and moved but both rotation and translation remained in the plane. |
| **Connect Relation**<br>  | Block Diagram<br>  | The Connect Relation establishes a fixed distance constraint between a point and one of the following three elements: a point, a line or a plane. |
| Connection Parameters and<br><br>resultant blocks |  | The user interface of the Connect Relation allows the user to select what geometries it will connect to on either side.<br><br>Three possible connector interfaces are available for Connect together with three corresponding internal constraints which are described next. |
| Connect Point to Point (Internal Component)<br>  | Block Diagram<br>  | This block is not used directly, but through the Connect block. It contains the constraints necessary to model the Point to Point connect assembly relation. |

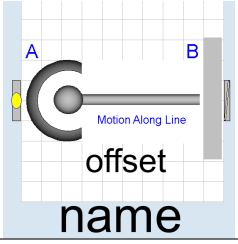
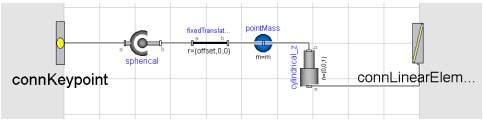
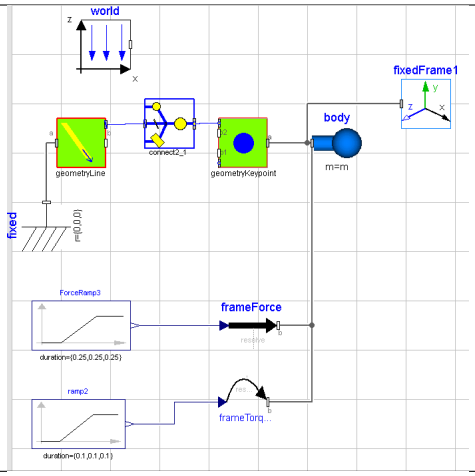
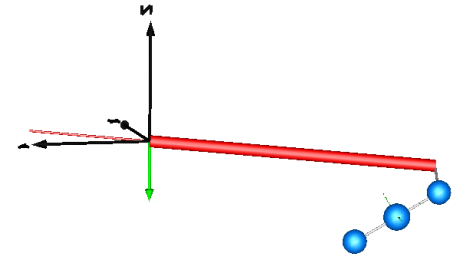
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Example 1**: Diagram of two points connected by a Point-to-Point connect relation |  | |
| Initial Position |  | Two pairs of points are seen on either side of the figure. Each pair represents a single geometryKeypoint model. One point of each pair is connected using the point-to-point connect relation represented as the grey cylinder. The green sphere on the right represents the body. |
| After displacement |  | Forces and torques move the body. The right keypoint pair is fixed and therefore only the right keypoint moves. Notice the distance between the two points remains fixed and that the body (green sphere) has rotated about its connection point. |

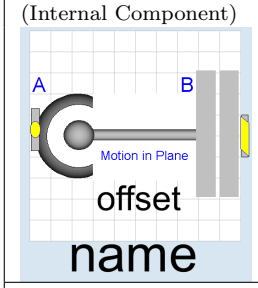
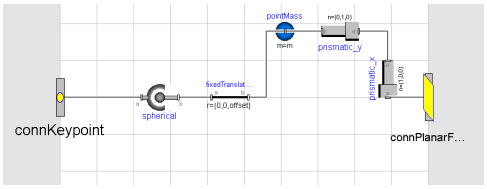
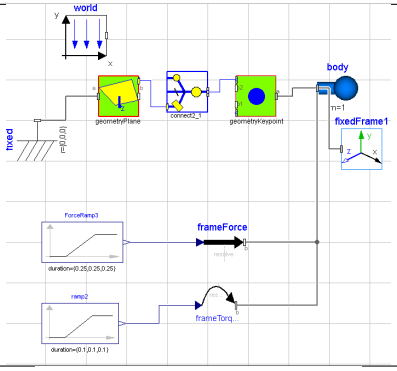
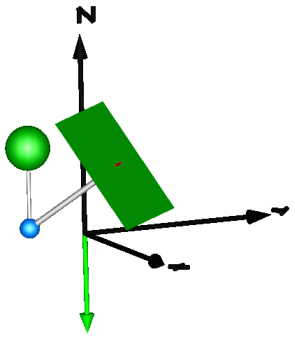
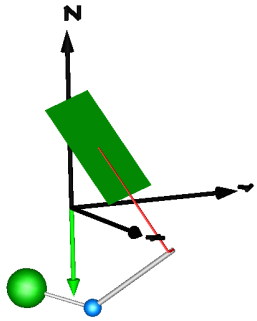
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Connect Point to Line (Internal Component)<br>  | Block Diagram<br>  | This block is not used directly, but through the Connect block. It contains the constraints necessary to model the Point to Line connect assembly relation. |
| **Example 2**: Diagram of a point and a line with a connect relation |  | |
| Initial Position |  | The red cylinder is the line geometry. The three balls on the left represent respectively a point, a body and a point. The first point is connect to the line with a connect relation with a set distance (not visible as its into the plane). The green arrow at the right is the force of gravity. Notice the line is at a sliding angle. |
| After displacement |  | As the forces and torques move the body, the figure shows that the body is free to rotate about the connection point and that the connection point is at a set distance from the line. The connection point has also been pulled down by the force of gravity. |

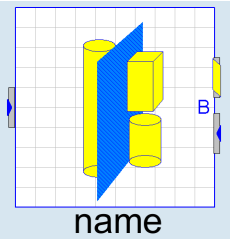
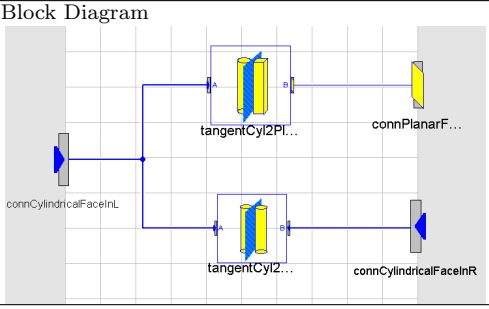
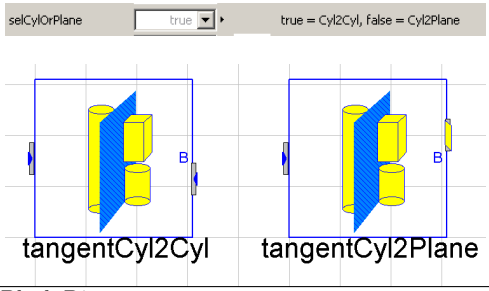
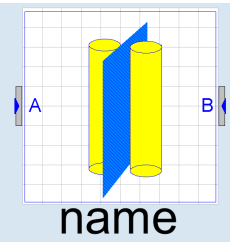
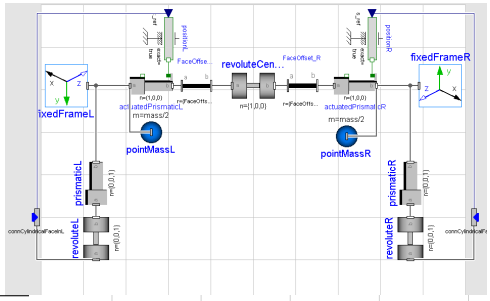
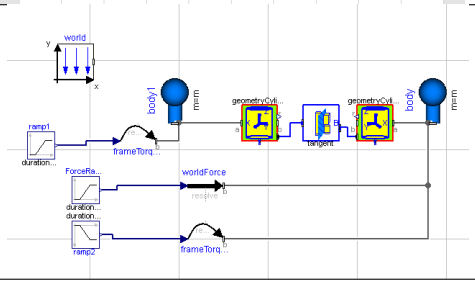
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Connect Point to Plane (Internal Component)<br>  | Block Diagram<br>  | This block is not used directly, but through the Connect block. It contains the constraints necessary to model the Point to Plane connect assembly relation. |
| **Example 3**: Diagram of a point and a plane with a connect relation |  | |
| Initial Position |  | A fixed plane (green surface) is provided. A body is attached to a point which is in turn connected to the plane via a connect relation which will maintain a fixed distance. |
| After displacement |  | Under the action of the forces and torques applied on the body, we see the body pivoting about the point and the point sliding down on a plane (distance fixed relative to the green plane). |

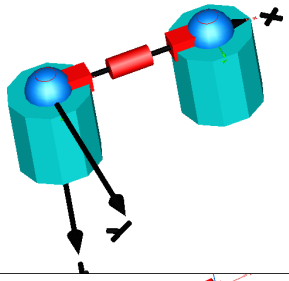
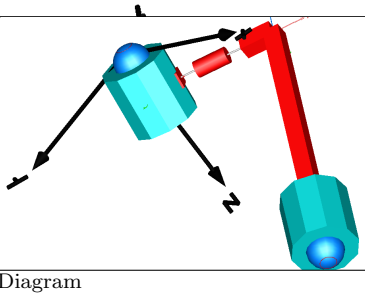
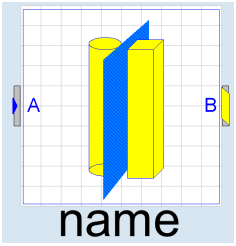
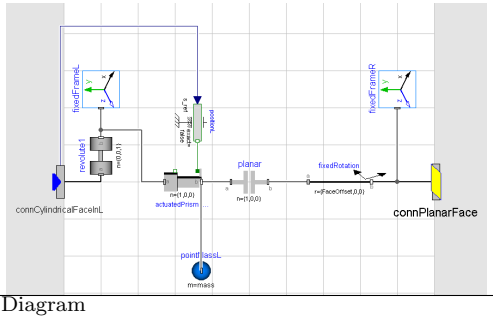
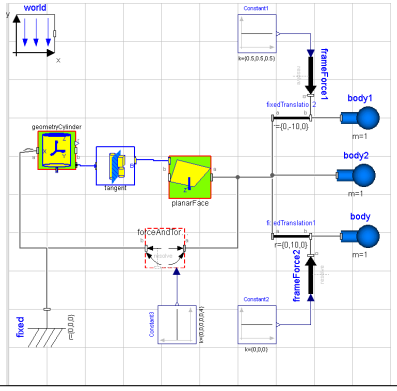
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Tangent Relation**<br>  | Block Diagram<br>  | The Tangent relation applies between a Cylinder and another Cylinder or Plane. The distance between the two geometries must be specified. The Tangent relation also needs as input the Radius of the Cylinders involved. |
| Connection Parameters and<br><br>Resultant Blocks |  | The user interface of the Tangent Relation allows the user to select what geometries it will connect to on either side.<br><br>This generates two possible options of connector interfaces in addition to different internal constraints which are described next. |
| Tangent Cylinder to Cylinder (Internal Component)<br>  | Block Diagram<br>  | This block is not used directly, but through the Tangent block. It contains the constraints necessary to model the Cylinder to Cylinder Tangent assembly relation. |
| **Example 1**: Diagram of a tangent relation between two cylinders. |  | |

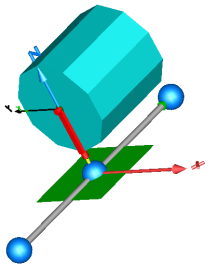
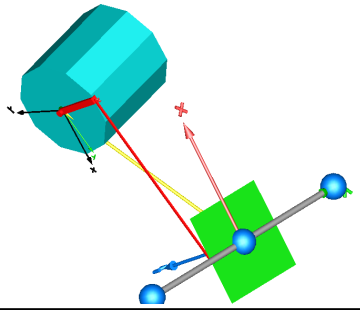
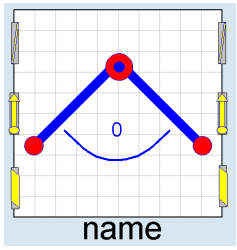
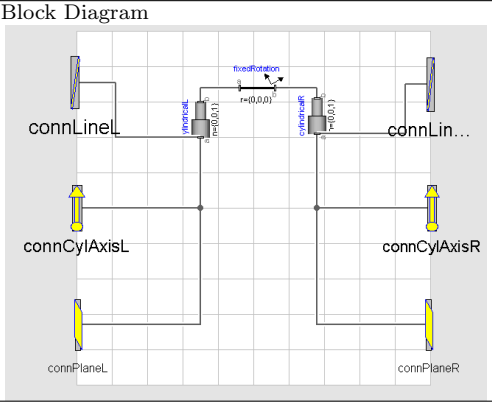
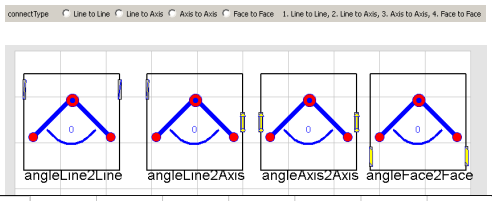
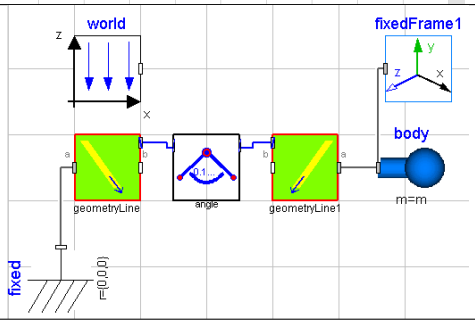
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Initial position |  | Two cylinders are connected with a tangent relation. The rotation degree of freedom is represent by the red cylinder whereas the two translational degrees of freedom by the prismatic red shapes. |
| After displacement |  | Under the action of forces and torques the free cylinder moves demonstrating the rotational and translational motions compliant with a tangent relation. |
| Tangent Cylinder to Plane (Internal Component)   | Block Diagram   | This block is not used directly, but through the Tangent block. It contains the constraints necessary to model the Cylinder to Plane Tangent assembly relation. |
| **Example 2**: Diagram of a tangent relation between a cylinder surface and a plane. | Diagram   | |

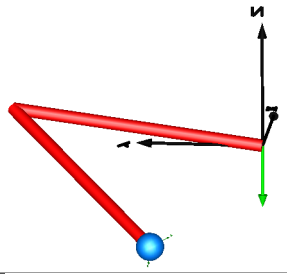
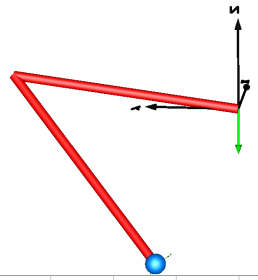
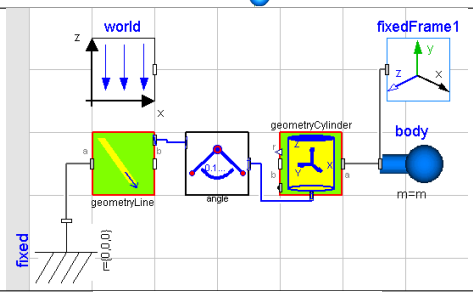
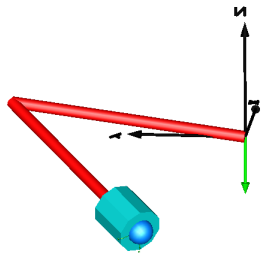
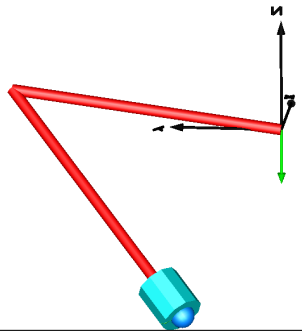
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Initial Position |  | The plane must maintain a fixed distance from the fixed cylinder. The red shape represents the combination of joints linking the two geometries. |
| After displacement |  | The plane has moved and the tangent relation is maintained. |
| **Angle Relation**<br>  | Block Diagram<br>  | The Angle Relation specifies a fixed angle to maintain between lines, axes and Faces. The allowed combinations are Line 2 Line, Line to Axis, Axis to Axis and Plane to Plane. |
| Connection Parameters and<br><br>Resultant Blocks |  | The user interface of the Angle Relation allows the user to select what geometries it will connect to on either side.<br><br>This generates four possible combinations of connector interfaces. |
| **Example 1**; Diagram of a fixed angle relation between to line geometries. |  | |

