Linköping Studies in Science and Technology

Dissertation No. 627

Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing

by

Vadim Engelson

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2000

Abstract

Mathematical models used in scientific computing are becoming large and complex. In order to handle the size and complexity, the models should be better structured (using object-orientation) and visualized (using advanced user interfaces). Visualization is a difficult task, requiring a great deal of effort from scientific computing specialists.

Currently, the visualization of a model is tightly coupled with the structure of the model itself. This has the effect that any changes to the model require that the visualization be redesigned as well. Our vision is to automate the generation of visualizations from mathematical models. In other words, every time the model changes, its visualization is automatically updated without any programming efforts.

The innovation of this thesis is demonstrating this approach in a number of different situations, e.g. for input and output data, and for two- and three-dimensional visualizations. We show that this approach works best for object-oriented languages (ObjectMath, C++, and Modelica).

In the thesis, we describe the design of several programming environments and tools supporting the idea of automatic generation of visualizations.

Tools for two-dimensional visualization include an editor for class hierarchies and a tool that generates graphical user interfaces from data structures. The editor for class hierarchies has been designed for the ObjectMath language, an object-oriented extension of the Mathematica language, used for scientific computing. Diagrams showing inheritance, part-of relations, and instantiation of classes can be created, edited, or automatically generated from a model structure.

A graphical user interface, as well as routines for loading and saving data, can be automatically generated from class declarations in C++ or ObjectMath. This interface can be customized using scripts written in Tcl/Tk.

In three-dimensional visualization we use parametric surfaces defined by object-oriented mathematical models, as well as results from mechanical simulation of assemblies created by CAD tools.

Mathematica includes highly flexible tools for visualization of models, but their performance is not sufficient, since Mathematica is an interpreted language. We use a novel approach where Mathematica objects are translated to C++, and used both for simulation and for visualization of 3D scenes (including, in particular, plots of parametric functions).

Traditional solutions to simulations of CAD models are not customizable and the visualizations are not interactive. Mathematical models for mechanical multi-body simulation can be described in an object-oriented way in Modelica. However, the geometry, visual appearance, and assembly structure of mechanical systems are most conveniently designed using interactive CAD tools. Therefore we have developed a tool that automatically translates CAD models to visual representations and Modelica objects which are then simulated, and the results of the simulations are dynamically visualized. We have designed a high performance OpenGL-based

3D-visualization environment for assessing the models created in Modelica. These visualizations are interactive (simulation can be controlled by the user) and can be accessed via the Internet, using VRML or Cult3D technology. Two applications (helicopter flight and robot simulation) are discussed in detail.

The thesis also contains a section on integration of collision detection and collision response with Modelica models in order to enhance the realism of simulations and visualizations.

We compared several collision response approaches, and ultimately developed a new penalty-based collision response method, which we then integrated with the Modelica multi-body simulation library and a separate collision detection library.

We also present a new method to compress simulation results in order to reuse them for animations or further simulations. This method uses predictive coding and delivers high compression quality for results from ordinary differential equation solvers with varying time step.

Acknowledgments

First, I would like to thank my supervisor Peter Fritzson for letting me do this research and development of several very interesting projects. I had an enjoyable collaboration with the co-authors of the papers included in the thesis: Dag Fritzson, Johan Gunnarsson, Lars Viklund, and Håkan Larsson. The master students Håkan Larsson, Johan Parmar, Daniel Larsson, and Andreas Gustafsson worked hard on the projects discussed in the thesis. The Programming Environment Laboratory led by Peter Fritzson has been a stimulating and comfortable research environment. Many thanks to Peter Fritzson, Robert Forschheimer, Hilding Elmqvist, Dag Fritzson, Daniel Costello, Patrik Nordling, Peter Aronsson and Peter Bunus for their comments on a draft of the thesis and interesting discussions. Thanks also to Gunilla Norbäck, Bodil Mattsson-Kilhström, Kit Burmeister and Lillemor Wallgren for their administrative work, and Ivan Rankin for improving my English.

Finally, I would like to thank my wife Yelena for her help and patience, and my son Daniel who behaved very well when I wrote this thesis.

Vadim Engelson
Linköping
April 2000

Contents

Thesis Overview

1 Introduction

Mathematical models used in scientific computing are becoming large and complex. There are at least two areas of technology development, shown in Figure 1, aimed at handling the size and complexity problem. These are better model structuring methodologies (i.e. enhancing modeling and programming paradigms, such as object-oriented programming) and better methods to present complex systems (such as advanced user interfaces). These technologies make it possible to raise the level of abstraction. Raising the abstraction level helps humans to concentrate on the most important components and properties of complex models.

Scientific computing (in comparison with other software technology areas) leads to additional challenges such as large and complex data structures (multi-dimensional arrays and deeply nested object-oriented class structures), huge data volumes, a high level of abstraction used in modeling, and the need for advanced 2D and 3D graphics to present computation results.

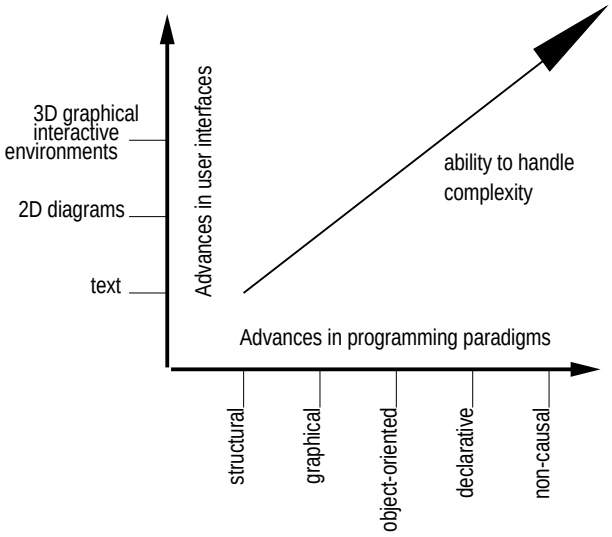


Figure 1: Advances in user interfaces and programming paradigms lead to increased ability to handle system complexity

The development and use of scientific computing software goes through several stages (see Figure 2), which can differ from application to application. Programming and simulation environments integrated with graphical user interfaces can help in different stages of software design and use. If the attention is focused at graphical user interfaces the most interesting stages are the following: design of data structures, input data preparation, execution and output data analysis. There are other stages, such as requirement specification, reengineering, maintenance, and debugging which might benefit from advanced graphical user interfaces, but we do not cover them in this work.

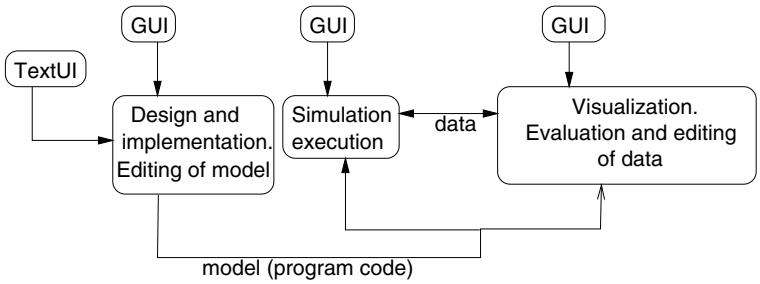


Figure 2: The development and use of software goes through several stages, some of which are depicted here. An appropriate user interface technology should be used in each phase.

1.1 Visualization and Editing Tools

We use the word visualization in a broad sense. Our interpretation is the representation of data structures and data values on computer displays by means of two- and three-dimensional graphical elements using an appropriate level of abstraction. The 3D visualization is a representation of three-dimensional scenes mapped onto a 2D display. Scientific visualization is a special case of visualization which usually means visual presentation of high volumes of numeric data defined over some continuous domain, such as time and/or space. Often computational results of scientific computing are displayed by scientific visualization tools (e.g. AVS[2], Data Explorer[16], and Vis5D[15]).

The information used in scientific computing falls into two categories: descriptions of mathematical models and descriptions of data. When a mathematical model (at some level of abstraction) is represented graphically as a diagram by some tool it is usually not called *visualization*, but rather *graphical model browsing* and/or *editing*.

There are many other kinds of information that are important in software design, which are not covered in this work. Particular examples are documents and document structures, entity-relation diagrams, scenario diagrams, database visualization and diagrams of program execution paths.

In the thesis we discuss and compare several stages where interactive graphical environments can be used:

- model editing (in particular, browsing and editing of model component diagrams during the design stage),
- input data editing,
- visualization during execution (e.g. interactive control of simulations, execution monitoring, and computational steering)
- output visualization in the form of 2D graphs and 3D interactive animations.

1.2 Models and Graphical User Interfaces

In a broad sense the goal of graphical user interfaces is to reflect the structure of the models and the data used (or produced) by the corresponding software.

For instance, a graphical user interface for model design and maintenance should be able to display the structure of the model. This structure contains model components (classes, variables, instances, functions, equations) and relations between the components. In graphical user interfaces which are used for model design each component is represented by a graphical element (vertex, icon, information box, etc.), and relations between the components are presented as links between these elements.

Graphical user interfaces for accessing model data should reflect the data structures. In most cases models have a tree-like organization of data. Certain larger components contain smaller components which in turn contain data items with elementary data types. A natural way to present data graphically is to use this tree-like structure for search and navigation. The leaves of the tree are represented by graphical elements (widgets) for user input and they are used for editing and inspection of the elementary data items. Usually not all data structure components used in the model should be available for editing and inspection by the end-user. Therefore some flexibility in the specification of graphical user interfaces for data is necessary.

In many application areas the model data contain some geometric information. Certain components of the model might have corresponding geometric shapes. Graphical user interfaces for such models should include display of the shape. In addition, the components may have some attributes and relations with other components. These relations and attributes are usually visualized in a graphical user interface as links between the shapes or visual properties of the shapes.

In all three of the above cases tight coupling between the model and graphical user interface exists.

Design and development of a graphical user interface for a mathematical model is a difficult task. User interfaces for scientific computing are often designed by scientific computing specialists, who lack expertise in user interface design. This leads to unnecessary waste of time, as well as error-prone and non-reusable visualization software.

In the above we have argued why the structure of a user interface is dependent on the structure of the corresponding model and model data. Each time the model changes, or model data structures are changed, the user interface should be partially redesigned. This leads to the need for additional design efforts. This disadvantage, however, can be turned into an advantage. Our vision is that graphical user interfaces can be *generated* from the corresponding models. In many cases this can be done automatically, without any design efforts. In other cases the design efforts needed for generating user interfaces automatically are negligible when compared with the efforts for traditional design and maintenance of user interface software. One of the challenging problems we attempt to solve in this thesis is to identify

cases when this automatic generation can be done, and to build tools that support this process.

Experience shows that the problem of automatic generation of user interfaces is quite application dependent. For instance, some representation of data is visualized automatically (e.g. as a tree like data structure diagram), but the applications may sometimes require a completely different view of the data (e.g. a plot containing certain data points). Mechanical models might use the same mathematical equations as electrical models, differing only in notation. However, a graphical user interface useful for mechanical simulation models differs very much from one for electrical simulations. Volume visualization or fluid dynamics visualization is not similar (in the general case) to the structure of equations used for the corresponding computations.

It appears that object-orientation would be helpful in automatic generation of user interfaces. We use a variety of object-oriented languages (ObjectMath, C++, Modelica) as the basis for our tools. Object-oriented models have a number of advantages, mainly for the following reasons:

- object-orientation imposes concise, hierarchical structures on models and data;
- information necessary for graphical user interface design can be extracted from such structures;
- object oriented languages provide the means to attach auxiliary attributes to existing structures. This can be done by specialization through inheritance. Then these attributes can be used for graphical user interface generation. Such attributes do not interfere with the data properties used for normal computation (e.g. simulation).

1.3 User Interaction

Static diagrams and graphics at an appropriate level of abstraction are useful tools, but are not enough for efficient work with complex software systems. In particular, dynamic presentations and interactivity are essential for three common activities – search (navigation), checking (analysis) and editing (modification). Users are expected to be active agents in the process of design and use of scientific software. Therefore user interfaces play a dominant role.

In descriptions of interactive tools we attempt to evaluate the quality of user interfaces. For this purpose typical user tasks are selected and user efforts required for performing these tasks are estimated. A good interface for visualization and editing should provide the following

- consistent and compact presentation of components (layout quality),
- ease of use in navigation and location of relevant subcomponents (navigation quality),

- consistent feedback from user actions, such as data editing or simulation steering (feedback quality).

1.4 Taxonomy of Visualizations

In order to discuss the variety of different visualizations used in scientific computing it is important to use a taxonomy. Such a taxonomy has been proposed by Shneiderman [28] for information visualization. We consider this taxonomy in the context of object oriented mathematical models used for scientific computing.

In this taxonomy seven kinds of visualizations are related to seven corresponding data types. Generation of some of visualizations in this taxonomy can be automated.

1-dimensional: This class of visualization and data structures includes textual documents and program source code. However, the models we consider are usually quite complex. Therefore, one-dimensional visualizations are not sufficient for presentation of such models. In particular, program source code has a well-defined structure. We suggest that this source code is converted into hierarchical structures with nodes (classes, objects, instances) and connections (relations between them). This structure is generated automatically from the textual representation and is visualized by our tools. For instance, ObjectMath models are represented by hierarchical diagrams, as described in **paper 2**.

2-dimensional: This kind of visualization includes geographical maps and plans, as well as 2-dimensional plots of functions (X-Y plots). This is a widespread way to present results from scientific computing applications. Most of the tools for scientific computing, e.g. Dymola[5], Mathematica[30], MathModelica (**paper 8**), Beast[10] have facilities for plotting variables in different ways and options for choosing variables to be plotted. The graphical user interface for selecting variables to be plotted is automatically generated from data structures, e.g. for Dymola, MathModelica and Beast.

3-dimensional: Items with volume, e.g. real world objects, and object designed using 3D CAD modeling tools are most naturally shown using 3-dimensional visualization. Both real and abstract objects (such as three dimensional plots of functions) can be viewed in this kind of visualization. Visualizations in three dimensions are often used for physics-based simulations. In this case, the structure of objects and their interrelations usually correspond to the structure of the mathematical model. Such a model contains descriptions of each physical component (e.g. rigid body) which is visualized as a separate graphical object in 3D. Therefore, creation of such visualizations can be automated (see **paper 5**). There are relations between the visualized components (contacts and various motion constraints), which usually are not

visualized explicitly as graphic elements, but which can easily be noticed when moving objects are observed.