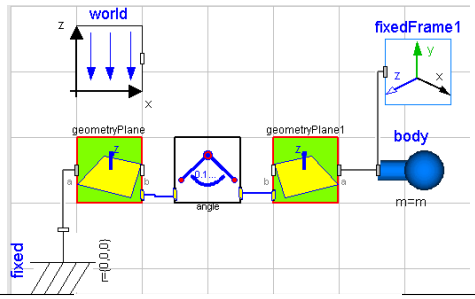
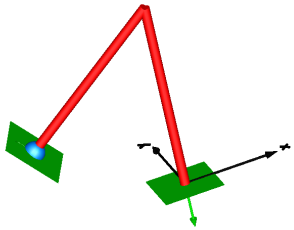
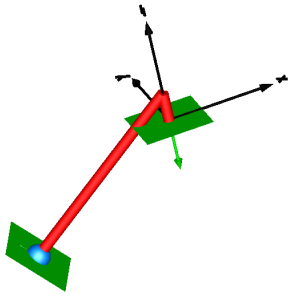
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Initial Position |  | A fixed angle is visualised as two red cylinders. The line geometries and the angle visualisation geometries are overlapping. One end is attached to a fixed frame. The other is attached to a body which will move under the force of gravity. |
| After displacement |  | The body is falling as if in free fall since the angle constraint does not apply any forces in this situation. However if a torque were to be applied in the appropriate direction, the angle constraint would constrain the rotation. |
| **Example 2**; Diagram of a fixed angle relation between a line geometries and the axis of a cylinder. |  | |
| Initial Position |  | A fixed angle is visualised as two red cylinders. One end connects to a line geometry and the other to the cylinder axis. A body is attached to the cylinder. |
| After displacement |  | The body is falling as if in free fall since the angle constraint does not apply any forces in this situation. However if a torque were to be applied in the appropriate direction, the angle constraint would constrain the rotation. |

*continued on next page*

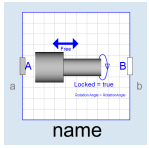
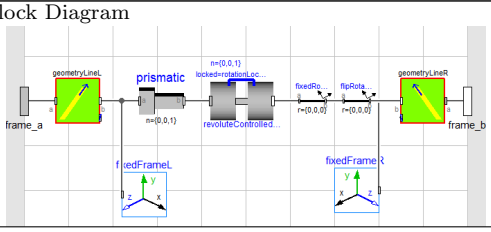
*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Example 3**; Diagram of a fixed angle relation between to plane geometries. | Diagram   | |
| Initial Position |  | A fixed angle constraint is applied between two planes. A body is attached to one plane and is under the influence of gravity (green arrow pointing down). |
| After displacement |  | The body is effectively free fall. Notice that the angle constraint didn't impose a rotation of the falling plane as expected. |

### 4.6.5.4 Assembly Relations: Part to Part version

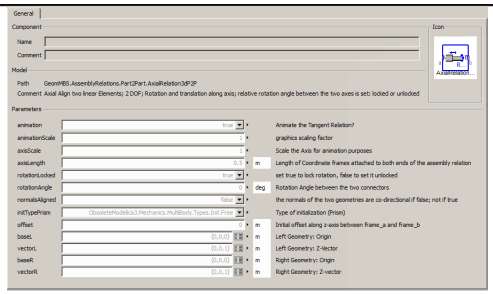
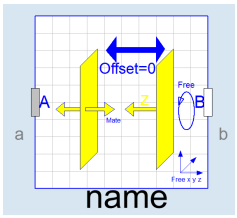
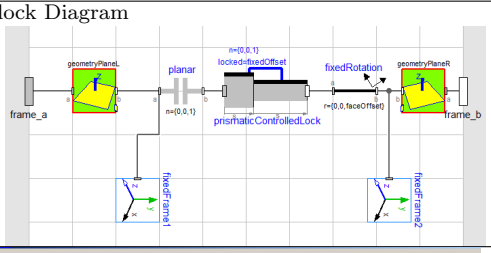
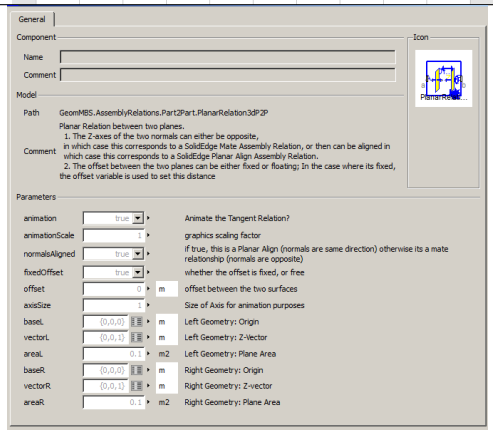
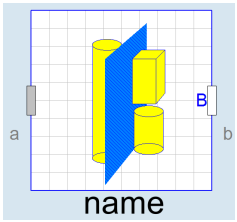
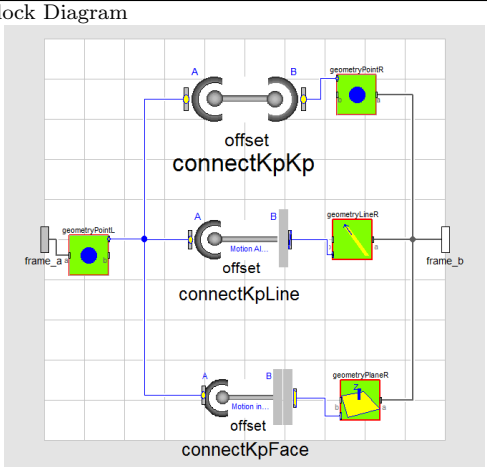
We have created a second version of the Assembly relations where the geometry is included in the Assembly Relation instead of being external to it. We call these **Part-2-Part** Assembly relations as opposed to the **Geometry-to-Geometry** Assembly relations because these can connect to a Part model that has no geometry information.

Table 4.5: Modelica Part-Part Assembly Relations

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Axial Relation**   | Block Diagram   | The Axial Relation block with the geometric entities embedded in the model |

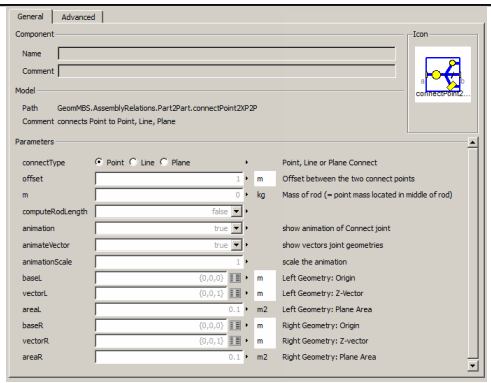
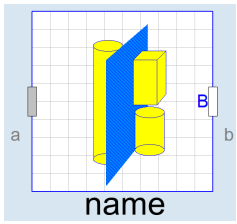
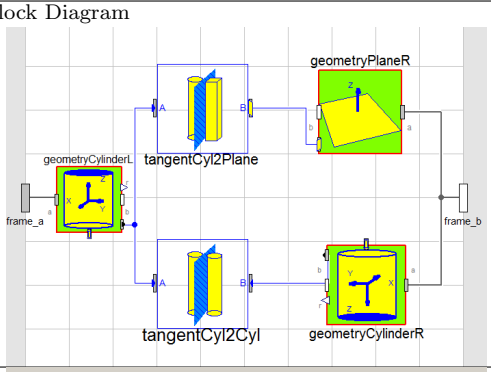
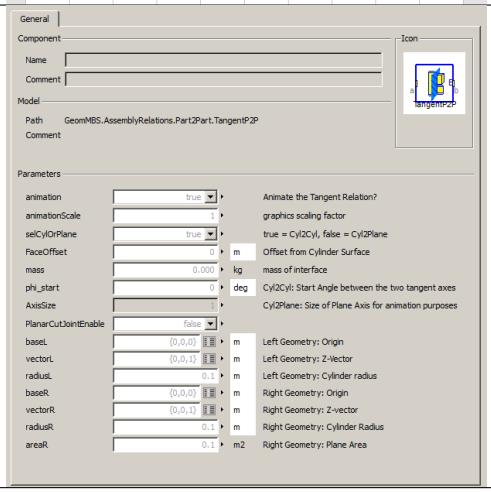
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Connection Parameters |  | The user interface of the Axial Relation allowing the user to set the geometry parameters. |
| **Planar Relation**   | Block Diagram   | A Planar Relation block with the geometric entities embedded in the model |
| Connection Parameters |  | The user interface of the Planar Relation allowing the user to set the geometry parameters. |
| **Connect Relation**   | Block Diagram   | The Connect Relation block with the geometric entities embedded in the model |

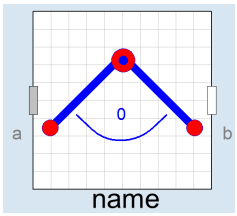
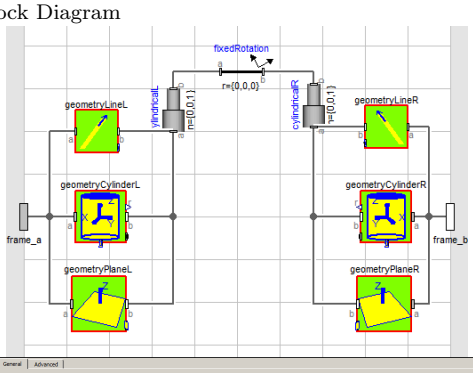
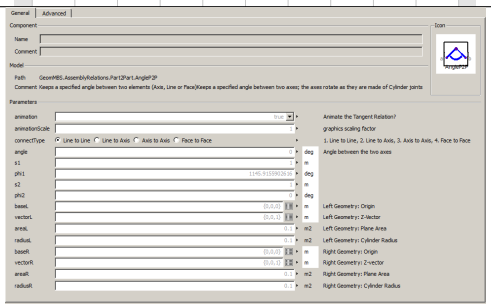
*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| Connection Parameters |  | The user interface of the Planar Relation allowing the user to set the geometry parameters. |
| **Tangent Relation**<br>  | Block Diagram<br>  | The Tangent relation block with the geometric entities embedded in the model |
| Connection Parameters and |  | The user interface of the Tangent Relation allowing the user to set the geometry parameters. |

*continued on next page*

*continued from previous page*

| *Model Icon* | *Details* | *Description* |
|---|---|---|
| **Angle Relation**   | Block Diagram   | The Angle Relation block with the geometric entities embedded in the model |
| Connection Parameters |  | The user interface of the Angle Relation allowing the user to set the geometry parameters. |

There are a few reasons for creating these:

- Minimise the complexity and size of the model: inserting all geometry information as we have done created too many equations to a point where we were Dymola could not check and compile the model.
- The user-interface to deal with so much geometric information is not there in Dymola or any other current Modelica supporting tools. Therefore, we require a more manageable set of models to map Solid Edge into Modelica.

#### 4.6.5.5 Assembly Relations: Modelica.Mechanics version

We can map Solid Edge Assemblies and Assembly relations directly to models constructed only with the Modelica.Mechanics library components (and none of the new geometric elements we have constructed). Instead of creating a specialised library to which Solid Edge Assembly relations can be converted in an isomorphic way, if we forego the isomorphism requirement we can map directly to the Modelica.Mechanics components. The conversion would be equivalent to converting Solid Edge to a Part-To-Part model as we described earlier but where all the geometries are replaced by coordinate transformations and any of our parametrised models by their underlying Modelica.Mechanics components directly. This is also equivalent to a flattened Part-to-Part model and a replacement of all Part-to-Part specific components by their underlying Modelica.Mechanics components.

### 4.6.6 CAD Exporter

In this section, we briefly describe the Solid Edge to Modelica exporter. The exporter takes a Solid Edge Assembly model and convert every composing Part into a corresponding Modelica GeomMBS sePart with or without the geometric information (user selectable) and for each of the set of Points, Lines, Planes, Cylinders array connectors usable by the GeomMBS geometric joints.

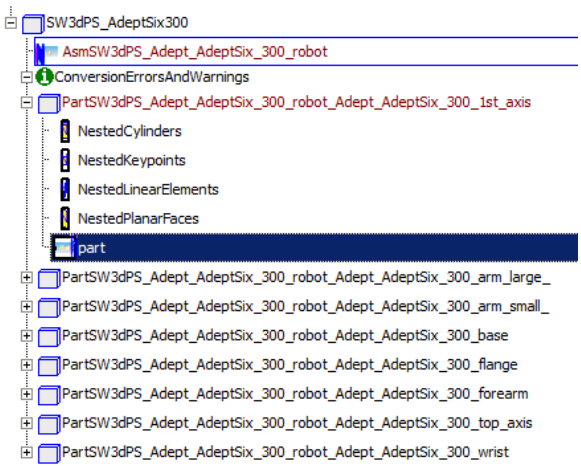



Figure 4.29: Modelica model three hierarchy screen-shot

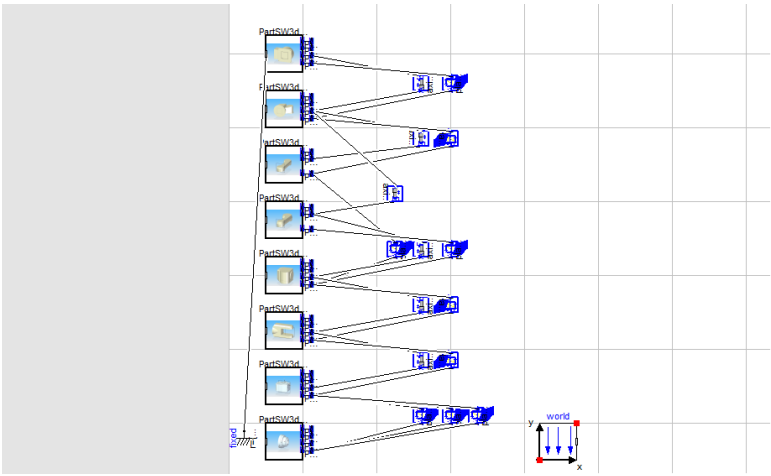



Figure 4.30: Assembly with geometry as generated from Solid Edge

Figure 4.29 shows the Modelica tree hierarchy resulting from the conversion of a representative Solid Edge Assembly of a robot. The first highlighted model represents the Modelica Assembly. The remaining packages contain the models for each Part. One of the Part packages is expanded and contains the four connector array definitions followed by the sePart model.

The diagram representation of the same assembly is seen in figure 4.30. On the left column, we have all the individual Parts. Since in the current Dymola (version 7.3) the user is not allowed to connect to components of a connector, we have duplicated the connectors of every sePart and inserted them directly next to the sePart. This allows to circumvent the previous limitation and select sub-components of a connector. In this case, the sub-component corresponds to an individual geometry connector.

To the right of the figure, we have aligned in rows the various Assembly relations that apply between the seParts. Notice that most Parts have more than one assembly relation applied. Unfortunately, this will create redundant equations which cannot be solved without further manipulation which we have not implemented. Therefore, to be able to run the model we must eliminate all assembly relations except the first between any two Parts. The result of this operation is shown in figure 4.31.

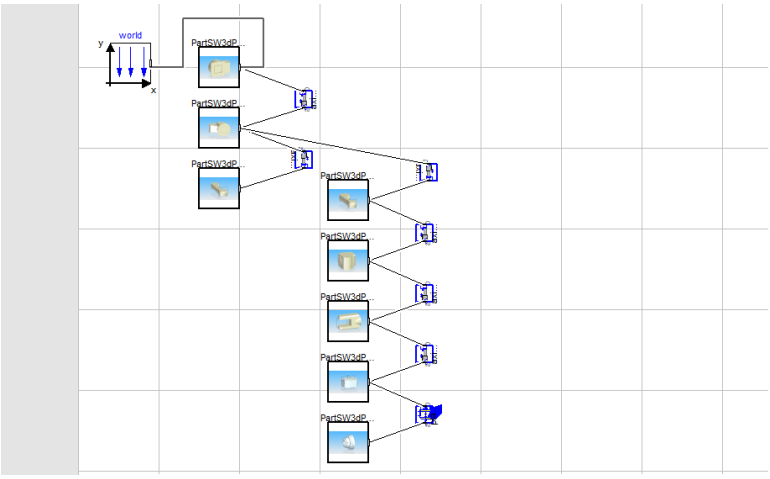



Figure 4.31: Manual elimination of joints to remove equation redundancy

## 4.7 Modelica to CAD Mapping

### 4.7.1 Introduction

In this section we explore some alternatives for mapping Modelica mechanical models back to Solid Edge. The objectives are to:

1. Fill the bi-directional mapping gap by providing a Modelica to CAD mapping in addition to the CAD to Modelica mapping we have already described
2. Provide a Modelica to CAD mapping for the GeomMBS library we have developed
3. Provide Modelica to CAD mapping for the Modelica.Mechanics.MultiBody default MBS library
4. Demonstrate how the limited geometric information contained in Modelica.Mechanics.MultiBody models can be used to generate geometric markers in Solid Edge that will serve as references for refining the geometry

**Mapping Modelica GeomMBS** We have provided equivalents for Solid Edge Parts and Assembly relations in a Modelica library that we referred to as GeomMBS (Geometric Multi-Body System). The GeomMBS was created to closely match elements of Solid Edge and therefore defining the reverse mapping from Modelica to Solid Edge should be relatively straightforward. In mapping GeomMBS to Solid Edge we already have the modelling elements to define specific geometric features (such as points, lines and planes). These were initially derived from geometric elements in Solid Edge and we will show how they can be mapped back although into elements different from what they originated from. This will be one approach for enabling a Modelica user to insert pointers and markers in the Solid Edge model as guiding elements for the construction of more detailed geometry. We will discuss this briefly the following sections.

**Mapping Modelica Mechanics** The Modelica.Mechanics library defines a mechanical assembly of bodies (solid physical model) with interconnecting joints and therefore it should be possible to represent these including their mechanical degrees of freedom using a Solid Edge model.

The limited amount of geometric information we have in a Modelica.Mechanics model will allow us to generate a skeletal Solid Edge model with geometry placeholders for further refinement. This will be

another approach for enabling a Modelica user to insert pointers and markers in the Solid Edge model as guiding elements for the construction of more detailed geometry. We will be covering this item in more detail.

### 4.7.2 Mapping Modelica GeomMBS models to Solid Edge

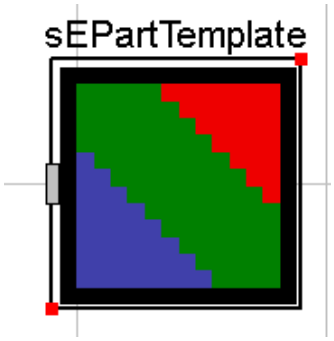
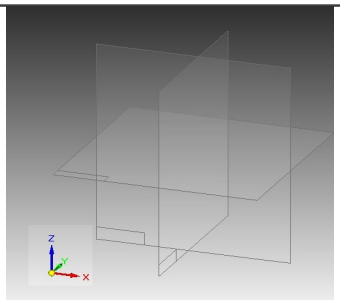
Given that we have defined Modelica counterparts for each of Solid Edge Parts, geometries and assemblies, and that the correspondence was one to one, we can apply the mapping in reverse. The user would start by constructing a Modelica Part with no geometries. Geometric elements would then be added constructing the skeleton of a 3D shape. This model is then converted to Solid Edge with each GeomMBS geometry being represented by an equivalent geometric feature in Solid Edge. These would become the markers that will allow us to refine the Solid Edge model.

#### 4.7.2.1 GeomMBS SEPart

The GeomMBS.Parts.SEPart Modelica model is the element corresponding to the Solid Edge Part model. By default, this model contains only the mass properties of a rigid object and a reference coordinate system. This then corresponds to an empty Solid Edge Part with no geometry.

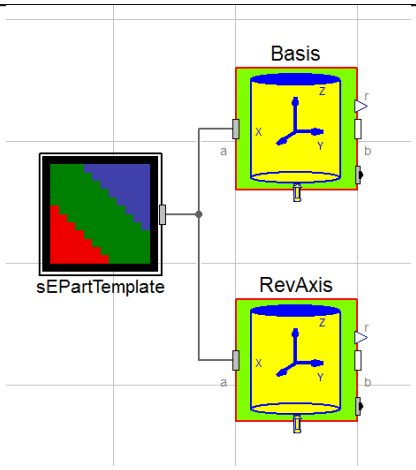
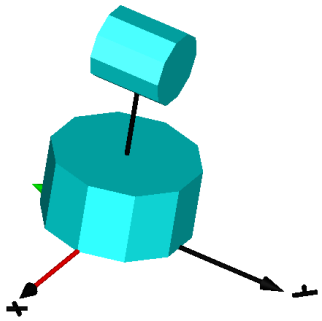
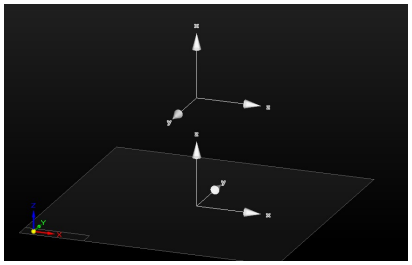
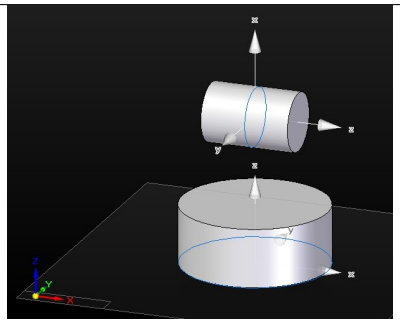
Table 4.6 provides an example of the relations between a GeomMBS SEPart model and a Solid Edge Part as both models are evolving in parallel and being synchronised.

Table 4.6: Evolution of a Body starting in Modelica

| *Description* | *Modelica model* | *Map* | *Solid Edge model* |
|---|---|---|---|
| A Modelica model is mapped to Solid Edge |  Empty Modelica Body with the capacity to grow to contain geometric information | ⇒ |  Model with no geometric features except the default reference framework and mass properties corresponding to those originated in Modelica |

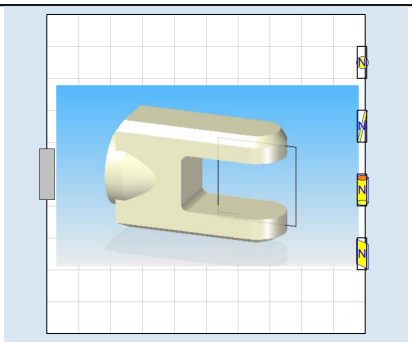
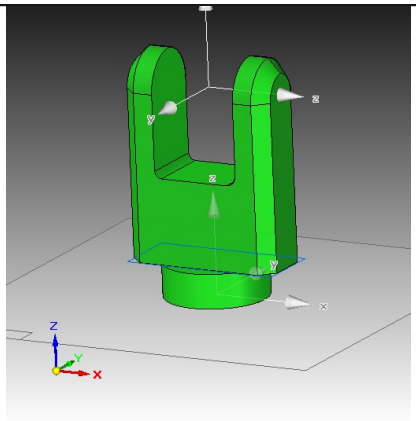
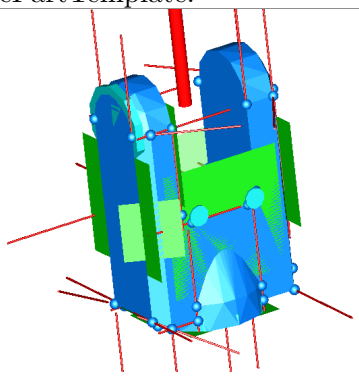
*continued on next page*

*continued from previous page*

| *Description* | *Modelica* | *Map* | *Solid Edge* |
|---|---|---|---|
| Two cylindrical geometries are added to the Modelica model |  Two geometric features attached to model | | Solid Edge model not updated yet |
| The 3D representation of the Modelica Modelica and the automatically generated Solid Edge model |  3D view of model in Modelica | ⇒ |  Automatically generated model. The two reference axes correspond to the cylindrical objects |
| The markers in Solid Edge were used to refine the geometry | Modelica model not synchronised with changes occurring in Solid Edge | |  Manually added geometry starting with the reference axes |

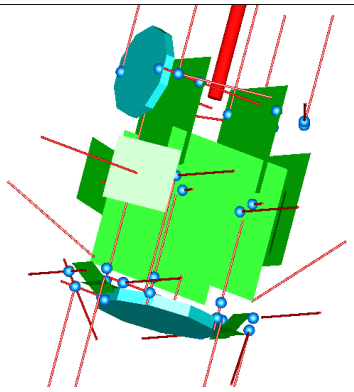
*continued on next page*

*continued from previous page*

| ***Description*** | ***Modelica*** | ***Map*** | ***Solid Edge*** |
|---|---|---|---|
| The markers in Solid Edge were used to further refine the geometry. The Solid Edge model used to update the *Modelica* model. |  Solid Edge model converted to Modelica. All the geometry is now folded inside the initial sePartTemplate. | ⇐ |  Final end-product of manual evolution |
| 3D view in Modelica only |  All the generated geometric features such as points, lines, planes and cylinders are shown in addition to the 3D CAD shape. | | |

*continued on next page*

*continued from previous page*

| *Description* | *Modelica* | *Map* | *Solid Edge* |
|---|---|---|---|
| 3D view in Modelica only |  Same model with the CAD shape hidden to highlight the two initial cylinders recreated from Solid Edge seen at the bottom and left of the figure | | |

#### 4.7.2.2 GeomMBS Geometries Mapping to Solid Edge

In table 4.6 we have seen visual representations of the GeomMBS Point, Line, Plane and Cylinder geometric features. We have also seen that a GeomMBS Cylinder is converted into a Solid Edge reference frame. In fact, we convert all the GeomMBS geometry elements into corresponding Solid Edge reference frame objects taking care to annotate each so as to remember the kind of the source. The following list provides an overview:

- **Point**: A GeomMBS Point is converted to a Solid Edge Reference frame where the origin represents the point
- **Line**: A GeomMBS Line is converted to a Solid Edge Reference frame where the z-axis represents the line
- **Plane**: A GeomMBS Plane is converted to a Solid Edge Reference frame where the origin represents a point on the plane, the xy-plane represents the plane and the z-axis is the plane normal.
- **Cylinder**: A GeomMBS Cylinder is converted to a Solid Edge Reference frame where the origin represents a point on the cylinder axis and the z-axis is the Cylinder axis. The annotation attached to the reference frame contains the value of the cylinder radius.

#### 4.7.2.3 GeomMBS Joints Mapping to Solid Edge

We have discussed the GeomMBS Joints in detail in the previous sections. The mapping we had provided (from Solid Edge to Modelica) can be used in reverse as it was constructed to be a bijective relation.

### 4.7.3 Mapping Modelica.Mechanics models to Solid Edge

It is of interest to consider a mapping from the Modelica.Mechanics library to Solid Edge as this is the default MultiBody System library available to Modelica users. The objective remains the same as

for the GeomMBS mapping and that is to generate a skeleton Solid Edge model which can then be further refined.

The relevant elements in a Modelica.Mechanics model are the Body and Joint models. Unlike GeomMBS, these elements do not explicitly define any geometries but they do it implicitly and we will be leveraging this to generate the Solid Edge skeleton.

Before we proceed with the details of the overall mapping, we need to develop a few points.

#### 4.7.3.1 Mapping Overview

In the MCAD to Modelica Mapping section we presented the mapping of Solid Edge elements to models composed with the Modelica.Mechanics library. Now we present the reverse mapping and we will cover this more rapidly as the details were already discussed.

**Modelica.Mechanics Body** The Modelica.Mechanics Body model is the simplest element in a mechanical assembly representing a rigid mass. The corresponding element in Solid Edge is the Part. Both have a default reference frame and mass properties. The simplest rule is to map a Modelica.Mechanics Body to an empty Solid Edge Part where the mass properties are matched and the reference frames correspond to each other.

**Modelica Composite Models** The hierarchical composite model we can construct in Modelica can be matched with hierarchical assemblies and sub-assemblies in Solid Edge irrespective of whether the Modelica models contain any Body models. When a Body is added at any level of the hierarchy, we can equate this with adding a Solid Edge Part in the corresponding assembly. Another approach might be to map a composite Modelica model by first flattening the mechanical sub-model and then mapping it to a flat Solid Edge Assembly.

**Modelica.Mechanics Joints** For most of the Modelica.Mechanics joints, we can find partial equivalents (equivalent only under restrictions to the Modelica models) in Solid Edge. Table 4.7 presents the Modelica Joints in the first column and the corresponding Solid Edge Assembly Relations (or Joints) in the last. Each Solid Edge Assembly relation requires geometries to connect to. We list these in the $2^{nd}$ and $3^{rd}$ columns. Joints that have no equivalent are also listed.

In order to convert Modelica joints to Solid Edge, we need to generate not only corresponding Assembly relations but also the geometric entities to which the Assembly relation connects. Therefore the equivalent of Modelica joints in Solid Edge consist of at least two or more geometries and Assembly relations linking them.

**Geometry** Modelica.Mechanics joint models do not explicitly define any geometry. However, as we have seen in table 4.7 the creation of joints (Assembly relations) in Solid Edge requires the insertion of new geometric features. These will be the geometric features (or markers) we will be inserting into the Solid Edge Parts and Assemblies when mapping a Modelica multi-body System to Solid Edge and therefore generate the skeleton Solid Edge shapes.

We deduce from the previous table that the following three types of geometries are sufficient to define our constraints: points, lines and planes. These points, line and planes are also specific in that they always fit the origin, axes and planes respectively defined by a coordinate system (or equivalently a reference frame) object as seen in figure 4.32. Moreover, it can be used to represent multiple geometries simultaneously such as a point and a line, two planes or three planes.

For example, the **Line + Point** required for Revolute joint correspond respectively to the origin and one of the axes of a Reference frame. Also, the **Plane** required for a Planar joints corresponds to the plane defined by any two axes of the reference frame. Similar associations exists for the other joints.

Table 4.7: Possible Mapping of Modelica Joints to Solid Edge

| **Modelica Joint** | **SE Geometry1** | **SE Geometry** 2 | **Solid Edge Joint(s)** |
|---|---|---|---|
| Frame connection | 3 ⊥ Planes | 3 ⊥ Planes | 3 Planar Aligns with 0 offsets |
| (or) | 1 Coordinate System | 1 Coordinate System | 3 Planar Aligns with 0 offsets |
| Parts.Fixed | None required | None required | Ground (Partial match) |
| Parts.FixedTranslation | 3 ⊥ Planes | 3 ⊥ Planes | 3 Planar Aligns with offsets |
| (or) | 1 relative Coordinate System | 1 relative Coordinate System | 3 Planar Aligns with offsets |
| Parts.FixedRotation | 1 relative Coordinate System | 1 relative Coordinate System | 3 Planar Aligns |
| Joints.Prismatic | 2 Planes | 2 Planes | 2 Planar Aligns |
| (or) | 1 Axis +1 ∥ Plane | 1 Axis +1 ∥ Plane | 1 Axial Align + 1 Planar Align |
| Joints.Revolute | 1 Axis +1 ⊥ Plane | Axis + ⊥ Plane | Insert |
| (or) | 1 Axis + 1 Point on Axis | 1 Axis + 1 Point on Axis | Insert |
| Joints.Cylindrical | 1 Axis | 1 Axis | Cylindrical |
| Joints.Universal | NA | NA | No Match |
| Joints.Planar | 1 Plane | 1 Plane | Planar Align with 0 offset |
| Joints.Spherical | Keypoint | Keypoint | Connect with 0 offset |
| Joints.FreeMotion | NA | NA | No Assembly Relation |
| SphericalSpherical | Keypoint | Keypoint | Connect with non-zero offset |
| UniversalSpherical | NA | NA | No Match |
| Joints.GearConstraint | Cylindrical Part on Axis | Cylindrical Part on Axis | Gear |

Another choice for geometric objects would have been to use points, lines and planes in Solid Edge. However, this is not advantageous for the following reasons:

1. Points do not exist in and by themselves in Solid Edge. They always appear as the vertices of another geometry such as the ends of a line.
2. Lines are objects that must be given a specific length. The length may be mismatched with the rest of the geometry. i.e. either too small or too big. If they are too small, we may have a hard time locating and selecting them. If they are too big, they may clutter the geometry.
3. Lines will automatically produce two points that will be superfluous.
4. Planes will suffer the same drawbacks we listed for lines. Planes will define extra lines and points. They may be too small or too big.
5. points, lines and planes are more time-consuming to insert and orient compared to achieving the same with Coordinate systems.

In comparison, here are some advantages as well as disadvantages of using Coordinate system objects instead of points, lines and plane objects in Solid Edge.

1. Advantages:
   (a) Coordinate systems are compact and can simultaneously represent points, lines or/and planes.
   (b) Solid Edge provides convenient user-interfaces to manipulate its position and orientation
2. Disadvantages:
   (a) Purpose of axis is ambiguous as it may be representing a point, a line, a plane or a combination.

Fortunately, we can remedy this disadvantage by supplementing the Coordinate system with a textual annotation visible in the drawing and which specifies its purpose. For these reasons, we will be using the Solid Edge Reference frame seen in figure 4.32 to represent the geometric features we will be extracting from the Modelica.Mechanics joints.

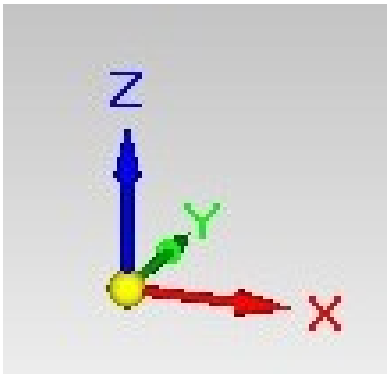



Figure 4.32: Solid Edge Reference Frame

#### 4.7.3.2 Mapping alternatives

Solid Edge assemblies are built by attaching assembly relations to geometric features. When we place any of the Modelica.Mechanics joints into a model, we are in effect introducing a few geometric references as well as one or more constraints. The difference therefore between Modelica.Mechanics and Solid Edge joints is that the former defines geometric information when a joint is inserted into the model while the latter requires the geometric information to be present before an assembly relation can be used. We can deduce two mapping approaches which are equivalent in terms of the mechanical behaviour but where the models would be partitioned differently.