There exist, however, 3-dimensional visualizations where the structure of graphical objects is completely different from the structure of mathematical model. For instance, this happens in scientific visualization tools used in computation fluid dynamics. Instead, the structure of graphical user interface for visualization control often corresponds to the structure and dimensionality of data.

Temporal: Visualization of time lines, historical information, and events are examples of temporal visualization. In our case, visualization of temporal data is just one feature for other visualization types, in particular 3-dimensional visualization. We use time in order to represent changes in objects and their relations during physics-based simulation. Output data of such simulations contains values of various variables at each time instant. Animation is used for presentation of such simulations.

Multi-dimensional: Tools working with objects with many attributes need multi-dimensional visualization. Such objects become points in n-dimensional space. Visualization tools map objects with these attributes to a 2- or 3-dimensional representation. In this thesis, work in this direction has been done with parametric functions of many parameters, which were defined in Mathematica. In our tool the way of mapping n-dimensional parametric functions to space coordinates and time can be selected interactively.

Tree: Tree-structured visualization is useful for structures with relations between parent and child nodes. A tree is a convenient way to represent data structures of a model. Furthermore, it is possible to automate creation of interactive visualizations based on data structures. This automation has been designed for C++, ObjectMath and MathModelica.

Networks and general graphs: Arbitrarily linked relations between nodes are conveniently visualized by networks and general graphs. This visualization is used where objects are related by connectors, for instance in Modelica[19] and Dymola[5]. In mechanical models joints and other contact elements are used as relations between rigid bodies. These relations can be generated using a graphical user interface. For instance, relations between bodies are specified using a CAD interface.

1.5 Thesis structure

This thesis work includes the investigation and development of several methods, languages and environments, which may appear somewhat diverse. However, all of these are based on a few common principles. The first of these principles is object orientation. All visualization tools, interactive environments and methods

developed in this work are based on object-oriented specification formalisms as input data. Target applications are complex, computationally intensive and can be considered as scientific computing applications. Advanced graphical user interfaces and dynamic 3D visualization are necessary in order to handle this level of complexity. A major theme in this work is that these graphical interfaces can be generated automatically or semi-automatically from object-oriented models.

The research presented in this thesis is very interdisciplinary, with all the studies, applications and discussions presented in the following chapters falling into the triangle between three basic aims of study:

- scientific computing,
- object-orientation
- 2D/3D graphical user interfaces.

This thesis is organized as follows: the thesis overview consists of an introduction and three other sections that are intended for general discussion and relating different research items and applications. After that, published papers, extended variants of published papers, and technical reports are included as chapters. These are devoted to particular problems or application areas. In the overview they are referred to as **paper 1**, **paper 2**, **paper 3**, etc.

Section 2 of the overview discusses model diagram editing and automatic generation of two-dimensional graphical user interface tools which are described in more detail in **paper 1**, **paper 2** and **paper 3**.

Section 3 of the overview discusses 3D visualization, about which more details can be found in **paper 4**, **paper 5**, **paper 6**, **paper 7** and **paper 9**. In the overview and in the papers we consider techniques for automatic generation of three-dimensional visualizations from object-oriented models.

Section 4 covers contributions to scientific computing which do not relate directly to visualization but rather to simulation and simulation environments. These are described in more detail in **paper 8**, **paper 10** and **paper 11**. Overviews of related work are given at the end of **paper 3**, in the first sections of **paper 5** and in several other papers.

2 Two-Dimensional Visualization

Two-dimensional visualization is a traditional method used in graphical user interfaces.

In this thesis two tools based on two-dimensional graphical interfaces are developed. Section 2.2 describes a graphical environment for editing class inheritance and composition diagrams for the object-oriented mathematical modeling language ObjectMath. The diagram components are mapped to corresponding syntactic constructs of the object-oriented modeling language. Section 2.3 describes

a universal environment for editing data values, automatically generated from an object-oriented model.

2.1 Graphical Tools for Editing Structure Diagrams of Software Systems

There is a wide spectrum of methods that have been designed in order to assist programmers in software design. There are also many approaches to using graphical tools in software design. These tools use some mapping between the program code structure and its graphical representation. Some specific methods are traditionally used for object-oriented programming. It is easier to visualize the structure of an object-oriented program than a conventional procedural program since there exist several language constructs (and relations between these) at a high level of abstraction. These are denoted differently in different object-oriented languages, but from an object-oriented methodology point of view these constructs are *classes* and *objects*, and relations are *inheritance*, *association* and *aggregation*.

2.2 Inheritance and Composition Diagrams and Their Use for the ObjectMath Object-Oriented Language.

During 1990-1993 a new object-oriented language for scientific computing called ObjectMath was developed at PELAB (Linköping University). The major innovation of this language is the introduction of object-oriented structure into a computer algebra language, making it possible to group equations, functions and formulae into classes. Such object-oriented mathematical models typically describe physical systems for the purpose of simulation. Such grouping is very useful for design of scientific computing applications. ObjectMath is based on the computer algebra language Mathematica. ObjectMath contains three object-oriented structuring constructs (*class*, *instance* and *part*), providing classes, single and multiple inheritance, instantiation and composition of parts. An integrated programming environment for ObjectMath has been developed. Our **paper 1** describes the motivation behind the ObjectMath design and gives an introduction to the language and the environment.

In many programming environments a mapping between object-oriented constructs and graphical elements is available. The Unified Modeling Language (UML) recently defined a standard for such mappings. However, traditional mapping rules existed a long time ago. In some programming environments (e.g. MicroSoft Visual C++) these diagrams cannot be edited directly and are just used for browsing. In other environments the diagrams can be edited and the editing operations are simultaneously applied to the designed program code. Different languages enforce different syntactic rules. Therefore the corresponding graphical environments have certain constraints that define how diagram components can be placed and attached to each other. Also there can be different strategies defining how the syntactic rules are enforced:

- all operations with the diagram that can lead to erroneous diagrams are explicitly forbidden by the tool;
- the tool includes a special feature for checking diagram correctness at any time during editing;
- the tool checks correctness when editing of a diagram is finished and it is saved or exported.

In our early work (done in 1994, some time before the UML standard appeared) we made the following contributions:

- formal definition of syntactic rules for relations between object-oriented constructs in ObjectMath;
- definition of a mapping between models in ObjectMath and class relationship (i.e. inheritance and composition) diagrams;
- definition of operations on relationships between object oriented constructs that can be performed by the user;
- definition and implementation of the above mentioned mapping and operations in a graphical class diagram editor.

There are two main properties which should hold for the set of defined operations:

Completeness: For any two syntactically valid class relationship diagrams A and B there exist a sequence of operations that transforms A to B.

Correctness: If a sequence of operations is applied to a syntactically correct diagram, the resulting diagram will be correct.

This work is described in detail in **paper 2**.

Code visualization for object-oriented modeling languages may be, for instance, object, and class relation diagrams that can be edited and browsed during design and maintenance of model specifications. There are context-dependent syntactical rules which should be observed in such diagrams. The challenge here is to map the rules into a user interaction process, in such a way that any modification of any syntactically correct diagram will always lead to a syntactically correct diagram.

The class diagrams are generated automatically from textual representations of object-oriented models. Graphical user interfaces are not generated from such models.

An editor has been designed for inheritance and composition relations between classes and instances of the object oriented language ObjectMath. The editor allows creation of classes and parts, specification of single and multiple inheritance

relations, replacing, collapsing, and deleting diagram fragments as well as supporting consistency checks between the current diagram contents and ObjectMath syntactic rules. Automatic component layout algorithms are employed for the display of directed graphs of relationships.

This work was done in 1994. The tool was developed in C++ using an object-oriented library for GUI programming called Interviews. A reduced variant of the tool supporting browsing only was developed in early 1997. This variant was based on C++ and another GUI design tool, Tcl/Tk. Since **paper 1** mentions very little about graphical design issues, a more detailed **paper 2** is included in the thesis.

2.3 Automatic Generation of Form-Based Interactive Environments from Object-Oriented Models

In this work object-oriented data structures are used for automatic generation of code for GUIs that support data input and editing. Such a GUI can serve as a default interface for programs written in object-oriented languages and requires no graphical programming at all.

There is usually an explicit or implicit association between the structure of a program and the data structures used in the program. The data structures should be created, filled with data values, used, and finally destroyed. Furthermore, the data has a nested structure. Therefore the program code for creation, filling in, using, and destruction has a similar nested structure.

The way software is structured usually matches the structure of corresponding components in the real world, i.e. in the application area. In order to provide consistent perception and comfortable navigation from the end-user point of view the structure of the graphical user interface should reflect the application.

For instance, the windows of a graphical user interface used for mechanical model description correspond to the bodies in the model. A graphical user interface for a flight ticket booking system corresponds to the structure of information written on tickets or other documents.

In our system the same object-oriented specification is used for three purposes:

- computation,
- generation of graphical user interfaces,
- generation of persistence routines, i.e. saving and loading application data.

When the code is generated, the graphical user interface and persistence routines can be compiled, linked and integrated with the rest of the application without any programming efforts.

Generation of persistence routines for C++ has been developed by Walter Tichy and Frances Paulisch in [26].

Our contribution is in development of display generation technique, automatic window layout, and user interface customization technique. Based on these principles we created the PDGen (Persistence and Display Generator) tool. This tool

automates the creation of graphical user interfaces. The software is intended for applications which require extensive editing of numerical data and complex data structures (arrays, trees etc.).

The tool was implemented in 1996. Conference **paper 3** gives the motivation, a history of the search for problem solutions, implementation details, and an overview of similar systems. The PDGen system is mainly described in Sections 1, 3, and 5 of this **paper**. Section 2 gives description of an old system; Section 4 discusses some specific issues of application of PDGen for ObjectMath.

More details about this system are given in [9].

3 Three dimensional graphical user interfaces

Development of modern technologies has made it possible to apply three-dimensional visualization in many application areas. In particular 3D is used for visualization of computation results and for modeling real world objects, such as objects constructed using CAD tools. Due to development of graphic hardware 3D animation recently became widely available for the users working on average computers and therefore 4D data (three space dimensions and one time dimension) can be used for visualization.

3D visualization in scientific computing falls into three categories:

- Visualization of numerical results, where displayed shapes depend on a particular computation. These shapes might depend on some specific parameters, e.g. time. We assume that points in 3D are denoted as (x, y, z). The set of displayed points in three-dimensional coordinate space can be expressed as
$$\{(F_x(u, v), F_y(u, v), F_z(u, v))\},$$
where $u_{min} < u < u_{max}$ and $v_{min} < v < v_{max}$. Some components of a graphical user interface for such visualization can be generated automatically. This approach is discussed in Section 3.1.

- Visualization of fixed shapes. Each shape corresponds to a real world object which is modeled as a rigid body (e.g. by a CAD tool). Movement of the body is constrained by the laws of physics. This kind of visualization is also called physics-based visualization. Visualization of results from physics-based simulations can be automated. This approach is discussed in Section 3.2.

- Volume visualization, where displayed shapes are isosurfaces computed from some volume data. This data can be the result of some other computations or measurements. The set of points displayed can be expressed as $\{(x, y, z)|F(x, y, z) = 0\}$. Rendering such visualizations is more difficult since it is hard to find the set of points and translate to 3D graphic primitives. Volume visualization is not in the scope of our work.

3.1 Interactive Visualization of Numerical Results of Computations Specified in Mathematica.

Assume that some numerical function (routine) with many input arguments and many output results is defined in the computer algebra language Mathematica. It is almost impossible to investigate the properties of this function by traditional methods, such as 2D plots. For instance, tens or hundreds of 2D plots are needed in order to find the maximal or minimal value of the function. It is very hard to find the area of non-continuity and observe how it changes when some of the function arguments change. The challenge is to automatically generate high performance 4D (three space and one time dimension) visualizations for this function.

Usually parametric surfaces are used for visualization of computational results when many inputs and outputs are involved in a computation. If two input and one output variables are used, the visualization of such a function is a surface composed by all points $\{(x, y, F(x, y))\}$, where $x_{min} < x < x_{max}$, $y_{min} < y < y_{max}$. If three input and three output variables are used, a dynamically changing surface can be composed from all the points $\{(F_x(u, v, t), F_y(u, v, t), F_z(u, v, t))\}$, where $u_{min} < u < u_{max}$, $v_{min} < v < v_{max}$, $t_{min} < t < t_{max}$.

In general, an arbitrary function $F : R^m \rightarrow R^n$ can be visualized using this method. The limitations are:

- Input values for a number of dimensions ($m-3$ dimensions) should be fixed.
- Three dimensions are chosen from n, whereas the other $n - 3$ dimensions are omitted.

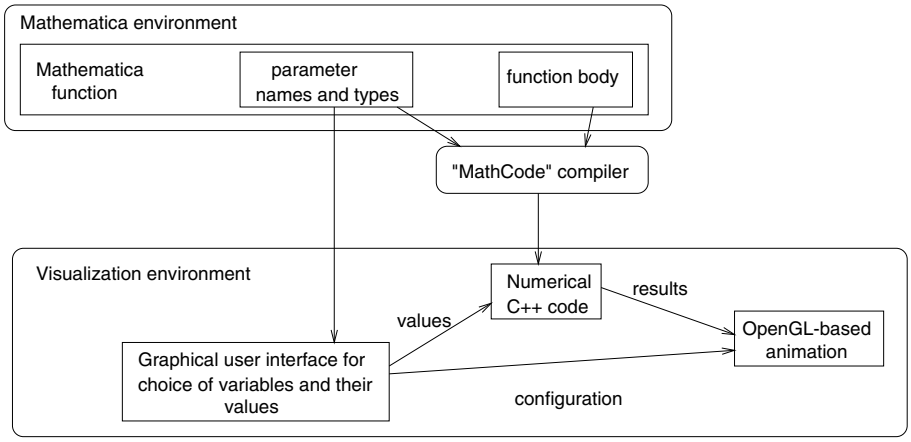


Figure 3: Generation of visualization for functions with multiple arguments and multiple output values defined in Mathematica.

The function F is initially specified symbolically in the Mathematica language (see Figure 3). This function is then compiled to C++. For this purpose a Mathematica to C++ compiler, MathCode[11, 12] is used. A fragment of C++ code is

generated which is compiled, linked with graphical user interface libraries and can be executed outside the Mathematica environment. In the case that Mathematica internal functions are used, this code can call Mathematica functions via MathLink.

The function F has a parameter list with names and types of the input and output parameters. A fragment of code used in the construction of the graphical user interface is generated from this information.