**Geometric Modelica.Mechanics mapping** The first approach will generate Solid Edge Parts, Assemblies and Joints where the Parts would inherit the geometric information and we are then capable of using the Solid Edge joints without requiring the creation of composite joints. The Modelica joint is therefore mapped to the geometry in the Parts and the Assembly relation simultaneously. The one-to-one mapping between Modelica joints and Solid Edge joints is lost. We refer to this as the **Geometric** Modelica.Mechanics mapping.

**One-to-One Modelica.Mechanics mapping** The second approach will generate Solid Edge Parts, Assemblies and joints where new composite joints are created and would simultaneously contain the geometries involved in addition to the Assembly relation. These composite joints would then correspond to the Modelica.Mechanics joints directly and we refer to this as the **one-to-one** Modelica.Mechanics mapping.

**Comparison** The preferred approach is the Geometric mapping as it fulfils the initial goal of creating a skeleton MCAD geometry from a MBS. Here are some of the benefits and drawbacks of the two mappings.

Benefits of the **Geometric** versus the **One-to-One** mapping:

- Builds the skeleton geometry of the CAD Part providing the user with the markers needed to refine it
- Uses the Solid Edge assembly relations directly: no composite assembly relations are needed

Drawbacks of the **Geometric** versus the **One-to-One** mapping:

- The mapping from Modelica to Solid Edge is no longer one-to-one but rather one Modelica joint is mapped to geometric elements in two Parts and to one Assembly relation

### 4.7.4 Geometric Modelica.Mechanics mapping

The Geometric Modelica.Mechanics mapping allows the transfer of intrinsic geometric information defined through Modelica.Mechanics joints to skeleton geometries in the Solid Edge models. This skeleton geometry originates from any Modelica.Mechanics elements that define a geometric position or orientation such as is the case for Modelica.Mechanics Reference frames, Joint axes, etc. The Reference frames provide relative positions with respect to other coordinate systems. They would be converted to Solid Edge Coordinate frames positioned relative to other coordinate frame or geometries. Joints will intrinsically define geometric elements such as points, axes, planes or combinations thereof. For example a cylindrical joint defines a direction of revolution and sliding motion which can be represented by a single line (a point and a direction vector). Such markers can then be used for a double purpose: to provide the geometric connection elements for Assembly relations and to further refine the geometry.

Figures 4.33 and 4.34 show a Modelica.Mechanics Body and its equivalent representation in Solid Edge when no geometry is yet implied on the Modelica side. The sphere represents the centre of mass and the coordinate frame represents the Modelica Body frame.

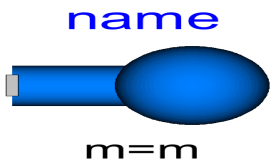



Figure 4.33: Modelica Mechanics Body

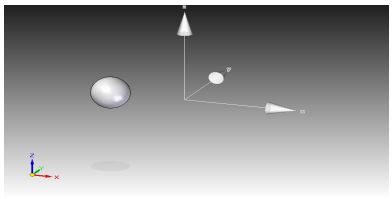


Figure 4.34: Corresponding Solid Edge model

Figure 4.35 shows a Modelica model with two rigid bodies linked by a cylindrical Joint. This model is converted to Solid Edge producing various geometric markers as seen in figure 4.36. The additional coordinate frames are inferred from the cylindrical joint.

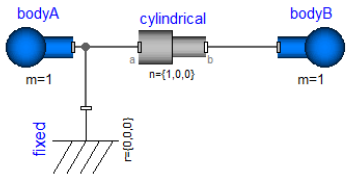



Figure 4.35: Body assembly and Cylindrical joint

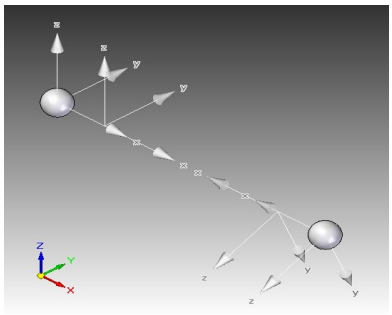


Figure 4.36: Initial Solid Edge Assembly model

Figure 4.37 shows one of the two Solid Edge Parts where the presence of cylindrical joint caused the addition of the additional coordinate frame which is marked with a textual annotation to indicate that it represents a Cylindrical axis. We have then used this axis to refine the geometry and added a cylinder shape.

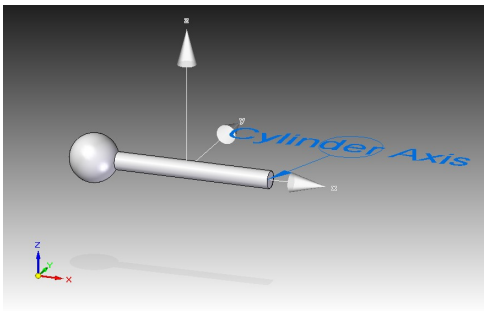



Figure 4.37: BodyA (or B) with refined geometry and annotation

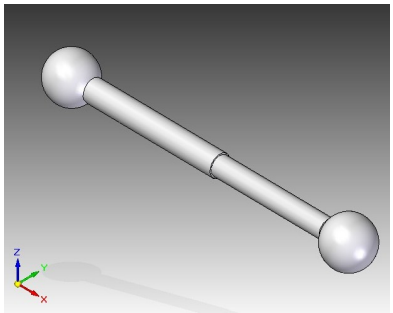


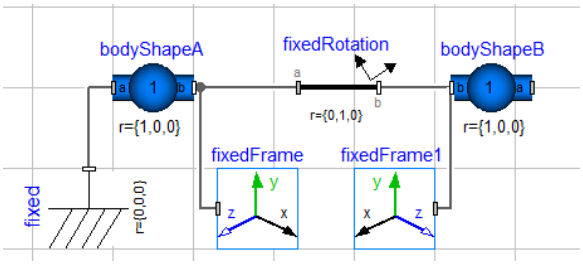
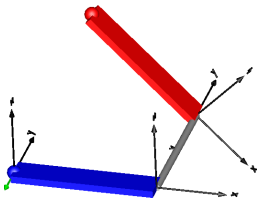
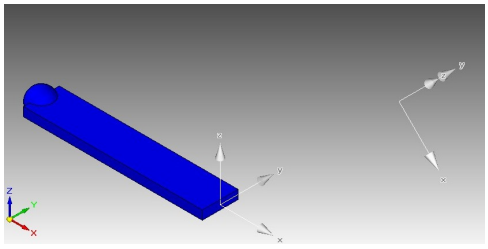
Figure 4.38: Assembly With Refined BodyA and BodyB Geometries

Figure 4.38 shows how the skeletal Solid Edge model was further refined and a hydraulic “shock” was created.

We have worked out this technique to the various Modelica.Mechanics joints and we present them here. We go through them individually in the following paragraphs. The tables are self-explanatory.

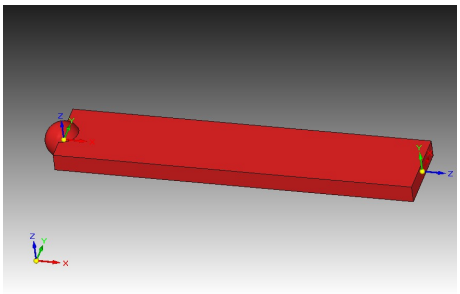
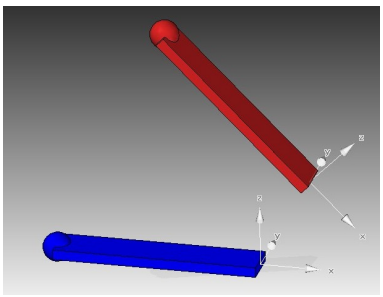
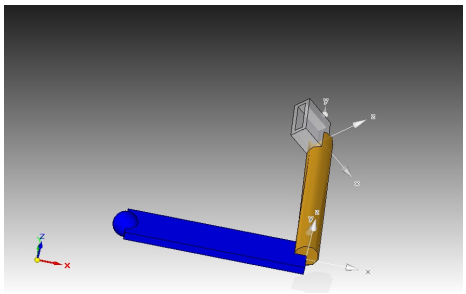
### Fixed Rotation

Table 4.8: Conversion of a Modelica **fixedRotation** model and its evolution

| ***Description*** | ***Model View*** |
|---|---|
| **Modelica *fixedRotation***: example diagram<br><br>and its 3D representation |  <br>Modelica diagram of a two body system linked by a Fixed rotation. *bodyShapeA* and *bodyShapeB* are the two bodies. *fixedRotation* is the joint. The remaining elements are accessories.<br><br>  <br>The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The grey rod connecting them represents the fixedRotation model. |
| **Solid Edge bodyShapeA**: fixedRotation merged |  <br>The Modelica fixedRotation and bodyShapeA models merged into one Solid Edge Part. |

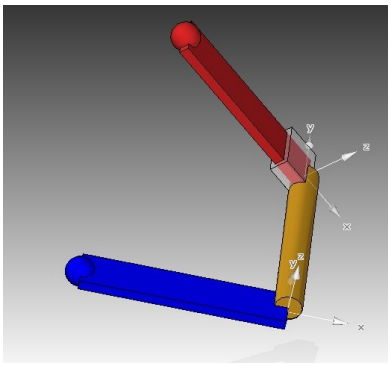
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| | The Solid Edge representation of bodyShapeA and fixedRotation was made to look the same as that in Modelica.<br>Notice the coordinate frame at the RHS. This represents the end point of the fixedRotation. |
| **Solid Edge bodyShapeB**: the Modelica bodyShapeB converted. |  <br>This contains only information from the Modelica bodyShapeB. It was a matter of choice to include the Modelica fixedRotation into the Solid Edge bodyShapeA instead of bodyShapeB. |
| **Solid Edge fixedRotation assembly**: results of conversion from Modelica |  |
| **Solid Edge bodyShapeA**: geometric evolution |  <br>The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the fixedRotation were used to define a more detailed geometry. Notice the receptacle shape for mechanically attaching the bodyShapeB. |

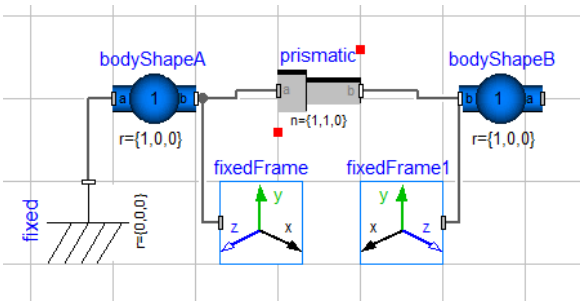
*continued on next page*

*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge fixedRotation assembly**: bodyShapeA, bodyShapeB and a Match Coordinates assembly relation |  Final result of the evolution |

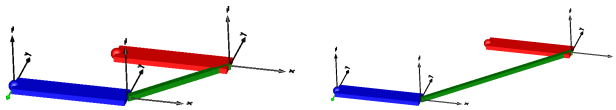
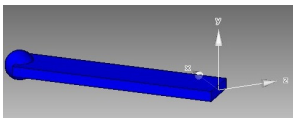
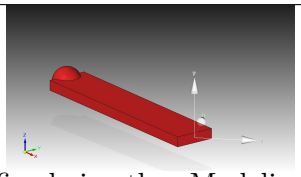
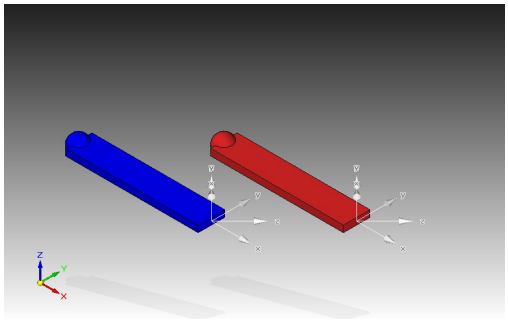
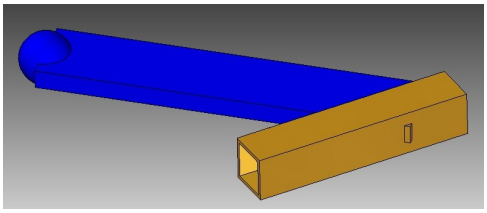
## Prismatic

Table 4.9: Conversion of a Modelica **Prismatic** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *Prismatic*:** example diagram |   Modelica diagram of a two body system linked by a Prismatic joint with only one degree of translational freedom. *bodyShapeA* and *bodyShapeB* are the two bodies. *prismatic* is the Prismatic joint. The remaining elements are accessories. |

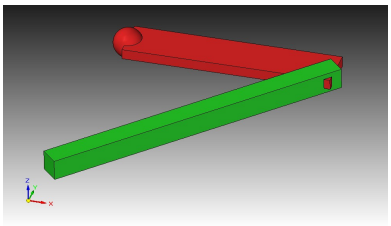
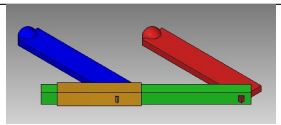
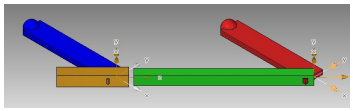
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| and its 3d representations. |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The green prismatic rod represents the prismatic joint. We have shown the system in two different configurations to emphasise the translational degree of freedom. |
| **Solid Edge bodyShapeA and bodyShapeB**: the prismatic joint's direction vector is merged |   The motion constraint defined in the Modelica prismatic joint was converted to a Coordinate frame with its z-axis oriented along the direction of motion and inserted into both Solid Edge Part. |
| **Solid Edge prismatic assembly**: results of conversion from Modelica |  |
| **Solid Edge bodyShapeA**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the prismatic joint were used to define a more detailed geometry. bodyShapeA was given the female prismatic shape. |

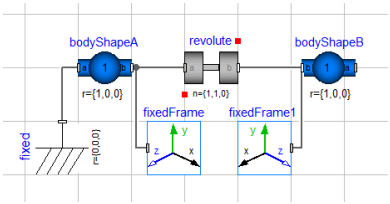
*continued on next page*

*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge bodyShapeB**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the prismatic joint were used to define a more detailed geometry. bodyShapeB was given the male prismatic shape. |
| **Solid Edge prismatic assembly**: bodyShapeA, bodyShapeB, Axial Align and Planar Align assembly relations |   Final result of the evolution. An Axial align is applied between the (invisible) Z-axes and a Planar align between the XZ or YZ-planes. The system is shown in two different allowed positions. |

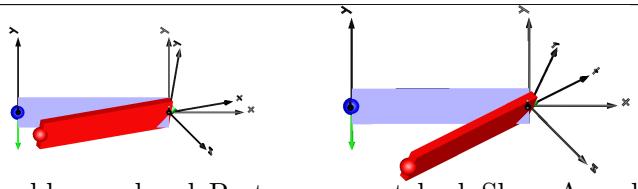
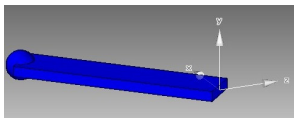
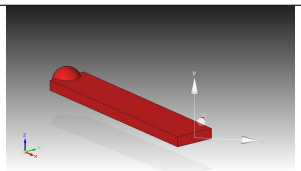
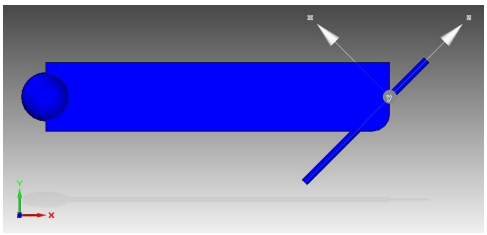
## Revolute

Table 4.10: Conversion of a Modelica **Revolute** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *Revolute*:** example diagram |   Modelica diagram of a two body system linked by a Revolute joint with only one rotational degree of freedom. *bodyShapeA* and *bodyShapeB* are the two bodies. *revolute* is the Revolute joint. The remaining elements are accessories. |

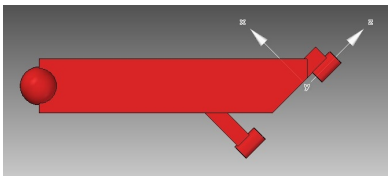
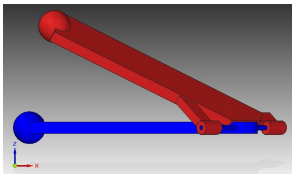
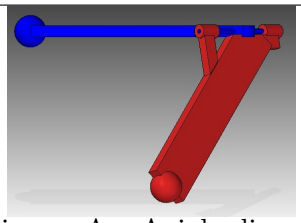
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| and its 3d representations. |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The green cylindrical rod at the intersection of both shapes represents the revolute joint. We have shown the system in two different configurations to emphasise the rotational degree of freedom. |
| **Solid Edge bodyShapeA and bodyShapeB**: the revolute joint's direction vector is merged |   The motion constraint defined in the Modelica revolute joint was converted to a Coordinate frame with its z-axis oriented along the direction of motion and inserted into both Solid Edge Part. The XY-planes in each will define the planar constraint of the revolute joint. |
| **Solid Edge bodyShapeA**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the rotation joint were used to define a more detailed geometry. |