When the function F is visualized the user can interactively choose which three of the total m function arguments will be used as the surface parameters u and v, and the time parameter t. The rest of the $m-3$ arguments should be fixed when the visualization is running. However they can be freely adjusted interactively during the visualization using input controls (e.g. scale bars). Also three variables of a total of n output variables should be chosen, and these will be used as x,y, and z coordinates of the surface points.

In **paper 4** we give an example of a simulation function. The model describes two balls that fall into the water and the waves that appear on the surface of the water. The initial coordinates of the two balls are presented by a total of 6 scalar values. These values are not fixed in the application. Instead, scale bars are created for each of these 6 variables. The visualization function computes the points of water surface at each time instant. In this way a specialized environment where a user can manipulate input parameters and obtain different dynamic visualizations is automatically generated from a function.

Our contribution is development of this visualization environment, as well as participation in development of MathCode, compiler from Mathematica to C++.

3.2 A Modelica-Based Design, Simulation, and 3D Visualization Environment for Mechanical Models and its Applications

Modelica [19] is a new object-oriented equation-based modeling language. Models described in this language are well-structured collections of variables, ordinary differential and algebraic equations, and functions.

The methodology of object-oriented equation-based modeling have been invented by Hilding Elmqvist in his dissertation [4] and initially implemented in the Dymola language and tool [5]. This methodology has been further developed by the international Modelica Design Group [19] which includes participants from both universities and industry. The result of this development has been reflected in Modelica language specification [20]. Tutorial and rationale for the language can be found in [21]. Two papers written by Martin Otter, Hilding Elmqvist and Sven Erik Mattsson [7, 23] cover the major properties of Modelica and hybrid modeling technique invented by these authors. The hybrid modeling in Modelica integrates discrete and continuous modeling methods in the same language. Object-orientation techniques used in Modelica are discussed in [13]. Our current Modelica activities are reflected in [27].

Modelica models can be simulated. The result of simulation is values of all declared variables during certain time interval.

Currently there are two design, simulation, and visualization environments for Modelica:

- *Dymola tool with Modelica Language Support*, developed by Dynasim [5] and
- *MathModelica*, developed by MathCore[18].

Differences between these environments are discussed in Section 3.4.

In these environments Modelica models can be created using a graphical user interface. For the purpose of simulation starting conditions and parameters can be entered by the user, and computation occurs without further user interaction. Visualization tools in these environments have facilities for construction of simple 2D plots of variables, as well as 3D animations presenting motion of primitive objects.

One of the distinctive features of Modelica is the ability to describe multidomain applications. Mechanical, electrical, hydraulic and control components can be described in the same language within the same model and communicate with each other. In many application areas (robotics, automotive device modeling, etc.) mechanical components play the dominant role, whereas electrical, hydraulic, and control components can be considered as ”subordinated” to mechanics.

Our vision is that the existing environments can be made more efficient and flexible if the following conditions are fulfilled:

- Use of automatic code generation, and automatic creation of interactive simulations and visualizations;
- Support for specific application areas, e.g. simulation of mechanical models and multi-body systems in general.

Our goal, therefore, is to *automate generation of interactive mechanical models*. Obviously, the models cannot be created completely automatically; instead, user-friendly tools working on appropriate level of abstraction, accompanied by model translation tools should allow designing complex models much easier, faster and error-free. Our **paper 5** contains a detailed discussion on both code generation and visualization. It explains, in particular, why Modelica has been chosen as a tool for mechanical modeling. A comparison of Modelica with several other tools is given in the paper.

Our contribution aimed to achieve this goal has been development of the following techniques and tools:

- Technique and tool for automatic generation of Modelica models (and model components) from mechanical CAD models.
- Several tools for interactive visualization of mechanical models designed in Modelica and optionally derived from CAD models.

- A technique and an interface between multi-body simulations in Modelica and collision detection and response routines.

A short overview of these activities can be found in a published **paper 7** in 1999. Unfortunately, some important references and updated background information are missing in this paper. We wrote a more detailed description of our activities which is given in **paper 5**. A collision detection and response interface is discussed in Section 4.1.

For mechanical modeling we use one of the existing Modelica component libraries, the Multibody System (MBS) Library. This object-oriented library has been invented by Martin Otter in his dissertation [22]. Overviews of the library are given in [24, 25]. This library has been first implemented in the Dymola language. Later this library has been translated to Modelica. The equations in this library are based on Newton's laws (and their consequences) for systems of rigid bodies connected by rotating and translating joints. The joints specify degrees of freedom for the bodies. There exist other component libraries (e.g. electrical and hydraulic) which can be used in combination with the mechanical library. Currently, in 2000, a new MBS library for Modelica is being developed, and new compilation and simulation algorithms are being added into the Dymola tool, which solve [8] many technical problems discussed in **paper 5**.

In order to model mechanical systems, first the geometry of the rigid bodies and the constraints between the bodies of a system should be specified. The most convenient way to perform this task is using an interactive design environment. CAD tools were used for many years for this purpose. Therefore we use an existing CAD tool and integrate specific features we need into this tool. The SolidWorks system has been chosen for this purpose. Several other alternative tools are considered in **paper 5**. We designed a converter from SolidWorks to Modelica; its implementation has been done by Håkan Larsson and the author at PELAB[17]. This tool extracts physical and geometric information from SolidWorks assemblies as well as information about degrees of freedom which are defined between the objects.

The existing 3D visualization tool for Modelica, called DymoView, which is a component of the Dymola tool, is not intended for mechanical engineering applications. However the classes for 3D visualization, developed by Hilding Elmqvist and included in the Dymola and Modelica tool libraries, can be used for this purpose.

During 1998–1999 in addition to this environment we developed a set of 3D visualization environments:

- A high performance and portable OpenGL-based environment for interactive engineering visualization.
- A VRML-based system for visualization via the Internet (developed by Daniel Larsson and the author at PELAB[27]).
- A system for interactive visualization via the Internet, which uses high quality rendering technology, called Cult3D (developed by Andreas Gustafsson

and the author[14]).

- A tool for creation of non-interactive visualizations (animations) in 3DStudioMax.

In order to add interactivity to Modelica simulations, we have designed a Tcl/Tk-based graphical user interface. During simulation some variables can get new values specified by the user. The new values are propagated using the equations to the output values of the simulation and the user can get immediate visual feedback corresponding to his actions.

The unique feature of the environment we have developed is the combination of three properties:

- It will be possible to generate code for high performance physics-based simulations from CAD models.
- It will be possible to integrate components from many physical domains in the same application.
- It will be possible to control simulations interactively.

The components of the environment for mechanical design, simulation and interactive visualization are not yet integrated into a single system. However, separate components of this system have been successfully used in two major applications (they are described in detail in **paper 6**):

Modeling of helicopter flight mechanics. This application demonstrates object-oriented modeling of helicopter flight dynamics and the use of an external controller which steers the helicopter model so that certain flight commands are performed.

Modeling of service robot-manipulator. This application demonstrates the use of an internal controller that steers the virtual robot according to the plan. Our **paper 6** also contains our Modelica-based solution to the inverse geometry problem.

Future work includes extensions of the integrated environment tools in several directions:

- More details about the joints and related objects which drive the joint can be specified at an early stage, i.e. in the Modelica generator interface we integrated into SolidWorks.
- Visualization can be integrated with a CAD tool, so that the same familiar environment is used both for mechanical design and for interactive visualization of simulation results.
- Interfaces to other CAD tools, such as Mechanical Desktop, Pro Engineer, etc. are planned.

3.3 Integration of 3D Graphics and Modelica

Our **paper 5** (Section 8) presents a description of two ways of integrating 3D graphics and Modelica:

- Primitive 3D objects can be described. All the object parameters (predefined geometry, size, color, position, rotation) are specified as Modelica model variables and constants.
- Arbitrary 3D objects are described in external files. Their geometry and color is defined by external formats, in particular, DXF[3], Stereo Lithography Interface (STL)[1], and VRML[29]. Their position and rotation are specified as model variables.

However there is a need for an intermediate approach. It should be possible to store all information about the 3D graphics in the same Modelica model. This information can be later used for both visualization and for computation of collision response forces. The 3D geometry and graphical information can be described in two additional ways:

- Complex 3D geometry objects can be constructed from primitives, such as points, lines, triangles, and quadrilaterals. Each primitive can have certain color properties, which specify how it is rendered during visualization.
- Objects can be described by inline insertion of descriptions in a format external to Modelica, for instance STL or VRML.

In Modelica models these objects are represented syntactically as class attributes (formal comments) or as instances of classes from a special class library.

In addition to the rigid bodies, the sources of light as well as the positions and attributes of a virtual camera can be described in the same way as 3D geometrical objects.

This approach is described in **paper 9**.

3.4 MathModelica: A Modelica Environment Embedded in the Mathematica Environment

The computer algebra tool Mathematica[30] can serve two goals in the context of Modelica-related tools:

- As a powerful computer algebra language it can serve as an implementation language for Modelica. Modelica models can be translated to Mathematica and these models can be simulated as Mathematica functions.
- As a powerful documentation system it can serve as an integrated environment for design, documentation, and visualization of Modelica models.

The major advantage of using Mathematica as a basis for a Modelica integrated environment is the ability to unify source code of models, documentation, tests, simulation, computation results, and visualization in the same format. Mathematica notebooks can be used as a storage place for structured documents. A user can work in the familiar Mathematica front-end environment with these documents. Since the notebooks have a strictly defined structure, Mathematica programs can process this structure and use it for code compilation and as well as for result visualization.

Our **paper 8** discusses the initial variant of a MathModelica environment for this purpose. Our contribution is a prototype translator of equations from Modelica models to corresponding equations in Mathematica, as well as prototype generator of graphical user interface for these Modelica models.

It should be noted that the paper describes the MathModelica version of 1998. The current variant of the MathModelica language and environment (MathModelica-2000) is designed by MathCore [18]. The major differences between these two versions are described below:

- In MathModelica-1998 the model was converted to a form appropriate for Mathematica. The solution was obtained from the numerical solver of differential equation systems (`NDSolve`) available in Mathematica. Only ordinary differential equations can be solved this way.

- In MathModelica-2000 the model is converted to (standard) Modelica, and the same Modelica compiler, simulation code and numerical libraries as those used in the Dymola environment [5] is utilized for finding the solution.

There is one major difficulty in designing and simulating with Modelica. When a Modelica model is specified but not simulated yet, it is very hard to predict whether a solution exists and if it will be found. When the computation starts and the solver stops for some reason it is hard to find the cause of this error in the source model. The failure of computations does not mean that there is no solution, simply that the solution cannot be found by the currently used integration method and method parameters (step, accuracy, etc.). Therefore, it is extremely important for the user to know the reason for a failure at an early stage.

Unfortunately the exact definition of what `NDSolve` can do and what it cannot do is missing. The major technical difficulty with MathModelica-1998 is that we cannot predict the behavior of `NDSolve` and know whether a system of equations can or cannot be solved by this tool before it is applied. Furthermore, there is no support for discrete events, as well as differential-algebraic equations. Only a subset of Modelica can be implemented this way.

In MathModelica-2000 the diagnostic messages from the Modelica compiler and the from solver are analyzed and much more information about the reason for the failure is obtained. It can be noted that this is not the ultimate solution either.

It is clear that in general it is not possible to determine whether a differential-algebraic equation is solvable or not. But at least for some cases certain rules and heuristics can be developed. Our research activities in the future will be directed toward increasing robustness and prevention of failures:

- Finding ”robust” model components and rules of connections between them so that simulation of models correctly built from such components does not fail. One example of such component collection is the MBS library developed by Martin Otter (see Section 3.2), where components should be connected according to certain rules.
- Developing methods and tools to analyze a model before computation starts and find out possible reasons for failure as early as possible.

4 Contributions to Simulation Techniques and Environments

Mathematical models used in scientific computing are becoming large and the number of computations needed for simulation is increasing. Simultaneously there are growing requirements on model performance and accuracy. When simulation models are designed we encounter several problems that must be solved in order to achieve these goals. Section 4.1 describe work on achieving higher accuracy and realism for mechanical models. In section 4.2 we discuss a way to increase performance of data input and output performed by simulation tools using data compression. These topic are not related to visualization directly and no graphical interface generation happens here. However the methods discussed in this section can be used for increasing the performance and realism of visualizations.

4.1 Collision Detection and Response for Mechanical Simulations in Modelica.

Collision detection and response is one of the most complicated and important parts of mechanical analysis. There are many methods for computation of collision response; some major methods are discussed in **paper 10**. We consider collisions of rigid bodies. The surfaces of the bodies are defined by a set of triangles in 3D. It is assumed that a single point of one body collides with a single point of another body. In the paper we discuss an impulse-based and force-based approach to collision response. Both of these models can be implemented either in pure Modelica or as external functions called from Modelica. Using external functions is more convenient for complex shapes.

In the impulse-based approach, each collision generates an event. When the event happens, Modelica simulation stops, velocities of objects change and simulation is restarted with new re-initialized velocities. In the general case it is hard to find the new velocities if there are many bodies linked together by revolute joints.

In the force-based model, colliding surfaces are connected by a virtual stiff spring as soon as they penetrate or occur too close to each other. The collision force is created by the spring, separating the objects. We have derived a new collision force equation such that the overall result of a collision matches the result of the impulse-based collision model.

For the purpose of collision detection a public domain collision detection package called SOLID has been used. We have made an interface allowing this tool to read complex shape descriptions in STL[1] files. This format is already in use for presentation of geometry of the bodies.

In the future several complementary and alternative approaches should be considered:

- Collision detection of complex shapes can be performed by CAD tools. Since the shapes are originally defined in CAD, the corresponding application programming interface packages can detect the collision and find collision parameters.

- Currently the body surfaces are represented as a set of triangles. Every face of the body is a plane. Methods exist that make it possible to detect and process collisions between two points belonging to objects described by splines of second order. Therefore these splines can be used instead of plane triangles. The CAD tools, however, typically export geometry information as triangle sets, not spline surfaces.

Our **paper 10** describes the architecture for integration of Modelica and collision detection software. In the general case (i.e. for models with complex geometry) several external functions need to be called from the Modelica model. The collision detection software can be used in combination with MBS library in Modelica.

Currently a new, generalized multi-body system library and general impulse handling model is being developed by Martin Otter [8].

4.2 A Lossless Compression Method for Computational Results.

Complex mathematical models used for simulation generate huge amounts of numerical data. This data is used for visualization, for further processing of computation results, and as starting values for other simulations.

Traditional text and binary data compression techniques do not work well with numerical data, such as double precision floating point numbers. Some numerical method for compression should be applied. This method should take into account that numerical values typically change smoothly from one simulation step to another.

For the purpose of compression we developed and applied a specialized method of predictive coding, discussed in **paper 11**. A series of variable values are compressed by this method. This method assumes that each saved variable value can

be extrapolated using several values at the previous time steps, and the residual (difference between the actual and extrapolated value) is very small.

The method is intended for lossless compression, since it is used in an application where precision is extremely important [10]. A lossy variant of the method is described at the end of the paper. A comparison with other numerical compression methods (e.g. *wavelet* algorithms) is given in the paper. When the method is used in our application (Beast[10]) the lossless variant of the algorithm gives a compression ratio of 3 to 6 times, depending on the contents of the series of variable values.

5 Conclusion

In this thesis we emphasize the idea of automatic generation of design, simulation, and visualization tools. Most of the work described in the various papers are attempts to realize this idea. The reader can find conclusions and descriptions of future work regarding particular tools in each paper or report included in the thesis. The Modelica effort (see **paper 5**) is an ongoing project with many activities that enhance the language as well as design, simulation, and visualization environments for using this language.

In several cases we succeed in showing that various useful tools can be generated from object-oriented models. These models and environments can be used not only in the context of scientific computing, but in a broad spectrum of applications. We probably cover just a fraction of the diversity of design, simulation, and visualization environments. The new environments appear on top of old environments, like our environment for mechanical modeling appears on top of an existing Modelica environment. The future, in our opinion, belongs to easily configurable, multi-level environments where some levels require intellectual effort on the part of the designer (and therefore requires some manual programming), but some other levels can be generated automatically, without any efforts from the user. All these levels can be configured and utilized by the users.

List of papers

Paper 1 Peter Fritzson, Vadim Engelson, Lars Viklund, Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment, In *Proceedings of the Conference on Design and Implementation of Symbolic Computation Systems (DISCO 93)*, vol. 722 of Lecture Notes in Computer Science, pp. 145–160. Springer-Verlag, 1993.

Paper 2 Vadim Engelson, *ObjectMath Inheritance and Composition Diagram Editor*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 006. Available at:
http://www.ep.liu.se/ea/cis/2000/006/

Paper 3 Vadim Engelson, Peter Fritzson, Dag Fritzson, Automatic Generation of User Interfaces From Data Structure Specifications and Object-Oriented Application Models. In *Proceedings of European Conference on Object-Oriented Programming* (ECOOP96), Linz, Austria, 8–12 July 1996, vol. 1098 of Lecture Notes in Computer Science, Pierre Cointe (Ed.), pp. 114–141. Springer-Verlag, 1996

Paper 4 Vadim Engelson, Peter Fritzson, Dag Fritzson, Using the Mathematica Environment for Generating Efficient 3D Graphics. In *Proceedings of EduGraphics/ CompuGraphics-97*, Vilamoura, Portugal, December 15-17, 1997, pp. 222 – 231.

Paper 5 Vadim Engelson, *Tools for Design, Interactive Simulation, and Visualization for Dynamic Analysis of Mechanical Models*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 007. Available at: http://www.ep.liu.se/ea/cis/2000/007/

Paper 6 Vadim Engelson, *Simulation and Visualization of Autonomous Helicopter and Service Robots*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 013. Available at: http://www.ep.liu.se/ea/cis/2000/008/

Paper 7 Vadim Engelson, Håkan Larsson, Peter Fritzson, A Design, Simulation and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in Modelica. In *Proceedings of 1999 IEEE International Conference on Information Visualization*, IEEE Computer Society, 14-16 July 1999, London, pp. 188-193, ISBN 0-7695-0210-5.

Paper 8 Peter Fritzson, Vadim Engelson , Johan Gunnarsson, An Integrated Modelica Environment for Modeling, Documentation and Simulation. In *Proceedings of The 1998 Summer Computer Simulation Conference* (SCSC 98) July 19–22, 1998, Reno, Nevada, pp. 308-313.

Paper 9 Vadim Engelson, *Integration of Modelica and 3D Geometry.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 009. Available at: http://www.ep.liu.se/ea/cis/2000/009/

Paper 10 Vadim Engelson, *Integration of Collision Detection with Multibody System Library in Modelica.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 010. Available at: http://www.ep.liu.se/ea/cis/2000/010/

Paper 11 Vadim Engelson, Dag Fritzson, Peter Fritzson, *Lossless Compression of High-Volume Numerical Data for Simulations.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 011. Available at: http://www.ep.liu.se/ea/cis/2000/011/. An abstract of this paper is published in *Proceedings of the 2000 IEEE Data Compression Conference,* Snowbird, Utah, March 28-30, 2000.

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] Advanced Visual Systems Inc., *AVS/Express. Reference Manual.*, http://www.avs.com

[3] Autodesk, Inc., *Autocad 2000 documenation. Drawing Interchange File Format.*, http://www.autodesk.com

[4] Hilding Elmqvist, *A Structured Model Language for Large Continuous Systems,* PhD thesis TFRT-1015, Department of Automatic Control, Lund University of Technology, Lund, Sweden.

[5] Hilding Elmqvist, Dag Brück, Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual, Version 4.0*, from Dynasim AB, Research Park Ideon, Lund, Sweden, http://www.dynasim.se

[6] Hilding Elmqvist, Sven-Erik Mattsson. Modelica – The Next Generation Modeling Language – An International Design Effort. In *Proceedings of First World Congress of System Simulation*, Singapore, September 1–3 1997.

[7] Hilding Elmqvist, Sven Erik Mattsson and Martin Otter *Modelica - A Language for Physical System Modeling, Visualization and Interaction*. Plenary paper. 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Hawaii, August 22-27, 1999

[8] Hilding Elmqvist, *Personal communication*, April 2000.

[9] Vadim Engelson, *An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing*, Linköping Studies in Science and Technology, Licentiate thesis No 545, Department of Computer and Information Science, Linköping University, March 1996, 72 pp.

[10] Dag Fritzson, Peter Fritzson, Patrik Nordling, Tommy Persson. Rolling Bearing Simulation on MIMD Computers. *International Journal of Supercomputing Applications and High Performance Computing*, 11(4), 1997.

[11] Peter Fritzson, *MathCode C++*, Available from MathCore AB, http://www.mathcore.com

[12] Peter Fritzson, Static and Strong Typing for Extended Mathematica. *Innovation in Mathematica - Proceedings of the Second International Mathematica Symposium*, Rovaniemi, Finland, 29 June - 4 July 1997, Computational Mechanics Publications, V. Keränen, P. Mitic, A. Hietamäki (Ed.), pp. 153–160.

[13] Peter Fritzson, Vadim Engelson, Modelica – A Unified Object-Oriented Language for System Modeling and Simulation, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP98)*, Brussels, July 20–24, 1998.

[14] Andreas Gustavsson, *Integration of Cult3D and Modelica Simulations*, Master Thesis, IDA, Linköping University, Sweden, to be published in May 2000.

[15] Bill Hibbard, *Vis5D*, http://www.ssec.wisc.edu/ billh/

[16] IBM, *Open Visualization Data Explorer*, http://www.research.ibm.com/dx/

[17] Håkan Larsson, *Translation of 3D CAD Models to Modelica*, Master Thesis, LiTH-IDA-Ex-99/30, IDA, Linköping University, Sweden, March 1999.

[18] MathCore, *MathModelica, software tool for modeling, simulation, and visualization.* http://www.mathcore.com.

[19] Modelica Design Group, *Modelica WWW site*, http://www.modelica.org

[20] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[21] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial and Rationale. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[22] Martin Otter, *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. Dissertation, Fortschrittberichte VDI, Reihe 20, Nr. 147, 1995.

[23] Martin Otter, Hilding Elmqvist and Sven Erik Mattsson *Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle*, 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Hawaii, August 22-27, 1999

[24] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, in *Proceedings of the NATO-Advanced Study Institute on Computer Aided Analysis of Rigid and Flexible Mechanical Systems* , Volume II, pp. 91-110, Troia, Portugal, 27 June - 9 July, 1993.

[25] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[26] Frances Paulisch, S. Manke, Walter Tichy, Persistence for Arbitrary C++ Data Structures, In *Proceedings of International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, FRG, May 1990, pp. 378–391.

[27] PELAB, *Modelica activities in PELAB*, The Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, http://www.ida.liu.se/~pelab/modelica

[28] Ben Shneiderman, The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization, in *Proceedings of 1996 IEEE Conference on Visual Languages*, Boulder, CO, Sept. 3–6,1996, pp. 336–343.

[29] VRML Consortium, *VRML WWW site*, http://www.vrml.org

[30] Wolfram Research, *Mathematica*, Wolfram Research Inc., http://www.wolfram.com

Paper 1

Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment[1]

Peter Fritzson, Vadim Engelson, Lars Viklund
Programming Environments Laboratory
Department of Computer and Information Science
Linköping University, S-581 83 Linköping,
Sweden

Email: {petfr,vaden,larvi}@ida.liu.se
Phone: +46 13 281000, Fax: +46 13 282666

Abstract. ObjectMath is a high-level programming environment and modeling language for scientific computing which supports variants and graphical browsing in the environment and integrates object-oriented constructs such as classes and single and multiple inheritance within a computer algebra language. In addition, composition of objects using the part-of relation and support for solution of systems of equations is provided. This environment is currently being used for industrial applications in scientific computing. The ObjectMath environment is designed to handle realistic problems. This is achieved by allowing the user to specify transformations and simplifications of formulae in the model, in order to arrive at a representation which is efficiently solvable. When necessary, equations can be transformed to C++ code for efficient numerical solution. The re-use of equations through inheritance in general reduces models by a factor of two to three, compared to a direct representation in the Mathematica computer algebra language. Also, we found that multiple inheritance from orthogonal classes facilitates re-use and maintenance of application models.

1 Background

The goal of the ObjectMath project is to develop a high-level programming environment that enhances the program development process in scientific computing, initially applying the environment to advanced machine element analysis (a machine element can loosely be defined as "some important sub-structure of a machine"). There is a clear need for such tools as the current state of the art in the area is still very low-level. Most scientific software is still developed in FORTRAN the traditional way, manually translating mathematical models into procedural code and spending much time on debugging and fixing convergence problems. See [7] for a detailed discussion. The ObjectMath programming environment is centered around the ObjectMath modeling language which combines object-oriented constructs with computer algebra. In this paper we describe the ObjectMath language and

1. This paper appears in the Proceedings of DISCO'93 - International Symposium on the Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept 1993, LNCS 722, Springer Verlag.

report experiences from using it for modeling and analyzing realistic machine element analysis problems.

The current practice in mechanical analysis software modeling and implementation can be described as follows: Theory development is usually done manually, using only pen and paper. Equations are simplified and rewritten by hand to prepare for solution of relevant properties. This includes a large number of coordinate transformations, which are laborious and error prone. In order to perform numerical computations the mathematical model is implemented in some programming language, most often FORTRAN. Existing numerical subroutines might be used, but data-flow between routines must still be written by hand. Tools such as finite element analysis (FEM) or multibody systems analysis programs can at best be used for limited subproblems as the total computational problem usually is too complex. The program development process is highly iterative. If correlation with experiments is not achieved the theoretical model has to be refined, which subsequently requires changes in the numerical program. Numerical convergence problems often arise, since the problems usually are non-linear. Often as much as 50-75% of the total time of a project is spent on writing and debugging FORTRAN programs.

The ideal tool for modeling and analysis in scientific computing should eliminate these low-level problems and allow the designer to concentrate on the modeling aspects. Some of the properties of a good programming environment for modeling and analysis in scientific computing are:

- The user works at a high level of abstraction.
- Modeling is done using formulae and equations, with good structuring support (such as object-oriented techniques).
- Support for symbolic computation is provided. One example is automatic symbolic transformation between different coordinate systems.
- The environment should provide support for numerical analysis.
- The environment should support changes in the model. A new iteration in the development cycle should be as painless as possible.

Symbolic computation capabilities provided by computer algebra systems [4] are essential in a high-level programming environment for scientific computing. Some existing computer algebra systems are Macsyma [15], Reduce [9], Maple [2], or Mathematica [18]. However, their support for structuring complex models is too weak.

In the following sections we describe the ObjectMath programming environment and language, especially with respect to uses of inheritance, composition and variants. Finally, we compare ObjectMath with related work and present conclusions.

2 The ObjectMath Programming Environment

The ObjectMath programming environment is designed to be easy to use for application engineers, e.g. in mechanical analysis who are not computer scientists. It is interactive and includes a graphical browser for viewing and editing inheritance hierarchies, an application oriented editor for editing ObjectMath equations and formulae, the Mathematica computer algebra system for symbolic computation, support for generation of numerical code from equations, an interface for calling external functions, and a class library. The graphical browser is used for viewing and editing ObjectMath inheritance hierarchies. ObjectMath

code is automatically translated into Mathematica code and symbolic computations can be done interactively in Mathematica. Figure 1 shows the environment during a typical session.

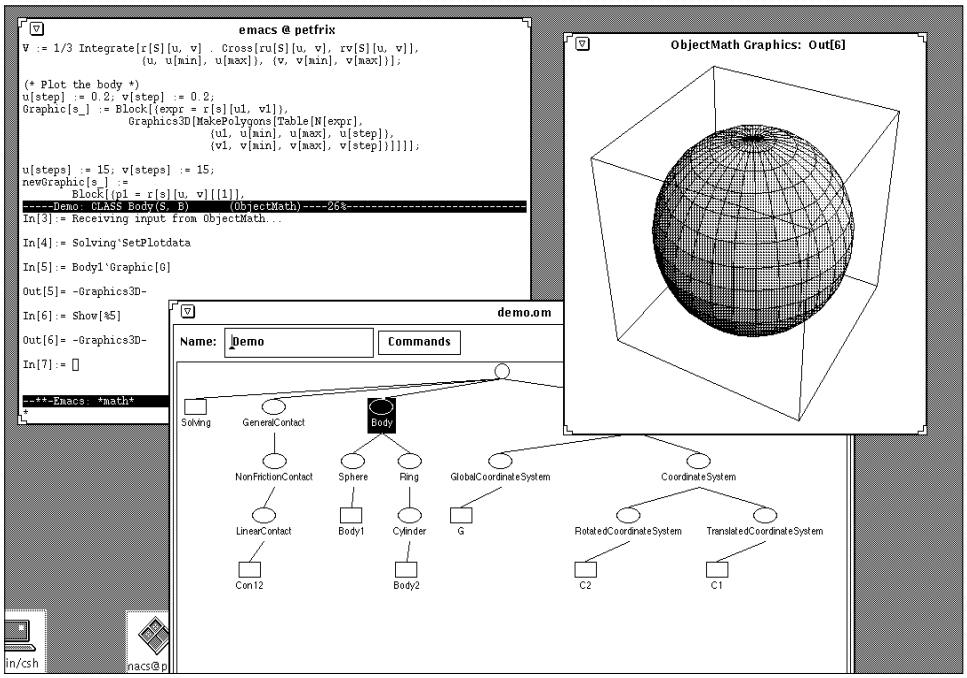


Fig. 1. The ObjectMath programming environment in use. The displayed tree in the graphical browser window shows the inheritance hierarchy of classes in the model, the text windows to the left show the currently edited class definition and the Mathematica window for symbolic computations, whereas the visualized object (Body1 in the window at upper right) is instantiated from a specialized Sphere class.