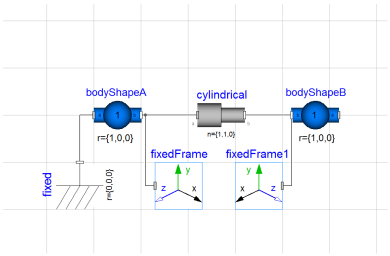
*continued on next page*

*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge bodyShapeB**: geometric evolution |  The Solid Edge bodyShapeB Part has evolved. The frame coordinates that were inherited from the revolute joint were used to define a more detailed geometry. |
| **Solid Edge prismatic assembly**: bodyShapeA, bodyShapeB, Axial Align and Planar Align assembly relations |   Final result of the evolution. An Axial align is applied between the (invisible) Z-axes and a Planar align between the two XY-planes. The system is shown in two different allowed positions. |

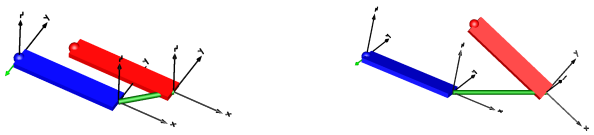
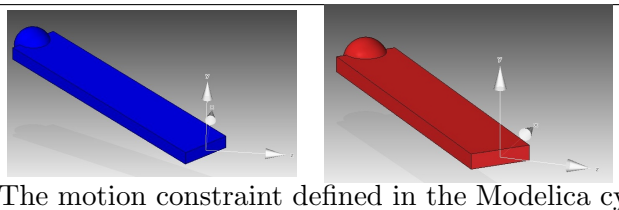
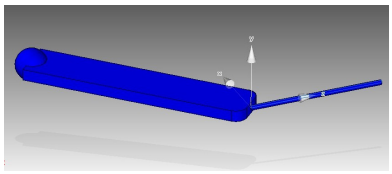
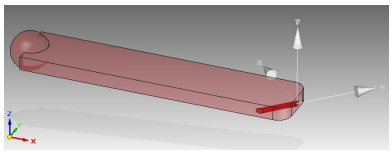
## Cylindrical

Table 4.11: Conversion of a Modelica **Cylindrical** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *Cylindrical*:** example diagram |   Modelica diagram of a two body system linked by a Cylindrical joint with one degree of translational and one degree of rotational freedom. *bodyShapeA* and *bodyShapeB* are the two bodies. *cylindrical* is the Cylindrical joint. The remaining elements are accessories. |

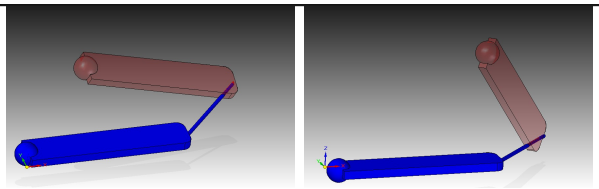
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| and its 3d representations. |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The green rod represents the cylindrical joint. We have shown the system in two different configurations to emphasise the one rotational and one translational degrees of freedom. |
| **Solid Edge bodyShapeA and bodyShapeB**: the cylindrical joint's direction vector is merged |  The motion constraint defined in the Modelica cylindrical joint was converted to a Coordinate frame with its z-axis oriented along the direction of motion and inserted into both Solid Edge Part. |
| **Solid Edge bodyShapeA**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the cylindrical joint were used to define a more detailed geometry. |
| **Solid Edge bodyShapeB**: geometric evolution |  The Solid Edge bodyShapeB Part has evolved. The frame coordinates that were inherited from the prismatic joint were used to define a more detailed geometry. Notice the cylindrical hole. |

*continued on next page*

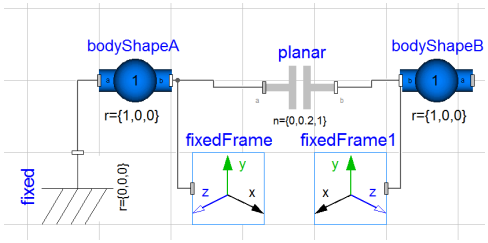
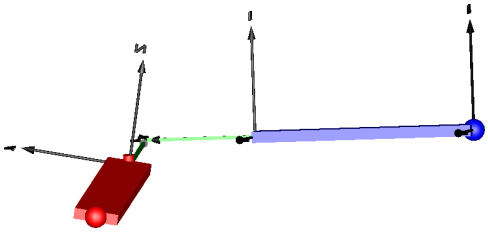
*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge cylindrical assembly**: bodyShapeA, bodyShapeB and Axial Align relation |  Final result of the evolution. An Axial align is applied between the (invisible) Z-axes. The system is shown in two different allowed positions. |

**Universal** Not implemented

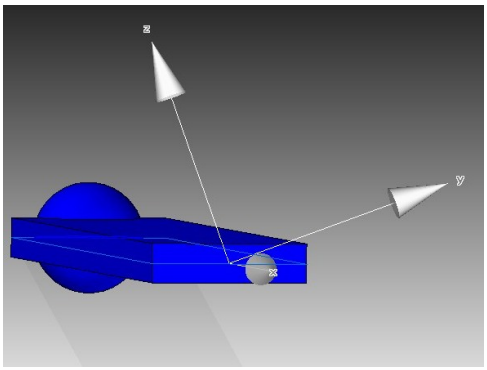
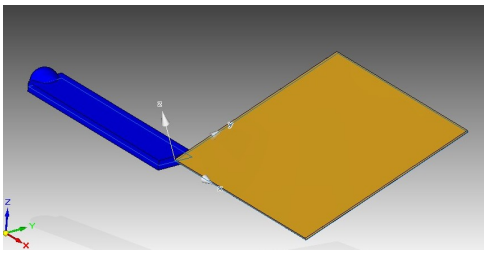
**Planar**

Table 4.12: Conversion of a Modelica **Planar** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *planar***: example diagram |   Modelica diagram of a two body system linked by a Planar joint *bodyShapeA* and *bodyShapeB* are the two bodies. *planar* is the joint. The remaining elements are accessories. |
| and its 3D representation |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The two green prismatic shapes (in the middle) connecting them represent the two degrees of translation freedom. A cylindrical shape (collinear with the z-axis of the bodyShapeB (red)) represents the rotation degree of freedom. |

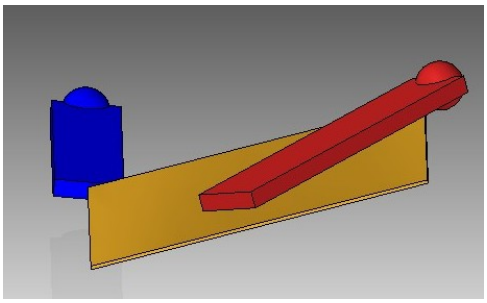
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| **Solid Edge bodyShapeA**: planar joint merged |  The Modelica planar joint and bodyShapeA models merged into one Solid Edge Part. Notice the coordinate frame at the RHS. This represents the planar joint's plane of motion. |
| **Solid Edge bodyShapeA**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. The frame coordinates that were inherited from the fixedRotation were used to define a more detailed geometry. Notice the plane for mechanically constraining the motion of bodyShapeB. |

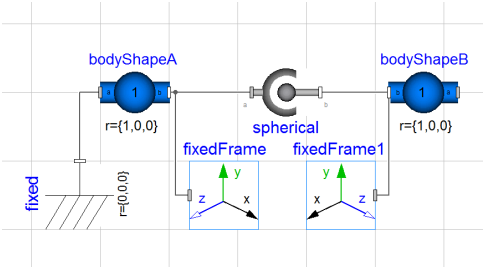
*continued on next page*

*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge planar assembly**: bodyShapeA, bodyShapeB and a planar align assembly relation |  Final result of the evolution. bodyShapeB (red Part) is sliding on the plane shape of bodyShapeA. |

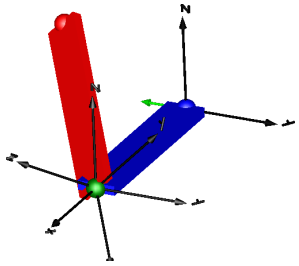
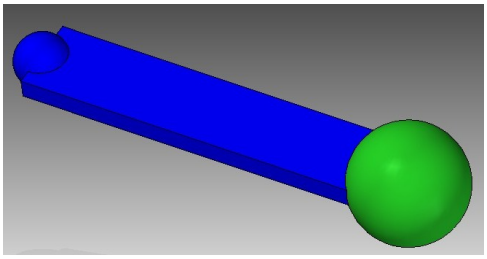
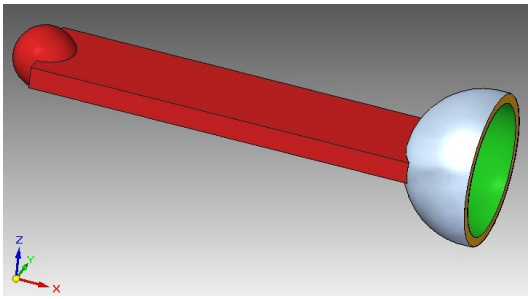
### Spherical

Table 4.13: Conversion of a Modelica **Spherical** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *spherical***: example diagram |   Modelica diagram of a two body system linked by a Spherical joint *bodyShapeA* and *bodyShapeB* are the two bodies. *spherical* is the joint. The remaining elements are accessories. |

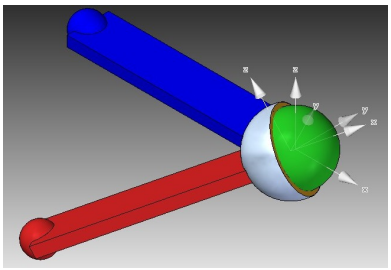
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| and its 3D representation |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The green spherical shape connecting them represent the spherical degree of freedom. |
| **Solid Edge bodyShapeA**: geometric evolution |  The Solid Edge bodyShapeA Part has evolved. |
| **Solid Edge bodyShapeB**: geometric evolution |  The Solid Edge bodyShapeB Part has evolved. |

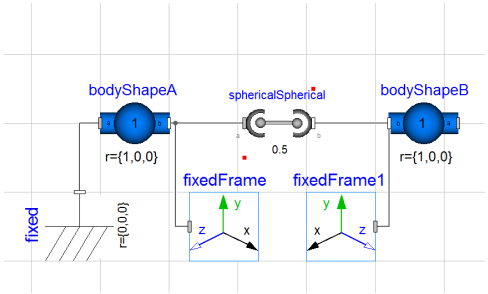
*continued on next page*

*continued from previous page*

| *Description* | *Model View* |
|---|---|
| **Solid Edge spherical assembly**: bodyShapeA, bodyShapeB and a point-to-point connect assembly relation |  Final result of the evolution. bodyShapeA and bodyShapeB are connected by a point-to-point connect assembly relation connecting the centres of the two spherical shapes. |

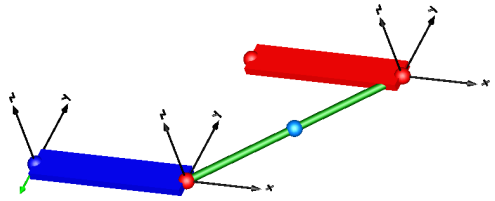
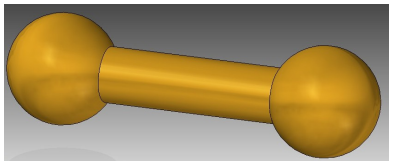
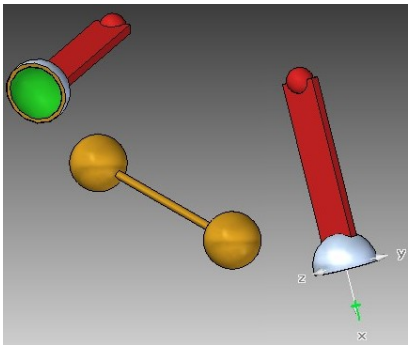
### SphericalSpherical

Table 4.14: Conversion of a Modelica **Spherical-Spherical** model and its evolution

| *Description* | *Model View* |
|---|---|
| **Modelica *spherical-spherical***: example diagram |   Modelica diagram of a two body system linked by a SphericalSpherical joint *bodyShapeA* and *bodyShapeB* are the two bodies. *sphericalSpherical* is the joint. The remaining elements are accessories. |

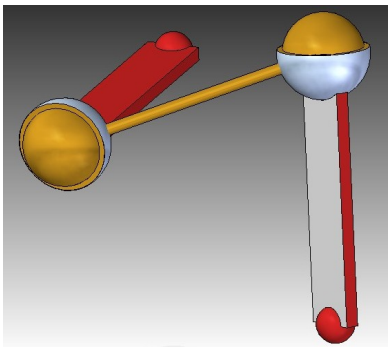
*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| and its 3D representation |  The blue and red Parts represent bodyShapeA and bodyShapeB respectively. The green rod with its two spherical red endpoints is the spherical-spherical joint. It is composed of two individual spherical joint with a rigid (green) rod separating the two. |
| **Solid Edge SphericalSphercial joint representation** |  Unlike previous examples where bodyShapeA or/and bodyShapeB's Parts were connected directly to each other, in the SphericalSpherical joint scenario we need to introduce a third Part that connects the two. |
| **Solid Edge SphericalSpherical assembly**: bodyShapeA, bodyShapeB, SphericalSpherical before constraints are applied |  This shows the state of the assembly before the two constraints are applied to connect the SphericalSpherical Part to bodyShapeA and bodyShapeB respectively. |

*continued on next page*

*continued from previous page*

| ***Description*** | ***Model View*** |
|---|---|
| **Solid Edge SphericalSpherical assembly**: bodyShapeA, bodyShapeB, SphericalSpherical and two point-to-point connect assembly relations |  Final result of the evolution. bodyShapeA and bodyShapeB are connected via a third Part (SphericalSpherical Part) and two point-to-point connect assembly relations. |

**UniversalSpherical** Not implemented

**Gear Constraint** Not implemented

# 5
# CONCLUSIONS AND FUTURE WORK

Current integration between MCAD models and MBS simulation tools is unidirectional from the former to the latter. We sought to provide a mapping approach demonstrated partially through implementation and partially through documented examples that a MBS to MCAD mapping is feasible and furthermore this mapping is coherent with its opposite MCAD to MBS mapping.

We first developed a Modelica library that possessed an almost one to one mapping to the components in Solid Edge and later showed our automated implementation of this mapping. The principal feature of the Modelica language that made this possible is that it is an equation based modelling language. The details of why this works were discussed in great details in the thesis.

We then detailed our new contribution to the subject which is the Modelica MBS to MCAD mapping. With the Modelica library we had developed earlier, the mapping was intended to be simple. A more interesting mapping was between the default Modelica.Mechanics library and the MCAD tool. This paves the way to implementations from a wide variety of MBS modelling tools onto MCAD tools. It is to be noted that the MCAD to Modelica mapping could have been made much simpler if there was not the particular requirement that the reverse mapping be also possible. This requirement lead to a lengthy development of new types, models and joints in Modelica to replicate as closely as possible the mapped components from MCAD.

We have learned valuable lessons in trying to implement the bidirectional mapping. In the following sections we will go through some of the problems we encountered and recommend solutions. We will then conclude by provide some directions for future work.

## 5.1 Modelica limitations

In trying to create the Modelica MCAD library, we have encountered several limitations in both the Modelica language and Dymola.

### 5.1.1 Mismatch in hierarchy concepts

**Problem** The concept of mechanical assembly in MCAD tools is closely matched by the concept of model composition in BD modelling. However, the analogy cannot be pushed too far. If we consider the interfaces available in both cases, then there is a marked difference. In a mechanical, all the parts composing it may be visible or not but still a joint constraint can be established with other parts. In other words, the parts in the assembly are accessible by default. On the contrary, the model composition concept serves the purpose of hiding all of its contents and their interfaces, and any interface it provides is customized. Therefore, given our goal to replicate the MCAD model operations in BD models as well, we require that the composite BD model corresponding to the MCAD mechanical assembly provides an interface to all its parts down to the deepest level.

To implement this in Modelica, none of the current available mechanisms are completely satisfactory.

The most robust approach is to propagate the connections from the deepest component up to the parent model. Unfortunately, this requires that the connectors at every level be adjusted any time that a child component is added or removed. Another approach is based on the use of expandable connectors, however the current implementation in Dymola is inadequate in addressing this problem.

**Recommended solution: browsable connector** In the current Dymola implementation of Modelica, accessing connectors hidden within another model is possible as long as the connect equation is stated in text. The graphical user interface does not allow such a connection to be made to preserve encapsulation. Our recommendation requires additions both to the language and to the tool:

- At the language level, add the new "browsable" keyword or prefix applicable to connectors which will make them visible and browsable when they are hidden within other models.
- At the tool level, provide the user with a mechanism to explore in a hierarchical tree all the browsable connectors within a model in order to make a connection. Visual indications may also be require to trace such a connection through intervening parents in the graphical browser.

### 5.1.2 Connector limitations

**Problem** The connector concept in a BD model is intended to provide a limited number of interfaces to a model. It runs into trouble when we require hundreds of connectors to a single model. In MCAD, a joint constraint can be applied to any of the geometric features associated with a given part. The number of choices can easily run into the hundreds for a moderately complicated geometry. This is then the problem we faced when we required one Modelica connector per geometric feature available in a MCAD part. Having hundreds of connectors on a model was not an option as it would have lead to graphic clutter and also it would have been very hard to find the appropriate connector. Our solution was to create nested connectors (i.e. one connector containing many sub-connectors). We now had a single connector (call it instanceA of type **nestedConnectorX**) appearing on the model. However, encapsulation rules implemented in Dymola's graphical interface forbid accessing nested sub-connectors. We side-stepped the problem by inserting an **instanceB** of nestedConnectorX which can connect to **instanceA** and then make our connection to any of the sub-connectors of **instanceB**.

**Recommended solution: browsable connector** The solution to this problem is the same as that suggested above. The user can then choose to make sub-connectors browsable therefore explicitly requiring that encapsulation rules not be applied.

### 5.1.3 Parameter limitations

In the course of developing the Modelica libraries in this thesis, we have encountered several limitations related to parameters.

#### 5.1.3.1 Limited applicability of external functions

**Problem** The Modelica language may be insufficient or inadequate to address certain problems, or it might happen that we have C-functions we wish to use. Unfortunately, in Dymola 7.3 such external functions cannot be used to initialize either constants or parameters. Only pure Modelica functions are allowed.

**Recommended solution** Allow external functions to initialize constants and parameters.

#### 5.1.3.2 Limited initialization options

**Problem** Parameter declaration and initialization are combined. This leads to inefficiencies or inconvenience when a single algorithm could have been used to calculate multiple parameter values at once.

**Recommended solution: algorithm parameter** In the same way that Modelica provides the **initial algorithm** and **initial equation** sections, add a new Modelica code section (call it **parameter algorithm**) where an algorithm can initialize constants and parameters.

#### 5.1.3.3 Parameter exchange

**Problem** Input and output signals allow the propagation of variables across models during simulation. It would be very useful in many problems that a similar capability for propagating parameter values across models existed. For example, in a hydraulic system where each component requires the parameters of the liquid in the pipe, with the current Modelica specifications we need to initialize each component with the same parameter values. Current solutions to this problem in Modelica require the use of a shared parameter. Another approach is to propagate the parameters through the connectors which is akin to the behaviour in the real system where the liquid propagates from component to component.

In this thesis, such a parameter exchange capability would have provided an elegant and much less demanding alternate solution for the problem of multiple connectors described above. Each connector was used to provide a mechanical reference frame attach to each geometric feature in a MCAD part. Therefore, we had to introduce a coordinate transformation with each such connector. Unfortunately, out of all the hundreds of connectors and associate coordinate transformations that we would introduce, a connection would be established to just a few of them. Therefore, a considerable number of equations are introduced and not used.

If it were possible to exchange parameters across connectors, then the parameters defining all the geometric features could have been shared across a single connector attached directly to the principal reference frame of the part. The connecting model would then index into these parameters and select the set that corresponds to the geometric feature of interest. A single coordinate transformation would then need to be initialized instead of hundreds.

**Recommended solution: input/output parameter** Extend the use of the input/output prefixes to apply to parameters the same way that it applies to variables.

### 5.1.4 User-interface limitations

The Modelica model user-interface is adequate in most cases, but it cannot be extended beyond what is allowed through the annotations and the support provided by the tool (Dymola). We will detail some of these limitations.

#### 5.1.4.1 Limited user-interface logic

**Problem** The user interface of models can be controlled to a limited extent using simple logic and annotations. For example, if we wish to present a dynamic user interface that adjusts based on user choices, we can hide unused parameters. Unfortunately, only very simple logic can be used to achieve this as anything else simply is not evaluated properly to hide the parameters.

**Recommended solution** Allow the full-strength of the Modelica language be used to define the user-interface behaviour. This approach would suggest that the Modelica code be compiled in stages. The

first stage would compile the user-interface code. Once the user has adjusted the model parameters, the model equations can be compiled.

#### 5.1.4.2 Query limitations

**Problem** The user interface has limited knowledge and can have limited dependence on the connections with its model. This may not be a particularly important problem, however it could prove useful in some cases. The example derives from the user interface provided in Solid Edge for defining constraints. In Solid Edge, when establishing for example a distance constraint, the user is asked to select the two geometric features involved. If the first geometric feature selected is ether a plane or a line, then the user will not be allowed to select nothing other than a point geometric feature for the second selected and the constraint will be respectively either a Plane-Point or Line-Point distance constraint. The parameters that need to be provided will also be adjusted accordingly. The user interface thus depends on the choice of connections made.

To have a similar user interaction in Modelica is not possible. The cardinality associated with a connector (a feature which is supposed to be deprecated in future versions of the language and Dymola) might allow us to determine whether a connection is made, but this is evaluated only when the equations are generated and thus useless in the step when we are entering the parameters values.

Instead, our closest implementation in Modelica is a model which implements all three types of constraints, provides all three connection types (plane, line and point) on one side of the model and the point connector on the other, requires the user to select the type of constraint and disables all connectors and equations which are not for that type. See the Connect relation in table 4.5.

**Recommended solution** Improve Modelica and Modelica tools to allow for fully programmable user-interfaces with reflection into the model as it is created.

#### 5.1.4.3 Constraint programming

**Problem** In Solid Edge, the tool will forbid the addition of a new constraint if it conflicts with any of the existing ones. In Modelica, a constraint conflict will only be detected after the equations are translated and compiled at which point there may be more than one conflicting constraint which would make it difficult to determine which one is at fault.

**Recommended solution** Augment the Modelica language with an Object Constraint Language (OCL) that can control the model creation process by enforcing model level constraints.

#### 5.1.4.4 Mechanical assembly creation support

**Problem** When creating a mechanical assembly in MCAD tools, the user is always aware of the relative positions of all parts. However, in many BD modelling tools with multi-body dynamics support, the user cannot visualize the relative positions of the mechanical until after compilation. This is the case with Dymola. However, some other tools such as SimulationX, MapleSim, 20-Sim all provide a constantly updated 3d view as the model is being assembled providing instant feedback to the user.

**Recommended solution** As this is a tool specific limitation, the recommendation is to improve Dymola with this new feature.

## 5.2 MCAD limitations

We suggest several additions or improvements to MCAD tools in order to facilitate bi-directional model mapping with BD models.

### 5.2.1 Support for non-mechanical modelling

**Problem** The MCAD tool Solid Edge which we used is mainly geared towards mechanical modelling. We made use of its model annotation features to extend its use to other domains. For example, we annotated geometric features to define hydraulic or electrical connection points. However, it was clear that we were bending the tool to uses that were not intended.

**Recommended solution** Therefore, one important recommendation is to augment MCAD tools to support the modelling of non-mechanical systems as described in section 4.5.

### 5.2.2 Merging MCAD and block-diagram modelling

**Problem** The separation of the MCAD and BD modelling features between two tools makes the process of maintaining synchronisation difficult.

**Recommended solution** One interesting approach to easing the mapping problems would be to merge both the Modelica and MCAD tools and models into a single tool. Operations on models can be harmonized and therefore they would act simultaneously on both the MCAD model and Modelica code. Some of those ideas have already been implemented in CATIA. Such a tool would offer the user both BD as well as CAD modelling capabilities. Modelica models would be associated with MCAD Parts and Assemblies and the user would be capable of accessing the associate code from the same user interface. Geometric connectors and connections can be mapped to BD connectors and connections as we described. CrUD operations will be harmonized. With this approach, the problem we had in selecting the correct geometric features in Modelica would also be eliminated as we could directly access the geometric features using the MCAD 3d capabilities. In the same step, the creation of the mechanical assembly would be carried out using the tool most appropriate for that purpose (MCAD).

## 5.3 Future Work

### 5.3.1 Generalised model mapping

Due to time constraints we did not implement an automated mapping from the MBS to MCAD although our explanations through examples show that this is feasible and we provided many details. As a further step, our future work should make use of models to define the rules of transformation from/to either model. This will allow more flexibility with the appropriate modelling formalism. Triple graph grammars is an interesting candidate in this respect [72].

### 5.3.2 Improved constraint mapping

Another limitation of our implementation is that multiple constraints can lead to redundant equation problems in Modelica. Mechanisms need to be developed to detect and remove these redundancies between the equations are processed. This problem is solved in [70] and would be the subject of future work to elaborate on the details.

Another weakness of the current constraint mapping implementation is that each Solid Edge constraint maps to a corresponding constraint in Modelica. This is a very simple implementation. An improvement would be to map only the resultant constraint. For example, if we specify three perpendicular planar constraints in Solid Edge then the two parts would effectively have a rigid connection with no DOF remaining. We can then map this to a single rigid joint in Modelica (a fixed coordinate transformation) instead of three planar joints. Another improvement would be to create Modelica joints that can adjust their DOF through parameters. Therefore, the addition or removal of joints in Solid Edge would correspond to a change in parameter values in this parametrised joint in Modelica.

### 5.3.3 Improved user-interfaces and model transformations

If we could improve both the CAD and Modelica tools as well as the Modelica language, then by eliminating the limitations we described above, much would have been done to improve the user experience in keeping CAD and Modelica models synchronised. Also, the transformations can be applied constantly to keep the models synchronised. This however would require much more work as many more details than addressed in this thesis need to be worked out first.

# Bibliography


[1] R. E. Douglas Jr., "SNEAKERS: A concurrent engineering demonstration system," MSc dissertation, Worchester Polytechnic Institute, 1998.

[2] Y. V. R. Reddy, K. Srinivas, V. Jagannathan, and R. Karinthi, "Computer support for concurrent engineering - guest editors' introduction.," **I**EEE Computer, vol. 26, no. 1, pp. 12–16, 1993.

[3] "ANSYS®." Website. http://www.ansys.com.

[4] D. Zill, **A First Course in Differential Equations: With Modeling Applications**. Cengage Learning, 2008.

[5] F. Hildebrand, **Introduction to numerical analysis**. Dover Books on Advanced Mathematics, Dover Publications, 1987.

[6] Wolfram Mathematica Documentation Center, "Introduction to Differential-Algebraic Equations (DAEs)," 2011. http://reference.wolfram.com/mathematica/tutorial/DSolveIntroductionToDAEs.html.

[7] B. Zupancic, R. Karba, M. Atanasijevic-Kunc, and J. Music, "Continuous systems modelling education - causal or acausal approach?," in **I**nformation Technology Interfaces, 2008. ITI 2008. 30th International Conference on, pp. 803–808, June 2008.

[8] B. Denckla, P. J. Mosterman, and H. Vangheluwe, "Towards an executable denotational semantics for causal block diagrams," 2005.

[9] J. L. Peterson, "Petri net theory and the modeling of systems," 1981.

[10] S. Lecturer, K. B. D. Greene, J. W. Forrester, and J. W. Forrester, "System dynamics and the lessons of 35 years," 1993.

[11] H. Elmqvist, **A** Structured Model Language for Large Continuous Systems. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[12] "NetLib: ODE and DAE mathematical functions." Website. http://www.netlib.org/ode.

[13] "ACSL." Website. http://www.acslsim.com.

[14] "GAUSS." Website. http://www.aptech.com/gauss.html.

[15] "O-MATRIX." http://www.omatrix.com.

[16] "MATLAB." Website. http://www.mathworks.com.

[17] "Macsyma." Website. http://en.wikipedia.org/wiki/Macsyma.

[18] "Mathematica." Website. http://www.wolfram.com.

[19] "Maple." Website. http://www.maplesoft.com.

[20] "MuPad." Website. http://www.mathworks.com/products/symbolic/description4.html.

[21] "REDUCE." Website. http://reduce-algebra.sourceforge.net.

[22] L. Dragan and S. M. Watt, "On the performance of parametric polymorphism in maple.," **K**otsireas, Ilias (ed.) et al., Maple conference 2006. Proceedings of the conference, Waterloo, Ontario, Canada, July 23-26, 2006. Waterloo: Maplesoft (ISBN 1-897310-13-7/pbk). 35-42 (2006)., 2006.

[23] Anonymous, "Maple V package for field theoretic Poisson brackets using object-oriented techniques," vol. 96, pp. 209–216, Aug. 1996.

[24] A. Kugi, K. Schlacher, and M. Kaltenbacher, "Object oriented approach for large circuits with substructures in the computer algebra program Maple V," in **P**roceedings of the 1996 3rd International Conference on Software for Electrical Engineering Analysis and Design, ELECTROSOFT'96, pp. 491–500, 1996.

[25] "Objectica." Website. http://www.objectica.net/.

[26] "ObjectMath Home Page." Website. http://www.ida.liu.se/labs/pelab/omath.

[27] "Simscape." Website. http://www.mathworks.com/products/simscape.

[28] "Simulink®." Website. http://www.mathworks.com/products/simulink.

[29] "20-Sim." Website. http://www.20sim.com.

[30] "AMESim." Website. http://www.lmsintl.com.

[31] "Dymola." Website. http://www.3ds.com/products/catia/portfolio/dymola.

[32] "MapleSim." Website. http://www.maplesoft.com/products/maplesim/index.aspx.

[33] "SimulationX." Website. http://www.iti.de/cms/en/simulationx/about-simulationx/.

[34] "EASY5." Website. http://www.mscsoftware.com/Contents/Products/CAE-Tools/Easy5.aspx.

[35] "EcosimPro." Website. http://www.ecosimpro.com/.

[36] S. Leibbrandt, "Object-oriented modeling with objectica," in **2**007 Wolfram Technology Conference, 2007.

[37] "MathModelica." Website. http://www.mathcore.com.

[38] L. Viklund and P. Fritzson, "Objectmath—an object-oriented language and environment for symbolic and numerical processing in scientific computing," **S**ci. Program., vol. 4, no. 4, pp. 229–250, 1995.

[39] "Modelica Standard Library." Website. http://www.modelica.org/libraries/Modelica/.

[40] "Modelica Home Page." Website. http://www.modelica.org.

[41] H. Elmqvist, D. Brück, and M. Otter, **D**ymola — User's Manual. Dynasim AB, Research Park Ideon,Lund, Sweden, 1996.

[42] M. Andersson, **O**bject-Oriented Modeling and Simulation of Hybrid Systems. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994. PhD thesis ISRN LUTFD2/TFRT–1043–SE.

[43] H. Elmqvist, S. E. Mattson, and M. Otter, "Modelica - a language for physical system modeling, visualization and interaction," in **I**EEE Symposium on Computer-Aided Control System Design (CACSD), pp. 630–639, August 1999.

[44] S. E. Mattsson, H. Elmqvist, and D. Ab, "Modelica - an international effort to design the next generation modeling language," Master's thesis, 1997.

[45] P. Fritzson and V. Engelson, "Modelica – a unified object-oriented language for system modeling and simulation," pp. 67–90, 1998.

[46] P. Fritzson, **P**rinciples of object-oriented modeling and simulation with Modelica 2.1. John Wiley & Sons, 2004.

[47] "OpenModelica Project." Website. http://www.ida.liu.se/~pelab/modelica/OpenModelica.html.

[48] "Scicos." Website. http://www-roc.inria.fr/scicos/.

[49] "ADAMS®." Website. http://www.mscsoftware.com.

[50] **I**nvestigation of Drive Systems using ADAMS and MATLAB/Simulink, 2000. Haarlem, The Netherlands.

[51] "Saber®." Website. http://www.synopsys.com/saber.

[52] V. Engelson, H. Larsson, and P. Fritzson, "A design, simulation and visualization environment for object-oriented mechanical and multi-domain models in Modelica," in **P**roceedings of the IEEE International Conference on Information Visualization, pp. 188 – 193, 1999.

[53] "A Unified Object-Oriented Language for Physical Systems Modeling." Website, since 1997. http://www.modelica.org.

[54] "SolidWorks® Dassault Systèmes." Website. http://www.solidworks.com.

[55] "SimMechanics®." Website. http://www.mathworks.com/products/simmechanics.