The programming environment currently runs on Sun workstations under the X window system. Recently additional capabilities has been added to the environment, such as multiple inheritance, composing objects of parts, variant handling of models and classes and extended code generation support. This is described later in the paper. A description of the implementation of an earlier version of ObjectMath can be found in [17]. Initial experience from using an early version of ObjectMath is reported in [8].

3 The ObjectMath Language

ObjectMath is both a language and a programming environment. The current ObjectMath language has recently been enhanced with features for *multiple inheritance* and modeling *part-of* relations between objects. Both of these features has turned out to be important in realistic application models. An early version of the ObjectMath language only supported single inheritance [16].

The ObjectMath language is an hybrid modeling language, combining object-oriented constructs with a language for symbolic computation. This makes ObjectMath a suitable language for implementing complex mathematical models, such as those used in machine

element analysis. Formulae and equations can be written with a notation that closely resembles conventional mathematics, while the use of object-oriented modeling makes it possible to structure the model in a natural way.

We have chosen to use an existing computer algebra language, Mathematica, as a basis for ObjectMath. Building an object-oriented layer on top of an existing language is not ideal, but this decision was made to make it possible to quickly implement a stable prototype that could be used to test the feasibility of our approach. The current ObjectMath language is a prototype and we plan to redesign the whole language in the future, making it independent of Mathematica.

Mathematica was chosen over other similar systems partly because it was already in use by our industrial partner, and partly because of its excellent support for three-dimensional graphics. The relationship between Mathematica and ObjectMath can be compared to that between C and C++. The C++ programming language is basically the C language augmented with classes and other object-oriented language constructs. In a similar way, the ObjectMath language can be viewed as an object-oriented version of the Mathematica language.

3.1 Object-Oriented Modeling

When working with a mathematical description that consists of hundreds of equations and formulae, for instance one describing a complex machine element, it is highly advantageous to *structure* the model. A natural way to do this is to model machine elements as *objects*. Physical bodies, e.g. rolling elements in a bearing, are modeled as separate objects. Properties of objects like these might include a surface description, a normal to the surface, forces and moments on the body, and a volume. These objects might define operations such as finding all contacts on the body, computing the forces on or the displacement of the body, and plotting a three-dimensional picture of the body.

Abstract concepts can also be modeled as objects. Examples of such concepts are coordinate systems and contacts between bodies. The coordinate system objects included in the ObjectMath class library define methods for transforming points and vectors to other coordinate systems. Equations and formulae describing the interaction between different bodies are often the most complicated part of problems in machine element analysis. This makes it practical to encapsulate these equations in separate *contact objects*. One advantage of using contact objects is that we can substitute one mathematical contact model for another simply by plugging in a different kind of contact object. The rest of the model remains completely unchanged. When using such a model in practice, one often needs to experiment with different contact models to find one which is exact enough for the intended purpose, yet still as computationally efficient as possible. The ObjectMath class library contains several different contact classes.

The use of *inheritance* facilitates *reuse* of equations and formulae. For example, a cylindrical roller element can inherit basic properties and operations from an existing general cylinder class, refining them or adding other properties and operations as necessary. Inheritance may be viewed not only as a sharing mechanism, but also as a concept specialization mechanism. This provides another powerful mechanism for structuring complex models in a comprehensive way. Iteration cycles in the design process can be simplified by the use of inheritance, as changes in one class affects all objects that inherits

from that class. *Multiple inheritance* facilitates the maintenance and construction of classes which need to combine different orthogonal kinds of functionality.

The *part-of* relation is important for modeling objects which are *composed* of other objects. This is very common in practice. For example, a car is composed of parts such as wheels, motor, seats, brakes, etc. This modeling facility was missing from the first version of ObjectMath, which caused substantial problems when applying the system to more complicated applications. Note that the notions of *composition* of parts, and *inheritance* are quite different and orthogonal concepts. Inheritance is used to model specialization hierarchies, whereas composition is used to group parts within container objects while still preserving the identity of the parts. Thus, composition has nothing to do with specialization. Sometimes these concepts are confused; there have been attempts to use inheritance to implement composition, usually with disastrous results for the model structure.

Object-oriented techniques make it practical to organize repositories of reusable software components. The ObjectMath class library is one example of such a software component repository. It contains general classes, for instance material property classes, different contact classes and classes for modeling simple bodies such as cylinders and spheres.

3.2 ObjectMath Classes and Instances

A *CLASS* declaration declares a class which can be used as a template when creating objects. ObjectMath classes can be parameterized. The ObjectMath *INSTANCE* declaration is, in a traditional sense both a declaration of class and a declaration of one object (instance) of this class. This makes the declaration of classes with singleton instances compact.

An array containing a symbolic number of objects can be created from one *INSTANCE* declaration by adding an index variable in brackets to the instance name. This allows for the creation of large numbers of nearly identical objects, for example the rolling elements in a rolling bearing. To represent differences between such objects, functions (methods) that are dependent upon the array index of the instance can be used. The implementation makes it possible to do computations with a symbolic number of elements in the array.

The bodies of ObjectMath *CLASS* and *INSTANCE* declarations contain formulae and equations. Mathematica syntax is used for these. Note that the Mathematica context mark, `, denotes remote access, i.e. `X`y` is the entity `y` of the object `X`.

There are some limitations in the current prototype implementation of the ObjectMath language for ease of implementation. Most notable is that a mechanism for enforcing encapsulation is missing. Encapsulation is desirable because it makes it easier to write reusable software since the internal state of an object may only be accessed through a well defined interface.

On the other hand, access to equations defined in different classes is necessary when specifying computer-algebra related transformations, substitutions and simplifications. Such symbolic operations on equations need to access the source representation of those equations. Since transformations are specified manually and executed interactively, equations should not be hidden from the user. Enforcing strict encapsulations of equations is only possible in systems which do not allow manual manipulation of equations.

3.3 Single Inheritance Examples: Cylinders and Spheres

In this section we use some of the classes of physical objects from an ObjectMath model to exemplify the ObjectMath language. In addition to classes describing bodies with different geometry depicted in the inheritance hierarchy of Figure 2, there are additional classes which describe interactions between bodies and coordinate systems. The full inheritance hierarchy is displayed in the lower part of Figure 1 and the full class definitions are available in Appendix A. Note that the inheritance hierarchy usually is edited graphically so that the user does not have to write the class headers by hand.

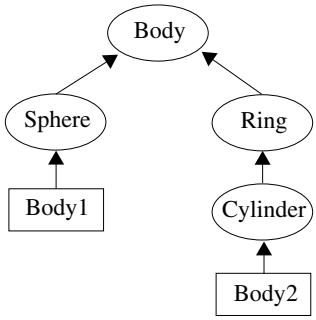


Fig. 2. An inheritance hierarchy of classes for modeling bodies with different geometries such as cylinders and spheres.

At the top of this inheritance sub-hierarchy is the class `Body`, which contains definitions and functions common to all bodies. This includes the function `r[S][u_,v_]` which describes the geometry of a body through a parametric surface; partial differentials of this surface, a general formula for the volume of a body, a function for plotting 3D graphic images of bodies, etc. This class has two parameters: `S` which is the name of the body-centered coordinate system, and `B` which is the set of bodies this body is interacting with. For example, when inherited down to `Body1`, the parametric surface function is instantiated to `r[C1][u_,v_]` since `C1` is the body-centered coordinated system of `Body1`.

```
CLASS Body(S, B)
(* Geometry defined through a parametric surface *)
 r[S][u_, v_];
 r[s_][u_, v_] := S`TransformPoint[r[S][u, v], s];
 ...
(* Partial differentials of surface *)
 ru[S][u_, v_] := D[r[S][u1, v1], u1] /. { u1 -> u, v1 -> v };
 rv[S][u_, v_] := D[r[S][u1, v1], v1] /. { u1 -> u, v1 -> v };
(* Volume of body *)
 V := 1/3 Integrate[r[S][u, v] . Cross[ru[S][u, v], rv[S][u, v]],
              {u, u[min], u[max]}, {v, v[min], v[max]}];
(* Graphic method for plotting bodies *)
 Graphic[s_] := ...
(* Forces and moments, equations for equilibrium, etc ...)
 ...
END Body;
```

The class `Sphere` contains a specialization of the parametric surface function to give the

special geometry of a sphere. Finally the class and instance `Body1` instantiates a specific sphere, which is in contact with the cylinder. It actually rests on top of the cylinder, exerting a contact force on the cylinder, as is shown in Figure 3. The class `Body` is also specialized as class `Ring`, which is further specialized as class `Cylinder` and instance `Body2`. The definitions of these classes can be found in Appendix A.

```
CLASS Sphere(S, B) INHERITS Body(S, B)
  R; (* Radius *)
  ...
  u[min] := 0; u[max] := Pi;   v[min] := 0; v[max] := 2 Pi;
  r[S][u_, v_] := R * { Sin[u]*Cos[v],  Sin[u]*Sin[v],  Cos[u] };
  ...
END Sphere;

INSTANCE Body1 INHERITS Sphere(C1, {Body2})
  Con[Body2] := Con12;  (* Define contact from this body to Body2 *)
  rho;                  (* Density *)
  m := rho V;           (* Mass *)
  (* External force from gravity *)
  F$[S1][Ext][1] := 0;   F$[S1][Ext][2] := 0;   F$[S1][Ext][3] := - g m;
END Body1;
```

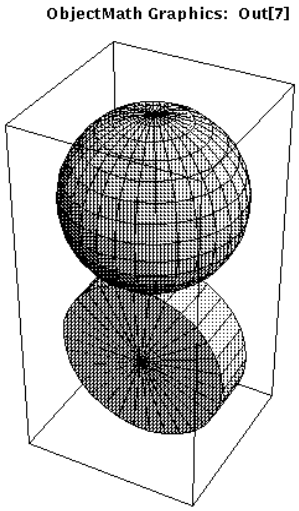


Fig. 3. The sphere rests on top of the cylinder.

3.4 Examples of Multiple Inheritance

Multiple inheritance is useful when combining orthogonal concepts. This is exemplified in Figure 2.

The filled lines denote single inheritance, whereas the dotted lines denote additional inheritance, i.e. we have multiple inheritance. Since material properties and geometry are orthogonal concepts there are no collisions between inherited definitions in this example.

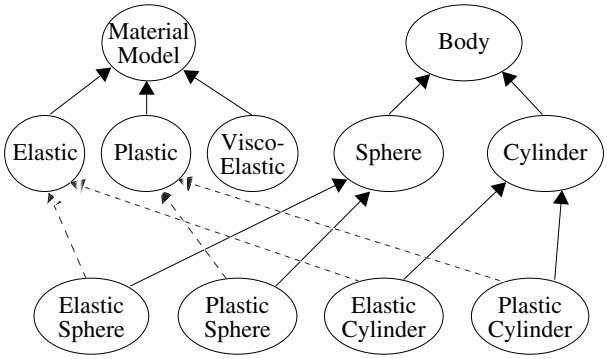


Fig. 4. Multiple inheritance hierarchy of bodies of different materials and geometries.

If instead we are forced to use a single-inheritance hierarchy as in Figure 5, we are have to repeat the equations describing material properties twice. This is bad model engineering since it would force us to repeat any changes to the material model twice. Also, this precludes the creation of pure material library classes which can be combined with other classes.

Here, the material equations describing elasticity or plasticity have to be repeated twice. This model structure is harder to maintain when changes are introduced into the model.

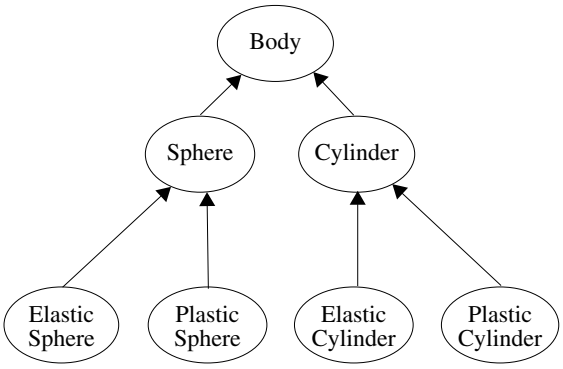


Fig. 5. Single inheritance version of the material-geometry model of Figure 2.

The general form of multiply inheriting class declarations follows below:

```
CLASS Child INHERITS Parent1,Parent2,... ParentN
  ...
END Child;
```

If there are conflicts between inherited definitions, e.g. if they have the same name, definitions from `Parent1` will override definitions from `Parent2`, which will override definitions from `Parent3`, etc. The special case when only `Parent1` is present corresponds to single inheritance.

Some example classes from the material-geometry model in Figure 2:

```
CLASS Sphere(S,B) INHERITS Body(S,B)
  R;  (* Radius - a variable *)
  r[S][u_, v_] := R * { Sin[u]*Cos[v],  Sin[u]*Sin[v],  Cos[u] };
  ....
END Sphere;

CLASS Elastic INHERITS Material_Model
  Force := k1 * delta;
END Elastic;

CLASS Plastic INHERITS Material_Model
  Force := k2 * Limit   /; delta > Limit;
  Force := k2 * delta   /; delta <= Limit;
END Plastic;

CLASS Elastic_Sphere INHERITS Sphere, Elastic
  ...
END Elastic_Sphere;
```

Another useful case of *multiple-inheritance* is shown below, where an integration method is inherited into classes from two separate inheritance hierarchies: a hierarchy of contact classes containing integrated forces and moments between bodies, and classes describing bodies themselves including integrated moments of inertia, mass and volumes.

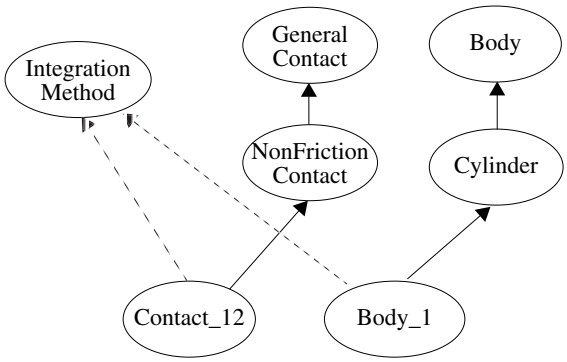


Fig. 6. Example of multiple inheritance of a numerical integration method into two different classes. Here to be used for integrating forces or volumes. One class contains contact equations; another contains volumes, moments and equilibrium equations.

The entities inherited from class `Integration_Method` will typically be a combination of entities such as procedural code, transformation rules (e.g. for symbolic differentiation), etc.

3.5 Modeling Part-Of Relations

As mentioned previously, the *part-of* relation is important for modeling objects which are composed of other objects, also noting that this concept is orthogonal to the concept of *inheritance* which is used to represent specialization. For example, a bicycle contain parts

such as wheels, frame, pedals, etc. A rolling bearing contain inner ring, outer ring, rolling elements, lubrication fluid, etc.

The ObjectMath syntax for expressing composition using the *part-of* relation is exemplified below for a `Bicycle` class:

```
CLASS Bicycle(C,P)
  ...
  PART frontwheel INHERITS Wheel(P);
  PART rearwheel INHERITS Wheel(P);
  PART frame INHERITS Body;
  ...
END Bicycle;
```

Another example is a small section of a class from a rather advanced model of four-body interaction developed by our industrial partner:

```
CLASS FourCylCurvSegBody(cBody, cmBody, cRef, cInert) INHERITS
                        DynRigidBody(cBody, cmBody,cRef, cInert)
   ...
   PART sRaceF INHERITS CylinderSeg(cBody);
   PART sRaceB INHERITS CylinderSeg(cBody);
   PART sSegL  INHERITS CylinderSeg(cBody);
   PART sSegR  INHERITS CylinderSeg(cBody);
   ...
END FourCylCurvSegBody;
```

4 Variants of Classes

During the development of complex mathematical models there is often a need to explore different variants of solution strategies and formulations of equations. One would like to experiment with alternative ways of expressing equations and transformations within a certain class and still keep the previous version of the class definition in the model. Each new variant of a class can of course be tried out by creating an entirely new model where all classes except one are identical compared to the previous model.

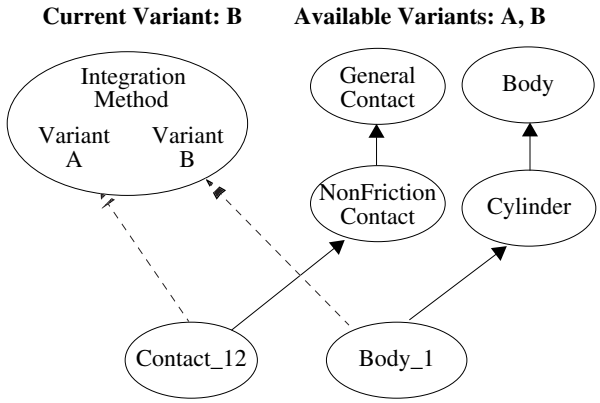


Fig. 7. The same example as in Figure 6 but with two available variants A and B. The currently active variant of the model is B, which will select Variant B of Integration Method and the default of other classes.

However, this solution is very undesirable from a software engineering point of view since it leads to a proliferation of almost identical models. For example, if a class within one model is modified, one will have to manually modify a large number of other, almost similar, models. This is both error-prone and cumbersome. The updating problem can of course be remedied by creating library modules which can be imported into models. The library solution is often not desirable however, since it implies additional restructuring work in creating libraries, especially if many classes are used only within a single model.

In order to provide a convenient solution to the variant problem we have introduced a mechanism in the ObjectMath environment which allows several variants of classes within a model.

The user specifies the names of all allowed variants within the model. For each class, it is possible to have one or more variants of the class body labeled using some of the specified variant names. There is also the notion of *current variant* of the model, which tells which variant of a class body should be selected when several are available. The most common case is that only one default unlabeled variant is available for a class, as is the case for all classes except `Integration_Method` in Figure 7. Then the single variant should of course be selected for inheritance or symbolic computations. If several variants are available, but no one matches, the unlabeled default should be selected. If no unlabeled variant is available the first one should be selected.

This mechanism is currently implemented as conditional inclusion of class-definitions using C-preprocessor style directives as shown below. If `VariantA` is the currently active model variant then the first definition of `Force` is selected; if `VariantB` is active then the second definition is selected, otherwise the third. An alternative implementation instead of the conditional inclusion mechanism would be the keep each variant separate. This has the advantage of looking somewhat cleaner, but the disadvantage of creating problems to keep the common parts of the variants consistent when the model is modified, which is why we did not select this alternative.

```
CLASS Plastic INHERITS Material_Model
#ifdef VariantA
  Force := k2 * LimitA - 35
#elif VariantB
  Force := k3 * Limit +100
#elif 1
  Force := k2 * Limit
#endif
END Plastic;
```

In this way we obtain a graceful mechanism for variant handling which is not noticed when it is not needed, and can be applied to a few classes of a model without disturbing the rest of the model. It also eliminates the problem of proliferation of almost identical models. This is similar to the notion of variants supported by NSE – the Network Software Environment [3].

However, a shortcoming of this simple variant mechanism is that it only handles variants of models when the variations are focussed to the contents of classes. It does not cope with variations in the structure of models, i.e. the inheritance and composition structure. This might be handled better by allowing models of different structure to be defined as different configurations of class definitions imported from a class library, rather extending the current variant mechanism.

5 Code Generation from ObjectMath

There is a clear need to be able to generate efficient numerical implementations from ObjectMath models. The ObjectMath system provides two alternative mechanisms for code generation. The first mechanism will translate a set of function definitions in ObjectMath syntax to corresponding C++ code, either by translating the code in a function as it is, or applying all available symbolic transformations producing a very large expression, which then is optimized using common subexpression elimination and used.

The second mechanism makes it possible to include hand-written C++ functions in ObjectMath classes as shown below. This is described in somewhat more detail in [16].

```
CLASS FindingContacts(B, Body1, Body2)
  INHERITS GeneralContacts(B, Body1, Body2)
  ...
  (* Declaration of a method written in C++ *)
  EXTERNAL "C++" ppa[su1_, v1_];
  ...
END FindingContacts;
```

The first translation mechanism is now in regular use by our industrial partner SKF Engineering & Research Centre to translate the computational parts of whole models (i.e. not the parts which specify symbolic transformations) into C++ code. For example, a two-body interaction model of approximately 150 kbytes of ObjectMath code was translated into 450 kbytes of Mathematica code, of which the computational part was translated into 440 kbyte of C++ code. Still, this C++ code is rather compact due to the use of overloading matrix and vector operations on arithmetic operators such as “*” and “+”. The re-use due to inheritance of ObjectMath classes and composition of parts makes the model approximately a factor of three more compact than if it would have been represented directly in Mathematica code.

The translator to C++ code is implemented in Mathematica. The whole translation process for the twobody model takes approximately 2 hours on a Sun Sparcstation ELC (a 20 Mips workstation). Most of the time is spent on symbolic transformations and simplifications, and on common subexpression elimination on the very large expressions generated during the transformations. Separate procedures consisting of many small statements are usually generated from such large expressions. To be able to generate efficient type-correct C++ code and to resolve some ambiguities, it was necessary to add type declarations of variables and functions to ObjectMath models.

Our aim is to eventually provide fully automatic translation of whole models. Currently, the user has to write calls to explicitly invoke the translator on each ObjectMath function, expression, or set of equations that should be translated into C++. In order to achieve this goal, additional type and structure information in the form of declarations will have to be added to models, as well as extending the capabilities of the translator itself.

6 Applications of ObjectMath

So far the ObjectMath environment has been tested and evaluated by modeling and analyzing several different problems, two of which are briefly described here:

- A three-dimensional model describing a rolling bearing, see Figure 8.

- An advanced surface description model, used for rolling bearing analysis.

The rolling bearing example was designed to be used as a realistic but manageable test case for the ObjectMath language and environment. It consists of over 200 equations and can easily be extended, for instance with more realistic contact models. Fritzson *et. al.* describes both the mathematical model [6] and the ObjectMath implementation [8]. Figure 8 shows a three-dimensional view of the bearing, automatically generated from equations in the ObjectMath model.

The bearing consists of an inner ring, an outer ring, and a number of rolling elements in between. Each of these correspond to an ObjectMath instance declaration, except for the rolling elements, where the ObjectMath feature of declaring an array of instances is used.

The second implemented ObjectMath model mentioned here is being used in the development of an advanced surface description model which is used to model the interaction between bodies. If ObjectMath had not been available, this implementation would have been developed by hand and coded in FORTRAN.

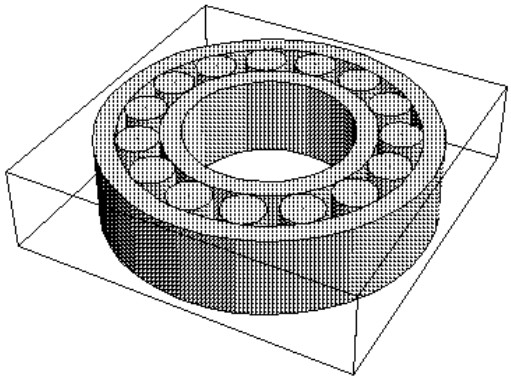


Fig. 8. A view of the example rolling bearing, automatically generated from an equational description.

7 Related Work

Object-oriented modeling has been used for a number of different application areas, and mathematical modeling in general [13]. ObjectMath also has a lot in common with conventional computer algebra systems (in particularly Mathematica). But, as mentioned in the introduction, these systems usually lack capabilities that are important for advanced scientific computing, most notably good structuring support, e.g. as provided by object oriented notions.

There are other computer algebra systems which are to some extent based on object-oriented concepts. One of the better known examples is AXIOM [11] and its predecessor SCRATCHPAD [10] which have type systems which in some sense are object-oriented, even though the language constructs provided are different from the ones usually found in object-oriented languages. Another example is the TASSO system [12]. However, the object-oriented notions supported by the ObjectMath language is primarily intended for modeling in scientific computation, for instance in engineering applications. Most other computer algebra systems employing object-oriented techniques focuses on using these techniques for modeling mathematical objects, implementing algebraic algorithms etc.

Omola [1] is an object-oriented modeling language with single-inheritance, also supporting composition. It is primarily designed as a simulation modeling language for continuous time systems, especially in the application area of control engineering. It contains no computer algebra support. A similar system, called ASCEND, is described by Piela *et. al.* [14].

8 Conclusions

There is a strong need for efficient high-level structuring tools and languages in scientific computing. We feel that the ObjectMath system is highly successful in satisfying part of this need. Complex mathematical equations and functions can be expressed at a high level of abstraction rather than as procedural code. ObjectMath integrates computer-algebra language features with object-oriented notions such as multiple inheritance, classes, as well as composition for modeling structured objects. Object-oriented notions allow better structure of models and permit reuse of equations. Variant support permits convenient incorporation of multiple solution strategies and different equational representations within a single model. Our conclusion that these facilities are important is supported by the successful modeling and analysis of several realistic applications.

9 Acknowledgments

Henrik Nilsson improved the english in this paper. Johan Herber has implemented the Emacs support in ObjectMath. Joakim Malmén has done most of the implementation work on the most recent version of the ObjectMath to C++ translator. Dag Fritzson has provided mathematical expertise and developed the majority of the currently available ObjectMath models. The ObjectMath parser is partly based on a Mathematica parser in Common Lisp written by Richard Fateman [5].

This work is supported by The Swedish Board for Technical and Industrial Development (NUTEK), in part under the Esprit project PREPARE. It is also supported in part by SKF Engineering & Research Centre, Nieuwegein, The Netherlands.

10 References

[1] Mats Andersson. Omola – an object-oriented language for model representation. Licentiate thesis, Department of Automatic Control, Lund Institute of Technology, P.O. Box 118, S-221 00 Lund, Sweden, May 1990.

[2] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1991.

[3] Willaim Courington, Jonathan Feiber, and Masahiro Honda. NSE highlights. In Mark Hall and John Barry, editors, *The Sun Technology Papers*. Springer-Verlag, 1990.

[4] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra – Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.

[5] Richard J. Fateman. A Mathematica parser in Common Lisp. Personal Communications. Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1991.

[6] Dag Fritzson and Peter Fritzson. Equational modeling of machine elements – applied to rolling bearings. Technical Report LiTH-IDA-R-91-05, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, March 1991.

[7] Peter Fritzson and Dag Fritzson. The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers & Structures*, 45(2):387–395, 1992. Also as technical report LiTH-IDA-R-91-04,Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden.

[8] Peter Fritzson, Lars Viklund, Johan Herber, and Dag Fritzson. Industrial application of object-oriented mathematical modeling and computer algebra in mechanical analysis. In Georg Heeg, Boris Magnusson, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems – TOOLS 7*, pages 167–181. Prentice Hall, 1992.

[9] A. C. Hearn. *REDUCE-3 User's Manual, version 3.3*. The Rand Corporation, Santa Monica, Califoria, USA, 1987. Publication CP78 (7/78).

[10] Richard D. Jenks. A primer: 11 keys to new SCRATCHPAD. In John Fitch, editor, *Proceedings of EUROSAM 84/International Symposium on Symbolic and Algebraic Computation*, July 1984.

[11] Richard D. Jenks and Robert S. Sutor. *AXIOM – The Scientific Computation System*. Springer-Verlag, 1992.

[12] C. Limongelli, A. Minola, and M. Temperini. Design and implementation of symbolic computation systems. In P. W. Gaffney and E. N. Houstis, editors, *Programming Environments for High-Level Scientific Problem Solving*, pages 217–226. North-Holland, 1992. Proceedings of the IFIP TC2/WG 2.5 Working Conference on Programming Environments for High-Level Scientific Problem Solving.

[13] Thomas W. Page, Jr., Steven E. Berson, William C. Cheng, and Richard R. Muntz. An object-oriented modeling environment. In *OOPSLA'89 Conference Proceedings*, pages 287–296, 1989.

[14] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. *Computers & Chemical Engineering*, 12(7):53–72, 1991.