[56] R. Sinha, C. J. Paredis, and P. K. Khosla, "Integration of mechanical CAD and behavioral modeling," in **P**roceedings of the IEEE/ACM international workshop on Behavioral Modeling and Simulation, pp. 31 – 36, 2000.

[57] R. Sinha, S. K. Gupta, C. J. J. Paredis, and P. K. Khosla, "Extracting articulation models from cad models of parts with curved surfaces," **J**ournal of Mechanical Design, vol. 124, no. 1, pp. 106–114, 2002.

[58] R. Sinha, S. K. Gupta, C. J. J. Paredis, and P. K. Khosla, "Capturing articulation in assemblies from component geometry," in **P**roceedings of the 1998 ASME Design Engineering Technical Conference, pp. 13–17, 1998.

[59] R. S. Christiaan, C. J. J. Paredis, and P. K. Khosla, "Kinematics support for design and simulation of mechatronic systems," in **T**he Fourth IFIP Working Group 5.2 Workshop on Knowledge Intensive CAD (KIC-4, pp. 246–258, 2000.

[60] T. Fernando, N. Murray, K. Tan, and P. Wimalaratne, "Software architecture for a constraint-based virtual environment," in **V**RST '99: Proceedings of the ACM symposium on Virtual reality software and technology, (New York, NY, USA), pp. 147–154, ACM, 1999.

[61] IEEE International Symposium on Assembly and Task Planning, **A**ssembly representations for capturing mating constraints and component kinematics, Aug 1997.

[62] V. Engelson, "Tools for design, interactive simulation and visualization for dynamic analysis of mechanical models," in **L**inkoping Electronic Articles in Computer and Information Science, ISSN 1401-9841, 2000.

[63] "LinkageDesigner." Website. http://www.linkagedesigner.com/Product.htm.

[64] "LMS Virtual.Lab Motion." Website. http://www.lmsintl.com/simulation/virtuallab/motion.

[65] M. Otter, H. Elmqvist, and S. E. Mattsson, "The new modelica multibody library," in **P**roceedings of the 3rd International Modelica Conference (P. Fritzson, ed.), (Linköpings Universitet, Linköping, Sweden), pp. 311–330, The Modelica Association The Modelica Association and Institutionen för datavetenskap, Linköpings universitet, Linköping : The Modelica Association and Linköping University, November 3-4, Nov 2003.

[66] X. Weigao, "The design and implementation of the îmodelica compiler," Master's thesis, School of Computer Science, Montreal, Canada, 2005.

[67] R. Smith, "Open Dynamics Engine." Website, 23 February 2006. http://www.ode.org.

[68] "SIMPACK." Website. http://www.simpack.com/products.html.

[69] D. T. Banach, T. Jones, and A. J. Kalameja, **A**utodesk Inventor 2010 Essentials Plus. 2010.

[70] H. Elmqvist, S. E. Mattsson, and C. Chapuis, "Redundancies in multibody systems and automatic coupling of catia and modelica," **P**roceedings of the 7th International Modelica Conference, Como, Italy, pp. 551–560, 20-22 September 2009.

[71] "StereoLithography Interface Specification." Website, 1989. http://www.ennex.com/~fabbers/StL.asp.

[72] A. Königs, "Model Transformation with Triple Graph Grammars," in **Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica**, 2005.

[73] Jan Van der Spiegel, "Spice - a brief overview." Website. http://howard.engr.siu.edu/elec/faculty/etienne/spice.overview.html.

[74] P. Fritzson, L. Viklund, J. Herber, and D. Fritzson, "High-level mathematical modeling and programming," **I**EEE Software, vol. 12, pp. 77–87, 1995.

[75] H. Elmqvist, D. Brück, and M. Otter, "Dymola — user's manual," 1996.

[76] M. Andersson, **O**bject-Oriented Modelling and Simulation of Hybrid Systems. PhD thesis, PhD thesis ISRN LUTFD2/TFRT–1043–SE, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[77] P. Sahlin, A. Bring, and E. Sowell, "The neutral model format for building simulation, version 3.02 technical report," tech. rep., Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.

[78] A. Jeandel, F. Boudaud, and E. Larivière, **A**LLAN Simulation release 3.1 description. M.DéGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, FRANCE, 1997.

[79] T. Ernst, S. Jahnichen, M. Klose, and R. Chaussee, "The architecture of the smile/M simulation environment," Sept. 12 1997.

[80] M. Abadi and L. Cardelli, **A** Theory of Objects. Springer-Verlag, 1996.

# Glossary

**ACSL** Advanced Continuous Simulation Language. 169

**AutoDesk Inventor** Mechanical Computer Assisted Design tool by AutoDesk. 52

**Mass properties** total mass, center of gravity and inertia matrix. 11

**Mechatronics** systems combining mechanics and electronics. 11

**Solid Edge** Mechanical Computer Assisted Design tool by Siemens. 8, 10, 51, 52, 55, 77, 78, 80, 82, 84–88, 91, 93, 96–104, 115, 118, 120–122, 124, 125, 127–139, 141, 143, 145

**SolidWorks** Mechanical Computer Assisted Design tool by Dassault. 34, 52, 55, 71, 80, 84

# Acronyms

**3D** 3-dimensional. 51, 54

**ACSL** Advanced Continuous Simulation Language. 26, 167

**API** Application Programming Interface. 78

**BD** Block Diagram. 38, 73, 82, 85, 91, 92

**CAD** Computer Assisted Design. 11

**CAS** Computer Algebra Systems. 26, 29, 30

**CE** Concurrent Engineering. 7, 8

**CrUD** Create/Update/Delete. 76, 77

**DAE** Differential Algebraic Equations. 15, 16, 18, 19, 23, 25, 32, 75

**DE** Differential Equations. 4, 15–18, 23, 27

**DoD** United States Department of Defense. 15

**DOF** Degrees of Freedom. 55

**FEA** Finite Element Analysis. 8, 9, 52, 71, 72

**FEM** Finite Element Model. 71–73

**MBS** Multi-Body Systems. i, 3, 4, 15, 33, 37, 69–71, 73, 74, 77, 78, 82–84, 88, 90, 91, 93, 94, 120, 128, 145

**MCAD** Mechanical Computer Assisted Design. i, 3, 4, 8, 9, 33–35, 38, 51, 52, 54, 55, 62, 69–74, 78–88, 90–98, 104, 128, 145

**ODE** Ordinary Differential Equations. 16–19, 32

**OOP** Object Oriented Programming. 29

**STL** stereolithography. 49, 62

**UCS** User-Coordinate Systems. 41

# Appendices

# A

# Modelica - A Unified Object-Oriented Language for System Modelling and Simulation

Peter Fritzson and Vadim Engelson
PELAB, Dept. of Computer and Information Science,
Linkping University, S-58183, Linkping, Sweden

A new language called Modelica for hierarchical physical modeling is developed through an international effort. Modelica 1.0 [http://www.modelica.org] was announced in September 1997. It is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages.

Compared with the widespread simulation languages available today this language offers three important advances:

1. *non-causal* modeling based on differential and algebraic equations;
2. *nonmultidomain* modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model;
3. a general type system that unifies object-orientation, multiple inheritance, and templates within a single *nonclass* construct.

A class in Modelica may contain variables (i.e. instances of other classes), equations and local class definitions. A function (method) can be regarded as a special case of local class without equations, but including an algorithm section.

The equation-based non-causal modeling makes Modelica classes more reusable than classes in ordinary object-oriented languages. The reason is that the class adapts itself to the data flow context where it is instantiated and connected. The multi-domain capability is partly based on a notion of *connectors*, i.e. certain class members that can act as interfaces (ports) when connecting instantiated objects. Connectors themselves are classes just like any other entity in Modelica.

Simulation models can be developed using a graphical editor for connection diagrams. Connections are established just by drawing lines between objects picked from a class library.

The Modelica semantics is defined via translation of classes, instances and connections into a flat set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by calling a symbolic and/or numeric solver. The generated C/C++ code is quite efficient.

In this paper we present the Modelica language with emphasis on its class construct and type system. A few short examples are given for illustration and compared with similar constructs in C++ and Java when this is relevant.

## A.1 Introduction

### A.1.1 Requirements for a modeling and simulation language

The use of computer simulation in industry is rapidly increasing. This is typically used to optimize products and to reduce product development cost and time. Whereas in the past it was considered sufficient to simulate subsystems separately, the current trend is to simulate increasingly complex physical systems composed of subsystems from multiple domains such as mechanic, electric, hydraulic, thermodynamic, and control system components.

### A.1.2 Background

Many commercial simulation software packages are available. The market is divided into distinct domains, such as packages based on block diagrams (SIMULINK[28], System Build, ACSL[13]), electronic programs (SPICE[73], Saber), multibody systems (ADAMS[49], DADS, SIMPACK), and others. With very few exceptions, all simulation packages are strong only in one domain and are not capable of modeling components from other domains in a reasonable way. However, this is a prerequisite to be able to simulate modern products that integrate, e.g., electric, mechanic, hydraulic and control components. Techniques for general purpose physical modeling have been developed some decades ago, but did not receive much attention from the simulation market due to lacking computer power at that time.

To summarize, we currently have three following problems:

- High performance simulation of complex multi-domain systems is needed. Current widespread methods cannot cope with serious multi-domain modeling and simulation.
- Simulated systems are increasingly complex. Thus, system modeling has to be based primarily on combining reusable components. A better technology is needed in creating easy-to-use reusable components.
- It is hard to achieve truly reusable components in object-oriented programming and modeling

### A.1.3 Proposed solution

The goal of the Modelica project[40] is to provide practically usable solutions to these problems, based on techniques for mathematical modeling of reusable components.

Several first generation object-oriented mathematical modeling languages and simulation systems (ObjectMath [74, 26], Dymola [75], Omola [76], NMF [77], gPROMS [26], Allan [78], Smile [79] etc.) have been developed during the past few years. These languages were applied in areas such as robotics, vehicles, thermal power plants, nuclear power plants, airplane simulation, real-time simulation of gear boxes, etc.

Several applications have shown, that object-oriented modeling techniques is not only comparable to, but outperform special purpose tools on applications that are far beyond the capacity of established block-oriented simulation tools.

However, the situation of a number of different incompatible object-oriented modeling and simulation languages was not satisfactory. Therefore in the fall of 1996 a group of researchers (see Section A.2.11) from universities and industry started work towards standardization and making this object-oriented modeling technology widely available.

The new language was called Modelica and designed for modeling dynamic behavior of engineering systems, intended to become a *de facto* standard.

Modelica is superior to current technology mainly for the following reasons:

- Object-oriented modeling. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- Non-causal modeling. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation.
- Physical modeling of multiple domains. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks. For application engineers, such physical" components are particularly easy to combine into simulation models using a graphical editor.

### A.1.4 Modelica view of object-orientation

Traditional object-oriented languages like C++, Java and Simula support programming with operation on state. The state of the program includes variable values and object data. Number of objects changes dynamically. Smalltalk view of object orientation is sending messages between (dynamically) created objects. The Modelica approach is different. The Modelica language emphasizes *structured* mathematical modeling and uses structural benefits of object-orientation. A Modelica model is primarily a declarative mathematical description, which allows analysis and equational reasoning. For these reasons, dynamic object creation at runtime is usually not interesting from a mathematical modeling point of view. Therefore, this is not supported by the Modelica language.

To compensate this missing feature *arrays* are provided by Modelica. An array is a set of objects of equal type. The size of the set is determined once at runtime. This construct for example can be used to represent a set of similar rollers in a bearing, or a set of electrons around an atomic nucleus.

### A.1.5 Object-Oriented Mathematical Modeling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other software. One way to handle this complexity is to use object-oriented techniques. Wegner [75] defines the basic terminology of object-oriented programming:

- *Objects* are collections of operations that share a state. These operations are often called *methods*. The state is represented by *instance variables*, which are accessible only to the operation's of the object.
- *Classes* are templates from which objects can be created.
- *Inheritance* allows us to reuse the operations of a class when defining new classes. A subclass inherits the operations of its parent class and can add new operations and instance variables.

Note that Wegner's strict requirement regarding data encapsulation is not fulfilled by object oriented languages like Simula or C++, where non-local access to instance variables is allowed.

More important, while Wegner's definitions are suitable for describing the notions of object-oriented *programming*, they are too restrictive for the case of object-oriented *mathematical modeling*, where a class description may consist of a set of equations, which implicitly define the behavior of some class of physical objects or the relationships between objects. Functions should be side-effect free and regarded as mathematical functions rather than operations. Explicit operations on state can be completely absent, but can be present. Also, causality, i.e. which variables are regarded as input, and which ones are regarded as output, is usually not defined by such an equation-based model.

There are usually many possible choices of causality, but one must be selected before a system of equations is solved. If a system of such equations is solved symbolically, the equations are transformed into a form where some (state) variables are explicitly defined in terms of other (state) variables. If the solution process is numeric, it will compute new state variables from old variable values, and thus operate on the state variables. Below we define the basic terminology of *object-oriented mathematical modeling*:

- An *object* is a collection of variables, equations, functions and other definitions related to a common abstraction and may share a state. Such operations are often called methods. The state is represented by *instance variables*.
- *Classes* are templates from which objects or subclasses can be created.
- *Inheritance* allows us to reuse the equations, functions and definitions of a class when defining objects and new classes. A subclass inherits the definitions of its parent class and can add new equations, functions, instance variables and other definitions.

As previously mentioned, the primary reason to introduce object-oriented techniques in mathematical modeling is to reduce complexity. To explain these ideas we use some examples from the domain of electric circuits. When a mathematical description is designed, and it consists of hundreds of equations and formulae, for instance a model of a complex electrical system, structuring the model is highly advantageous.

## A.2 A Modelica overview

Modelica programs are built from *classes*. Like in other object-oriented languages, class contains variables, i.e. class attributes representing data. The main difference compared with traditional object-oriented languages is that instead of functions (methods) we use *equations* to specify behavior. Equations can be written explicitly, like $a = b$, or be inherited from other classes. Equations can also be specified by the *connect* statement. The statement $connect(v1, v2)$ expresses coupling between variables $v1$ and $v2$. These variables are called *connectors* and belong to the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

In the following sections we introduce some basic and distinctive syntactical and semantic features of Modelica, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants. Powerful parametrization capabilities (which are advanced features of Modelica) are discussed in Section A.2.10.

### A.2.1 Modelica model of an electric circuit

As an introduction to Modelica we will present a model of a simple electrical circuit as shown in Figure A.1.

The system can be broken into a set of connected electrical standard components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of such components are available in Modelica class libraries.

A declaration like one below specifies that $R1$ to be of class *Resistor* and sets the default value of the resistance, $R$, to 10.

```
Resistor R1(R=10);
```

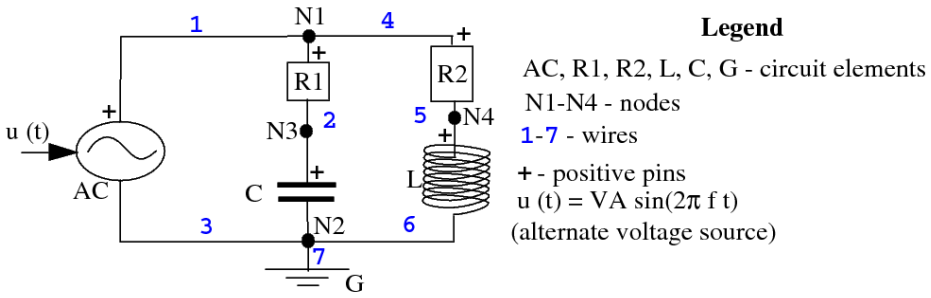



Figure A.1: A Connection diagram of the simple electric circuit example

.

A connection diagram of the simple electrical circuit example.

A Modelica description of the complete circuit appears as follows:

```
 class circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  Ground    G;

equation
  connect (AC.p, R1.p);     // Wire 1
  connect (R1.n, C.p);      // Wire 2
  connect (C.n,  AC.n);      // Wire 3
  connect (R1.p, R2.p);     // Wire 4
  connect (R2.n, L.p);      // Wire 5
  connect (L.n,  C.n);      // Wire 6
  connect (AC.n, G.p);      // Wire 7
end circuit;
```

A composite model like the circuit model described above specifies the system topology, i.e. the components and the connections between the components. The connections specify interactions between the components. In some previous object-oriented modeling languages connectors are referred to cuts, ports or terminals. The keyword *connect* is a special operator that generates equations taking into account what kind of interaction is involved as explained in Section 2.3.

Variables declared within classes are public by default, if they are not preceded by the keyword *protected* which has the same semantics as in Java. Additional *public* or *protected* sections can appear within a class, preceded by the corresponding keyword.

### A.2.2 Library classes

The next step in introducing Modelica is to explain how library model classes can be defined.