[15] Symbolics Inc. *MACSYMA Reference Guide*, 1985.

[16] Lars Viklund and Peter Fritzson. An object-oriented language for symbolic computation – applied to machine element analysis. In Paul S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 397–405. ACM Press, 1992.

[17] Lars Viklund, Johan Herber, and Peter Fritzson. The implementation of ObjectMath – a high-level programming environment for scientific computing. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction – 4th International Conference, CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 312–318. Springer-Verlag, 1992.

[18] Stephen Wolfram. *Mathematica – A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, second edition, 1991.

Appendix A. The bearing model expressed in the ObjectMath language

This appendix contains the full model SphereCylinder referred to earlier in this paper.

```
MODEL SphereCylinder;
PACKAGES "Vectors`", "Subst`", "Misc`", "Plot`";

g; (* Gravity constant *)

CLASS AbstractCoordinateSystem
  ...
END AbstractCoordinateSystem;

CLASS CoordinateSystem(Reference, A, R) INHERITS AbstractCoordinateSystem
  ...
END CoordinateSystem;

CLASS TranslatedCoordinateSystem(Reference, R) INHERITS
 ...
END TranslatedCoordinateSystem;

INSTANCE C1 INHERITS TranslatedCoordinateSystem(G, {0,0,d})
END C1;

CLASS RotatedCoordinateSystem(Reference, Phi) INHERITS ...
  ...
END RotatedCoordinateSystem;

INSTANCE C2 INHERITS RotatedCoordinateSystem(G, {Pi/2,0,0})
END C2;

CLASS GlobalCoordinateSystem INHERITS AbstractCoordinateSystem
 FromGlobal := {this};
END GlobalCoordinateSystem;

INSTANCE G INHERITS GlobalCoordinateSystem
END G;

CLASS Body(S, B)
 (* Surface *)
 u[min]; u[max]; v[min]; v[max]; (* virtual *)
 r[S][u_, v_]; (* virtual *)
 r[s_][u_, v_] := S`TransformPoint[r[S][u, v], s];

 (* Partial differentials of surface *)
 ru[S][u_, v_] := D[r[S][u1, v1], u1] /. { u1 -> u, v1 -> v };
 rv[S][u_, v_] := D[r[S][u1, v1], v1] /. { u1 -> u, v1 -> v };

 (* Normal of surface *)
 n[S][u_, v_] := Cross[ru[S][u, v], rv[S][u, v]] /
         AbsVector[Cross[ru[S][u, v], rv[S][u, v]]];
 n[s_][u_, v_] := S`TransformVector[n[S][u, v], s];

 (* Volume of body *)
 V := 1/3 Integrate[r[S][u, v] . Cross[ru[S][u, v], rv[S][u, v]],
         {u, u[min], u[max]}, {v, v[min], v[max]}];

 (* Plot the body *)
 u[step] := 0.2; v[step] := 0.2;
 Graphic[s_] := Block[{expr = r[s][u1, v1]},
         Graphics3D[MakePolygons[Table[N[expr],
                {u1, u[min], u[max], u[step]},
                {v1, v[min], v[max], v[step]}]]]];

 (* Coordinate system to solve equilibrium in *)
 S1 := S; (* May be overridden in subclasses *)

 (* Contact objects *)
```

```
 Con[b_]; (* virtual *)

 (* Forces and moments *)
 F[S1][b_] := Array[F$[S1][b], 3];
 F[s_][b_] := AccessMember[S1, "TransformVector"][F[S1][b], s];
 M[S1][b_] := Array[M$[S1][b], 3];
 M[s_][b_] := AccessMember[S1, "TransformVector"][M[S1][b], s];

 (* External loading *)
 F[S1][Ext] := Array[F$[S1][Ext], 3];
 M[S1][Ext] := Array[M$[S1][Ext], 3];
 p[S1][Ext] := {0, 0, 0};

 (* Equilibrium *)
 Eq[1] := Plus @@ F[S1] /@ B + F[S1][Ext] == {0, 0, 0};
 Eq[2] := Plus @@ ( M[S1][#] +  Cross[AccessMember[AccessMember[#, "Con"]
       [this], "p"][S1], F[S1][#]] &) /@ B +  M[S1][Ext] + Cross[p[S1][Ext],
          F[S1][Ext]]== {0, 0, 0};
END Body;

CLASS Ring(S, B) INHERITS Body(S, B)
 (* Variables *)
 Rb; (* Inner radius *)
 Ra; (* Outer radius *)
 L; (* Width *)

 (* Definition of parametric surface *)
 u[min] := 0; u[max] := 4;   v[min] := 0; v[max] := 2 Pi;

 x1[u_] := Rb + (Ra - Rb) u;
 x2[u_] := Ra;
 x3[u_] := Rb + (Ra - Rb) (3 - u);
 x4[u_] := Rb;

 x[u_] := x1[u] /; 0 <= u < 1;
 x[u_] := x2[u] /; 1 <= u < 2;
 x[u_] := x3[u] /; 2 <= u < 3;
 x[u_] := x4[u] /; 3 <= u <= 4;

 z1[u_] := 1 / 2 L;
 z2[u_] := -L (u - 3 / 2);
 z3[u_] := - 1 / 2 L;
 z4[u_] := L (u - 7 / 2);

 z[u_] := z1[u] /; 0 <= u < 1;
 z[u_] := z2[u] /; 1 <= u < 2;
 z[u_] := z3[u] /; 2 <= u < 3;
 z[u_] := z4[u] /; 3 <= u <= 4;

 (* The is the complete rotation surface *)
 r[S][u_, v_] := { x[u] Cos[v],   x[u] Sin[v],   z[u] };

 (* Partial differentials of surface *)
 Dx[u_] := x1'[u] /; 0 <= u < 1;
 Dx[u_] := x2'[u] /; 1 <= u < 2;
 Dx[u_] := x3'[u] /; 2 <= u < 3;
 Dx[u_] := x4'[u] /; 3 <= u <= 4;

 Dz[u_] := z1'[u] /; 0 <= u < 1;
 Dz[u_] := z2'[u] /; 1 <= u < 2;
 Dz[u_] := z3'[u] /; 2 <= u < 3;
 Dz[u_] := z4'[u] /; 3 <= u <= 4;

 x' := Dx;   z' := Dz;
 u[step] := 1; v[step] := Pi/15;   u[steps] := 4; v[steps] := 31;
END Ring;

CLASS Cylinder(S, B) INHERITS Ring(S, B)
 Rb := 0;   u[max] := 3;
END Cylinder;
```

```
INSTANCE Body2 INHERITS Cylinder(C2, {Body1})
 Con[Body1] := Con12;
END Body2;

CLASS Sphere(S, B) INHERITS Body(S, B)
 (* Variables *)
 R; (* Radius *)
 (* Definition of parametric surface *)
 u[min] := 0; u[max] := Pi;
 v[min] := 0; v[max] := 2 Pi;
 r[S][u_, v_] := R { Sin[u] Cos[v],  Sin[u] Sin[v], Cos[u] };
 u[step] := Pi/15; v[step] := Pi/15;   u[steps] := 16; v[steps] := 31;
END Sphere;

INSTANCE Body1 INHERITS Sphere(C1, {Body2})
 Con[Body2] := Con12;
 rho;          (* Density *)
 m := rho V;   (* Mass *)
 (* External force from gravity *)
 F$[S1][Ext][1] := 0;   F$[S1][Ext][2] := 0;   F$[S1][Ext][3] := - g m;
 (* No external moment *)
 M[S1][Ext] := {0, 0, 0};
END Body1;

CLASS GeneralContact(S, Body1, Body2)
 (* Variables: *)
 delta;   u[Body1]; v[Body1];   u[Body2]; v[Body2];
 (* The contact point: *)
 p[S] := Body1`r[S][u[Body1], v[Body1]] + 1/2 delta Body1`n[S][u[Body1],
v[Body1]];
 p[s_] := S`TransformPoint[p[S], s];
 (* Equations *)
 Eq[1] := Body1`n[S][u[Body1], v[Body1]] + Body2`n[S][u[Body2], v[Body2]]
           == { 0, 0, 0 };
 Eq[2] := delta Body1`n[S][u[Body1], v[Body1]] ==
     Body2`r[S][u[Body2], v[Body2]] - Body1`r[S][u[Body1], v[Body1]];
 Eq[3] := Body1`F[S][Body2] + Body2`F[S][Body1] == { 0, 0, 0 };
 Eq[4] := Body1`M[S][Body2] + Body2`M[S][Body1] == { 0, 0, 0 };
END GeneralContact;

CLASS NonFrictionContact(B, Body1, Body2) INHERITS GeneralContact(B, Body1,
Body2)
 k[delta_];   (* The contact stiffness function *)
 (* Equations: *)
 Eq[10] := Body1`F[S][Body2] == k[delta] Body1`n[S][u[Body1], v[Body1]];
 Eq[11] := Body1`M[S][Body2] == { 0, 0, 0 };
END NonFrictionContact;

CLASS LinearContact(B, Body1, Body2) INHERITS NonFrictionContact(B, Body1,
Body2)
 k[delta_] := Cstiff[delta] delta - 1;
 Cstiff[delta_] := Csoft /; delta > 0;
 Cstiff[delta_] := Chard /; delta <= 0;
END LinearContact;

INSTANCE Con12 INHERITS LinearContact(C1, Body1, Body2)
END Con12;

INSTANCE Solving
 SetPlotdata := ( Body1`R := 1; Body2`Ra := 1; Body2`L := 0.6;
        C1`d := Body1`R + Body2`Ra;
        );
 ClearPlotdata := ( Clear[Body2`Ra, Body2`L, Body1`R, C2`d]; );
 Force := Flatten[Solve[{ Body1`Eq[1], Body2`Eq[1], Con12`Eq[3],
            g!=0, Body1`rho!=0, Body1`R!=0 },
            Body2`F[C2][Ext],
            { Body2`F$[C2][Body1][2], Body1`F$[C1][Body2][3] } ]];
END Solving;
```

Paper 2

ObjectMath Inheritance and Composition Diagram Editor

Abstract

ObjectMath is a new object-oriented modeling language for scientific computing. The major innovation of this language is the introduction of object-oriented structure into a computer algebra language making it possible to group equations and formulae into classes. ObjectMath contains three object-oriented structuring constructs (class, instance and part) providing classes, single and multiple inheritance and composition of parts. Typical models in ObjectMath include between 10 and 30 such constructs. Inheritance and composition relationships are established between them. The problem is how to inspect, browse, and modify these relations in a convenient way. Our solution to this problem is using a graphical two-dimensional diagram editor connected to a text editor. An integrated programming environment for ObjectMath includes this editor. The report describes how object-oriented constructs of ObjectMath are mapped to their graphical representation. The ObjectMath syntax rules are mapped to diagram editor operations so that only syntactically correct models can be created.

1 Introduction

ObjectMath is a mathematical modeling language and an integrated environment for programming in this language [1]. This language combines computer algebra capabilities available in Mathematica[2] with object oriented constructs. One of the reasons to introduce object-orientation into Mathematica is to facilitate *reuse* of functions and expressions. Knowledge about abstract and real world objects and their numerical properties can be expressed in variables, functions and equations. However, in order to reuse them, these variables and equations should be encapsulated into classes, which in turn can be gathered into reusable class libraries.

The use of inheritance facilitates reuse of equations and formulae. It appears that in the case of ObjectMath, like in many other object-oriented languages, an application might include many levels of inheritance. In addition, some classes are composed of objects which instantiate other classes. Multiple inheritance allows constructing classes from other classes combining different orthogonal kinds of functionality.

Typical ObjectMath models contain from 10 to 30 classes and instances. It is hard to get an overview of the whole system without graphical presentation of relations between classes and instances. This is the reason why a graphical editor is necessary. Initially an class diagram editor was developed for ObjectMath 3.0 by Lars Viklund and Rickard Westman. Figure 1 of [1] (**paper 1** in this thesis) shows this graphical editor in use. This editor was able to display directed graphs without

loops only and therefore could not be used for multiple inheritance and for part-of relations between classes. We developed a new editor for ObjectMath, version 4.0, in 1993.

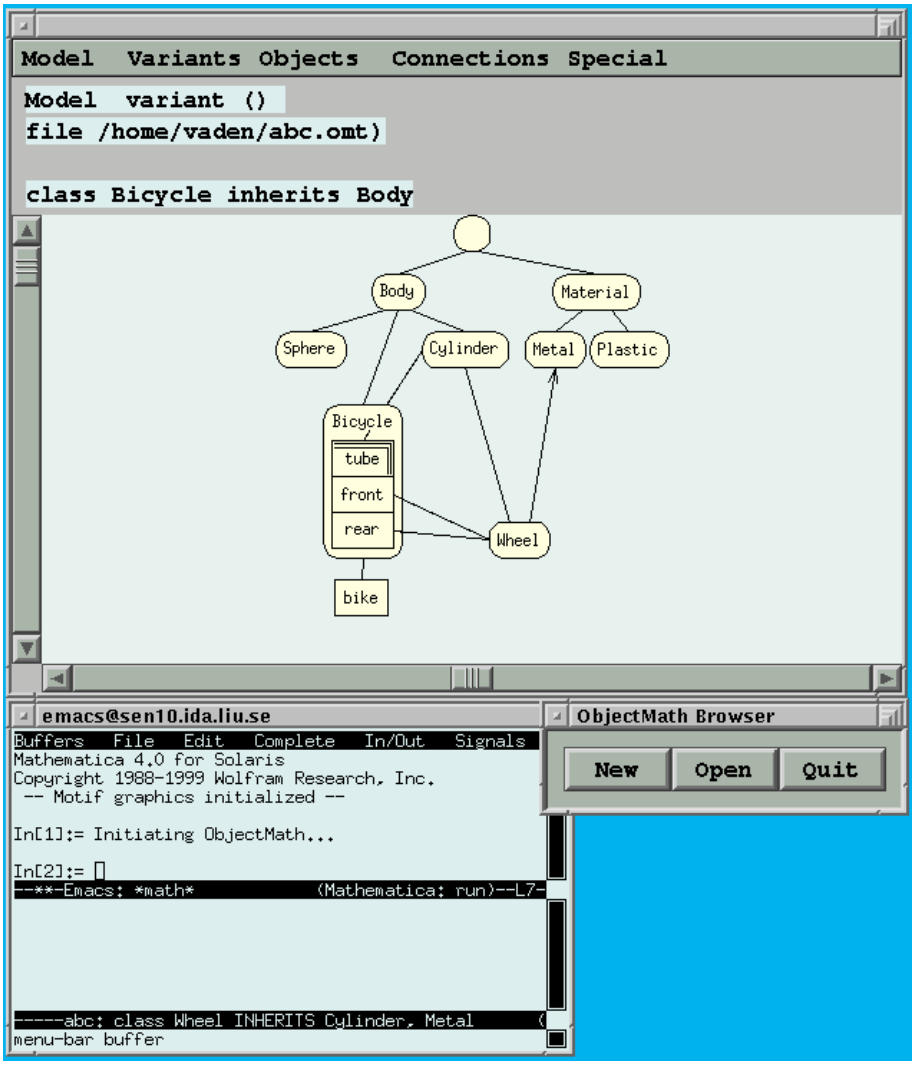


Figure 1: *The ObjectMath environment consisting of a diagram editor window, a program text window and the start window.*

This report contains:

- formal definition of the syntactic rules for relations between object-oriented constructs in ObjectMath;
- definition of a mapping between models in ObjectMath and class relationship (i.e. inheritance and composition) diagrams;

- definition of operations on relationships between object oriented constructs that can be performed by the user;
- definition and implementation of the above mentioned mapping and operations in a graphical class diagram editor.