A connector must contain all quantities needed to describe an interaction. For electrical components we need the variables *voltage* and *current* to define interaction via a wire. The types to represent those can be declared as

```
class Voltage = Real;
class Current = Real;
```

where *Real* is the name of a predefined variable type. A real variable has a set of default attributes such as unit of measure, initial value, minimum and maximum value. These default attributes can be changed when declaring a new class, for example:

```
class Voltage = Real(unit="V", min=-220.0, max=220.0);
```

In Modelica, the basic structuring element is a *class*. There are seven restricted class categories with specific keywords, such as *type* (a class that is an extension of built-in classes, such as *Real*, or of other defined types) and *connector* (a class that does not have equations and can be used in connections). For a valid model replacing the *type* and *connector* keywords by the *class* keyword is fully equivalent, because the restrictions imposed by such a specialized class are fulfilled by a valid model. Other specific class categories are *model*, *package*, *record*, *function* and *record*.

The idea of restricted classes is advantageous because the modeler does not have to learn several different concepts, but just one: the class concept. All properties of a class, such as syntax and semantic of definition, instantiation, inheritance, generic properties are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified considerably because only the syntax and semantic of a class have to be implemented along with some additional checks on restricted classes. The basic types, such as *Real* or *Integer* are built-in type classes, i.e., they have all the properties of a class. The previous definitions can be expressed as follows using the keyword *type* which is equivalent to *class*, but limits the defined type to be extension of a built-in type, record or array.

```
type Voltage = Real;
type Current = Real;
```

### A.2.3 Connector classes

A connector class is defined as follows:

```
connector Pin
  Voltage      v;
  flow Current i;
end Pin;
```

Connection statements are used to connect instances of connection classes. A connection statement $connect(Pin1, Pin2)$, with $Pin1$ and $Pin2$ of connector class $Pin$, connects the two pins so that they form one node. This implies two equations, namely:

```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix *flow* is used. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems.

When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes. A common set of connector classes used in all components in the library supports compatibility of the component models.

### A.2.4 Virtual (partial) classes

A common property of many electrical components is that they have two pins. This means that it is useful to define an "interface" model class,

```
partial class TwoPin  "Superclass of elements
   with two electric pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

that has two pins, $p$ and $n$, a quantity, $v$, that defines the voltage drop across the component and a quantity, $i$, that defines the current into the pin $p$, through the component and out from the pin $n$ (Figure A.2)

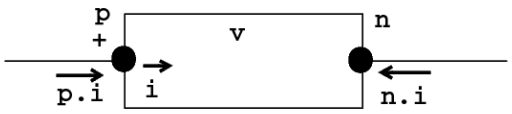



Figure A.2: Generic TwoPin Model

The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added. The keyword *partial* indicates that this model class is incomplete. The keyword is optional. It is meant as an indication to a user that it is not possible to use the class as it is to instantiate components. String after the class name is a comment.

### A.2.5 Equations and non-causal modeling

Non-causal modeling means modeling based on equations instead of assignment statements. Equations do not specify which variables are inputs and which are outputs, whereas in assignment statements variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equations-based models is unspecified and fixed only when the equation systems are solved. This is called non-causal modeling.

The main advantage with non-causal modeling is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The data flow context is defined by telling which variables are needed as *outputs* and which are external *inputs* to the simulated system.

The non-causality of Modelica library classes makes these more reusable than traditional classes containing assignment statements where the input-output causality is fixed.

For example the equation from resistor class below:

```
R*i = v;
```

can be used in two ways. The variable v can be computed as a function of i, or the variable i can be computed as a function of v as shown in the two assignment statements below:

```
i := v/R;
v := R*i;
```

In the same way the following equation from the class *TwoPin*

```
v = p.v - n.v
```

can be used in three ways:

```
v   := p.v - n.v;
p.v := v + n.v;
n.v := p.v - v;
```

### A.2.6 Inheritance, parameters and constants

To define a model for a resistor we exploit TwoPin and add a definition of a *parameter* for the resistance and Ohm's law to define the behavior:

```
class Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R(unit="Ohm") "Resistance";
equation
    R*i = v;
end Resistor;
```

The keyword *parameter* specifies that the variable is constant during a simulation run, but can change values between runs. This means that *parameter* is a special kind of constant, which is implemented as a static variable that is initialized once and never changes its value during a specific execution. A *parameter* is a variable that makes it simple for a user to modify the behavior of a model.

A Modelica *constant* never changes and can be substituted inline.

The keyword *extend*s specifies the parent class. All variables, *equation*s and *connect*s are inherited from the parent. Multiple inheritance is supported in Modelica.

Just like in C++ variables, equations and connections of the parent class cannot be removed in the subclass.

In C++ a virtual function can be replaced by a function with the same name in the child class. In Modelica 1.0 the equations cannot be named and therefore we cannot replace equations. When classes are inherited, equations are accumulated. This makes the equation-based semantics of the child classes consistent with the semantics of the parent class.

An innovation of Modelica is that type of a variable of the parent class can be replaced. We describe this in more detail in Section A.2.10.

### A.2.7 Time and model dynamics

Dynamic systems are models where behavior evolves as a function of time. We use a predefined variable time which steps forward during system simulation.

A class for the voltage source can be defined as:

```
class VsourceAC "Sin-wave voltage source"
  extends TwoPin;
  parameter Voltage VA = 220 "Amplitude";
  parameter Real f(unit="Hz") = 50  "Frequency";
  constant  Real PI=3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```

A class for an electrical capacitor can also reuse the *TwoPin* as follows:

```
class Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(unit="F") "Capacitance";
equation
    C*der(v) = i;
end Capacitor;
```

where $der(v)$ means the time derivative of v.

During system simulation the variables $i$ and $v$ evolve as functions of time. The solver of differential equations computes the values of $i(t)$ and $v(t)$ ($t$ is time) so that $Cv'(t) = i(t)$ for all values of $t$.

Finally, we define the ground point as a reference value for the voltage levels

```
class Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;
```

### A.2.8 Functions

Sometimes Modelica non-causal models have to be complemented by traditional procedural constructs like function calls. This is the case if a computation is more conveniently expressed in an algorithmic or procedural way. For example when computing the value of a polynomial form where the number of elements is unknown, as in the formula below:

$y = \sum_{i=1}^{size(a)} a_i.x^i$

Modelica allows a specialization of a class called *function*, which has only public inputs and outputs (these are marked in the code by keywords *input* and *output*), one *algorithm* section and no equations:

```
function PolynomialEvaluator
     input  Real a[:];// array, size defined at run time
     input  Real x;
     output Real y;
   protected
     Real   xpower;
   algorithm
     y := 0;
     xpower := 1;
     for i in 1:size(a, 1) loop
       y := y + a[i]*xpower;
       xpower := xpower*x;
     end for;
   end PolynomialEvaluator;
```

The Modelica function is side-effect free in the sense that it always returns the same outputs for the same input arguments. It can be invoked within expressions and equations, e.g. as below:

```
p = PolynomialEvaluator2(a=[1, 2, 3, 4], x=time);
```

More details on other Modelica constructs are presented in [40].

### A.2.9 The Modelica notion of subtypes

The notion of subtyping in Modelica is influenced by type theory of Abbadi and Cardelli [80]. The notion of inheritance in Modelica is separated from the notion of subtyping. According to the definition, a class A is a *subtype* of class B if class A contains all the public variables declared in the class B, and types of these variables are subtypes of types of corresponding variables in B. The main benefit of this definition is additional flexibility in the composition of types. For instance, the class *TempResistor* is a subtype of *Resistor*.

```
class TempResistor
  extends TwoPin
  parameter Real R, RT, Tref ;
  Real T;
equation
  v=i*(R+RT*(T-Tref));
end TempResistor
```

Subtyping is used for example in class *instantiation*, *redeclarations* and function calls. If variable $a$ is of type $A$, and $A$ is a subtype of $B$, then $a$ can be initialized by a variable of type $B$. Redeclaration is discussed in the next section.

Note that *TempResistor* does not inherit the *Resistor* class. There are different equations for evaluation of $v$. If equations are inherited from *Resistor* then the set of equations will become inconsistent

in *TempResistor*, since Modelica currently does not support named equations and replacement of equations. For example, the specialized equation below from *TempResistor*:

```
v=i*(R+RT*(T-Tref))
```

and the general equation from class *Resistor*

```
v=R*i
```

are inconsistent.

### A.2.10 Class parametrization

A distinctive feature of object-oriented programming languages and environments is ability to fetch classes from standard libraries and reuse them for particular needs. Obviously, this should be done without modification of the library codes. The two main mechanisms that serve for this purpose are:

- *inheritance*. It is essentially "copying" class definition and adding more elements (variables, equations and functions) to it.
- *class parametrization* (also called generic classes or types). It is replacing a generic type identifier in whole class definition by an actual type.

In Modelica we propose a new way to control class parametrization. Assume that a library class is defined as

```
class SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

Assume that in our particular application we would like to reuse the definition of *SimpleCircuit*: we want to use the parameter values given for $R1.R$ and $R2.R$ and the circuit topology, but exchange *Resistor* with the temperature-dependent resistor model, *TempResistor*, discussed above.

This can be accomplished by redeclaring $R1$ and $R2$ as follows.

```
class RefinedSimpleCircuit = SimpleCircuit(
  redeclare TempResistor R1,
  redeclare TempResistor R2);
```

Since *TempResistor* is a subtype of *Resistor*, it is possible to replace the ideal resistor model. Values of the additional parameters of *TempResistor* can be added in the redeclaration:

```
redeclare TempResistor R1(RT=0.1, Tref=20.0)
```

This is a very strong modification but it should be noted that all equations that could be defined in SimpleCircuit are still valid.

#### A.2.10.1 Comparison with C++

The C++ language is chosen as a representative of object-oriented language with static type system. We consider which complications arise if we attempt to reproduce Modelica class parametrization in C++.

We can define a template class in the library

```
class Resistor {
   public:
       float R;
};

template <class TResistor, class TResistor1>
// Several template arguments can be given here

class SimpleCircuit {
  public:
   SimpleCircuit(){ R1.R=100.0; R2.R=200.0; R3.R=300.0;  };
TResistor R1; // We should explicitly specify which two resistors will be replaced.

   TResistor1 R2;
   Resistor R3;
void func() {R3.R=R2.T;};
// Note: anything can be written here. The code is checked when it is instantiated only.
};
```

Code which reuses the library classes should look like

```
class TempResistor {
   public:
    float R,T,Tref,RT;
    };
class RefinedSimpleCircuit:public
SimpleCircuit<TempResistor,TempResistor> {
// Template parameters are passed
  RefinedSimpleCircuit(){ R1.RT=0.1; R1.Tref=20.0; }
...
};
```

To summarize we can reproduce the whole model in C++ but it is not possible to specify the SimpleCircuit class without specifying explicitly which data members (e.g. $R1$ and $R2$) are controlled by a type parameter such as $TResistor$. The C++ template construct requires this. Therfore the possible use of type parameters in C++ always has to be anticipated by making types explicit parameters of templates. In Modelica this generality is always available by default. Therefore C++ classes typically are less general and have lower degree of reusability compared to Modelica classes.

### A.2.10.2 Comparison with Java

Java is another object-oriected language with static type system. There are no options for generic classes. Instead we can use explicit type casting. The same approach can be used in C++, using pointers. However, type casting gives clumsy and less readable code.

**Example 1** If we permit *TempResistor* to be a subclass of *Resistor*, the code is straightforward:

```
class Resistor { public  double  R; };

class SimpleCircuit
{  public  SimpleCircuit() {
    R1=new Resistor();  R1.R=100.0;
    R2=new Resistor();  R2.R=200.0;
    R3=new Resistor();  R3.R=300.0;};
    Resistor R1, R2, R3;
    void  func(){R3.R=R1.R;};
 };

class TempResistor extends Resistor
{  public  double  T,Tref,RT; };

class RefinedSimpleCircuit extends SimpleCircuit
{ public
  RefinedSimpleCircuit() {
    R1=new TempResistor();  R2=new TempResistor();
// Type casting is necessary below:

    ((TempResistor)R1).RT=0.1;((TempResistor)R1).TRef=20.0;}
};
```

There is no way to initialize and work further with the variables $R1$ and $R2$ without type casting.

**Example 2** If we do not permit *TempResistor* to be a subclass of *Resistor*, the code is full with type casting operators:

```
class Resistor { public  double  R; };

class SimpleCircuit
{  public  SimpleCircuit() {
    R1=new Resistor();  ((Resistor)R1).R=100.0;
    R2=new Resistor();  ((Resistor)R2).R=200.0;
    R3=new Resistor();  ((Resistor)R3).R=300.0;};
    Object R1, R2, R3;
    void  func(){((Resistor)R3).R=((Resistor)R1).R;

// This causes exception if R1 has runtime type TempResistor.

  };
```

```
};

class TempResistor
{  public  double  R,T,Tref,TR; };
class RefinedSimpleCircuit extends SimpleCircuit
{ public
  RefinedSimpleCircuit() {
    R1=new TempResistor(); R2=new TempResistor();
// Type casting is necessary below

    ((TempResistor)R1).RT=0.1;((TempResistor)R1).TRef=20.0;}
};
```

The class *Object* is the only mechanism in Java that we can use for construction of generic classes. Since strong type control is enforced in Java, type cast operators are necessary for every access to $R1$, $R2$ and $R3$. Actually we remove type control from compilation time into the run time. This should be discouraged because it makes the code design more difficult and makes the program error-prone.

To summarize we can reproduce the whole model in Java and build an almost general library. However, many explicit class casting operations make the code difficult and non-natural.

#### A.2.10.3 Final components

The modeler of the *SimpleCircuit* can state that a component cannot be redeclared anymore. We declare such component as final.

```
final Resistor R3(R=300);
```

It is possible to state that a parameter is frozen to a certain value, i.e. is not a parameter anymore:

```
Resistor R3(final R=300);
```

#### A.2.10.4 Replaceable classes

To use another resistor model in the class *SimpleCircuit*, we needed to know that there were two replaceable resistors and we needed to know their names. To avoid this problem and prepare for replacement of a set of classes, one can define a replaceable class, $ResistorModel$. The actual class that will later be used for $R1$ and $R2$ must have $Pins$ $p$ and $n$ and a parameter $R$ in order to be compatible with how $R1$ and $R2$ are used within $SimpleCircuit2$. The replaceable model $ResistorModel$ is declared to be a Resistor model. This means that it will be enforced that the actual class will be a subtype of *Resistor*, i.e., have compatible connectors and parameters. Default for $ResistorModel$, i.e., when no actual redeclaration is made, is in this case *Resistor*. Note, that $R1$ and $R2$ are in this case of class $ResistorModel$.

```
class SimpleCircuit2
  replaceable class ResistorModel = Resistor;
protected
  ResistorModel R1(R=100), R2(R=200);
  final Resistor R3(final R=300);
```

```
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit2;
```

Binding an actual model *TempResistor* to the replaceable class *ResistorModel* is done as follows.

```
class RefinedSimpleCircuit2 =
  SimpleCircuit2(redeclare class ResistorModel = TempResistor);
```

This construction is similar to the C++ template construct. *ResistorModel* can serve as a type parameter. However, in C++ the type parameter cannot have default value. In Modelica the class *SimpleCircuit*2 is complete and can be used for variable instantiation. In C++ the class *SimpleCircuit*2 is a template, which must be instantiated first:

```
template <class ResistorModel>
class SimpleCircuit2 {
    ResistorModel R1(R=100);
...
}

class RefinedSimpleCircuit2 : public
   SimpleCircuit2<TempResistor>
    { ... }
```

### A.2.11 Acknowledgments

The Modelica definition has been developed by the Eurosim Modelica technical committee under the leadership of Hilding Elmqvist (Dynasim AB, Lund, Sweden). This group includes Fabrice Boudaud (Gaz de France), Jan Broenink (University of Twente, The Netherlands), Dag Brck (Dynasim AB, Lund, Sweden), Thilo Ernst (GMD-FIRST, Berlin, Germany), Peter Fritzson (Linkping University, Sweden), Alexandre Jeandel (Gaz de France), Kaj Juslin (VTT, Finland), Matthias Klose (Technical University of Berlin, Germany), Sven Erik Mattsson (Department of Automatic Control, Lund Institute of Technology, Sweden), Martin Otter (DLR Oberpfaffenhofen, Germany), Per Sahlin (BrisData AB, Stockholm, Sweden), Peter Schwarz (Fraunhofer Institute for Integrated Circuits, Dresden, Germany), Hubertus Tummescheit (GMD FIRST, Berlin, Germany) and Hans Vangheluwe (Department for Applied Mathematics, Biometrics and Process Control, University of Gent, Belgium). The work by Linkping University has been supported by the Wallenberg foundation as part of the WITAS project.