2 Syntactic Rules of ObjectMath and Mapping between Textual and Graphical Representation

In this section we give an example of textual representation of an ObjectMath model. There are grammar rules (context-free and context-dependent) defining the ObjectMath syntax (Section 3).

The graphical representation has been designed in order to reflect the structure of ObjectMath models written according to the rules. In Section 4 we introduce the ObjectMath graphical notation and explain the mapping of syntax rules into operations of the diagram editor.

3 The Textual Syntax of ObjectMath

We consider an ObjectMath model of a bicycle consisting of two wheels (front and rear) and five tubes (cylinders). Currently we ignore all class variables and all equations since they do not affect graphical user interface and are not shown in the diagram. The ellipsis (`...`) mark the places where variable declarations, functions and equations can be inserted.

The Figure 2 contains an ObjectMath model of a bicycle in textual notation.

```
model abc;
                              ...
class Body      ...   end;
class Material  ...   end;
class Metal         inherits Material  ... end;
class Plastic(b,b)  inherits Material  ... end;
class Sphere        inherits Body      ... end;
class Cylinder      inherits Body      ... end;
class Wheel inherits Cylinder, Metal   ... end;
class Bicycle inherits Body
    part tube[five] inherits Cylinder;
    part front      inherits Wheel(t);
    part rear       inherits Wheel;
    ...
end;
instance bike inherits Bicycle  ...  end;
```

Figure 2: *An ObjectMath example (bicycle model) in textual notation.*

To explain the components of the ObjectMath textual representation further we describe the syntax of ObjectMath using extended BNF notation. The following syntactic meta symbols are used in this description:

- `[ ]` – optional grammar elements.
- `{ }` – grammar elements repeat zero or more times.
- `|` – alternative elements.
- Keywords are written in **`bold`** face.

This description (Figure 3) contains 11 formal rules of the ObjectMath grammar, and 5 rules in natural language defining context dependencies. There are many more rules which specify how *variables and equations* are written in ObjectMath. These rules are ignored here since they do not affect graphical user interface and are not shown in the diagram.

Rule 8 should be explained in more detail using mathematical notation.

Assume that a graph consists of k nodes $\{n_1, ..., n_k\}$ and m directed edges $\{(n_{1_1} \rightarrow n_{1_2}), ..., (n_{m_1} \rightarrow n_{m_2})\}$. A directed loop is a sequence of edges $((n_{j_1} \rightarrow n_{j_2}), (n_{j_2} \rightarrow n_{j_3}), ..., (n_{j_p} \rightarrow n_{j_1}))$, where $p > 0$. In Rule 8 we represent all classes, parts and instances as nodes. An edge $(n_a \rightarrow n_b)$ exists if n_b inherits class n_a or n_b contains n_a. The rule states that if an ObjectMath model is correct there *should be no directed loops* in the graph.

4 Graphical Representation of ObjectMath Models

We suggest a notation for ObjectMath diagrams, shown in Figure 4. The example in textual notation (Figure 2) corresponds to the diagram depicted in Figure 1. This notation is sufficient for static (non-changeable) graphical representation. However, in order to work with such diagrams in an interactive environment a mapping between syntactical rules and operation of the editor should be designed.

The practice of development of interactive environments for diagram editing suggests that there can be three different strategies defining how the syntactic rules are enforced:

Violation impossible: All manipulations with a diagram that can lead to erroneous diagrams are not available in the tool.

Permanent check: The tool includes a special feature for checking diagram correctness at any time during editing. All editing operations that lead to incorrect diagram are canceled, and therefore cannot be applied.

Commit check: The tool checks correctness when editing of a diagram is finished and the diagram is saved or exported. Some rules might be violated unless the diagram is used by some other tool, e.g. a compiler.

All three these types of rules occur in the ObjectMath diagram editor.

```
1 program   ::= model  model-name ";"
               equations
              { definition }
2 definition ::= ( class  class-name formal-parameter-list
               | instance instance-name  [ array-definition ] )
               inheritance
               { part-definition }
               equations
               end ";"
3  Rule: class-names and instance-names are unique within
the model.
4  part-definition    ::= part  part-name [ array-definition ]
                          inheritance  ";"
5  Rule: part-names   are unique within the container,
    i.e. with the class or instance  definition.
6  inheritance     ::= [ inherits
                       class-name actual-parameter-list
                       { "," class-name actual-parameter-list}]
7  Rule: Inherited class name should be defined somewhere else
   in the hierarchy. This class is called a superclass.
8  Rule: If inheritance and composition relations are
   represented as directed edges of a graph they cannot
   form loops (i.e.  directed loops)
9  Rule: number of actual class parameters in the inherits
   clause matches the number of formal class parameters in the
   class definition.
10  array-definition      ::=    "[" name "]"
11  actual-parameter-list ::=    name {"," name}
12  formal-parameter-list ::=    name {"," name}
13  class-name            ::=    name
14  instance-name         ::=    name
15  part-name             ::=    name
16  equations             ::=    ...
A name is an arbitrary identifier.
```

Figure 3: *The grammar and syntax rules of the ObjectMath textual notation.*

4.1 The ”Violation Impossible” Rules

The violation of the following rules (the rule numbers refer to Figure 3) is not possible in the ObjectMath graphical notation used in the diagram editor:

Rule 1: The diagram consists of icons. The order of the definitions does not affect ObjectMath semantics. Therefore the icons can be placed in any order.

Rules 2 and 10: there are class and instance icons; there is an instance array icon.

Rule 4 and 10: there are part icons placed within a class or instance icon. There is a part array icon (same as the instance array icon).

Rule 6: Relations between icons are shown as connection lines. Some lines have arrow heads. The order in which classes are inherited (in case of multiple inheritance) affects the semantics. In the case of conflicts between definitions

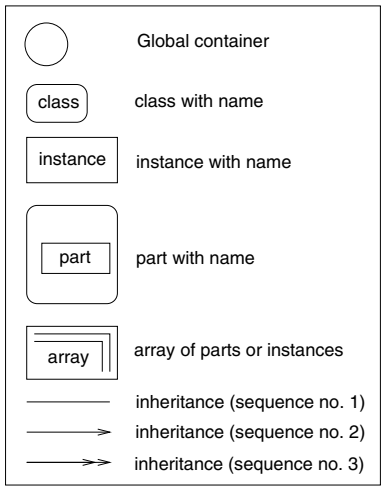


Figure 4: *Graphical notation for ObjectMath class diagrams. The inheritance relations are numbered because the order of classes in case of multiple inheritance affects the program semantics. The container for global objects (*Global container*) is used for two purposes. First it contains global variables, functions and equations which do not belong to any particular instance. Second, the icon of* Global container *is connected to all classes and instances that have no superclasses.*

inherited via two different superclasses, the definitions inherited via the first inheritance override the definitions inherited via the second inheritance relations. A line without arrows is used for single (or the first) inheritance relation. The direction of this relation is not shown in the diagram, but it is obvious in most cases. A line with n arrows is used for the $(n + 1)$-th inheritance relation.

Rule 7: All connection lines represent inheritance relations and such lines must go from some icon to a superclass icon.

Rules 13, 14, 15: The names are shown in the icons.

4.2 Permanently Checked Rules

The operation of the editor includes some checks which enforce the following rules:

Rules 3, 5: Names cannot be repeated in the diagram. Each time a new class, instance, or part is created (or renamed), its name is checked.

Rule 8: No directed loops can be created when diagrams are modified. This rule cannot be easily enforced by graphical notation. Non-directed loops are allowed since multiple inheritance exists. Directed loops are not allowed since

inheritance cannot form a loop[1].

Rules 11, 12: The parameters are not shown in the graphical notation. However, when an icon or a connection line is created or selected for editing, the parameters are displayed in the dialog window.

Before each diagram modification that can potentially lead to violation of rule 8, the new model is checked and the modification is not allowed if the rule is violated. Rules 3, 5, 11, 12 are checked each time the corresponding icon or connection is created or modified. If these rules are violated, the operation is canceled.

4.3 Rules Checked when an Editor Session is Finished

The following rule is checked each time the model is read, saved or sent for compilation:

Rule 9: Comparison of number of actual and formal parameters passed at inheritance. The semantics of class parameter passing in ObjectMath is explained in [1], Section 3.3.

This rule is violated, for instance in the case of the part `front` (see Figure 2). The **`inherits`** clause contains the class `Wheel` with *one* parameter. The number of the parameter names can be changed if the line representing an inheritance relation is selected and the relation is edited. The class `Wheel`, however, has *zero* parameters. This can be changed when the class `Wheel` icon is selected and its parameters are modified. These two places (class `Wheel` properties and the properties of inheritance relation between `Wheel` and `front`) can be edited by the user independently and in arbitrary order. Therefore the violation of this rule cannot be forbidden. Instead, when the model is read, saved, or sent for compilation, a message about this error appears.

5 Operations of ObjectMath diagram editor

The operations that can be performed with the diagram editor are presented in Figure 5.

The major operations that affect the diagram are adding, deleting and moving objects.

5.1 Adding Objects and Relations

If the currently selected object is a class, then another class (a subclass), or an instance (of this class) can be created. An inheritance relation is established. Also, a part can be added to this class.

[1] One way to obey this rule in the graphical interface is to guarantee that all the icons with inheritance relations should be placed so that the inherited class is located *above* a subclass or an instance. Editing diagrams using such editor is, however, quite difficult and is not convenient.

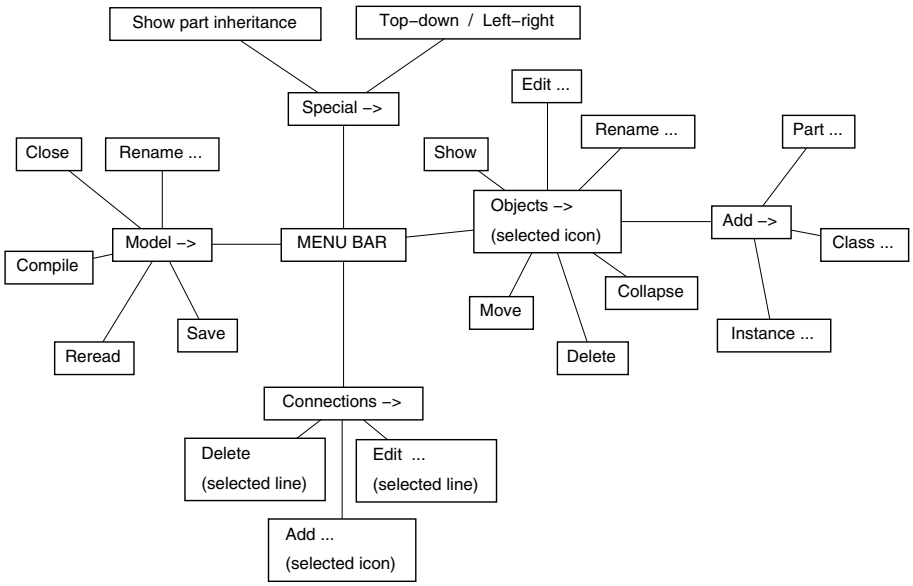


Figure 5: *Menu choices of the ObjectMath class diagram editor. The alternatives leading to new dialogs are marked with ellipsis (. . .)*

If the currently selected object is an instance, a part can be added to this instance.

When a class is created, its formal parameters must be specified. When an instance, a part or a class is created the actual parameters are specified.

To create a new inheritance relation the user selects a subclass or a part, chooses the corresponding menu item, and then selects the parent class.

These operations allow construction of an arbitrary syntactically correct class relationship diagram from the empty diagram.

5.2 Deleting objects

If a class is deleted, all descendants of the class are deleted, too.

When the last left inheritance relation to the superclass is deleted, the subclass is moved to the top level of the hierarchy (i.e. becomes attached to the *Global container*).

When a class or an instance is deleted all its components are deleted, too.

This way an empty diagram (i.e. a diagram with a single *Global container* icon) can be created from an arbitrary diagram.

5.3 Moving Objects

A class or an instance can be moved. First the icon of a class or an instance is selected, then the menu item is chosen, and finally a new parent class is selected.

Also a part can be moved. First the icon of a part is selected, then the menu item is chosen, and finally a new container class is selected.

5.4 Other Operations

There are several other operations in the editor:

Edit A class, instance, part, or connection can be selected and its attributes (parameters, inheritance number, array index) can be edited.

Rename Classes, instances and parts can be renamed.

Show Classes and instances have textual part (i.e. variables, functions, equations) can be displayed in the text editor and modified there.

Collapse When a class is collapsed all its descendants are temporarily hidden.

5.5 Completeness and Correctness

The operations mentioned above have two important properties:

Completeness: For any two syntactically valid class relationship diagrams A and B there exist a sequence of operations that transforms A to B.

This can be done in many different ways, for instance:

- all nodes of A can first be deleted;
- only the *Global container* is left;
- all the class nodes of diagram B can be added in order from the top classes to the bottom classes of the inheritance hierarchy;
- all instances are added;
- all parts are added;
- all additional inheritance relations in case of multiple inheritance are added.

Correctness: If a sequence of editing operations is applied to a syntactically correct diagram, the resulting diagram will be correct.

This is performed by checking the diagrams for violation of the rules as described in section 2.

6 Layout

Each diagram editor should have facility for automatic layout of the icons in a diagram. In this section we discuss the layout of a directed acyclic graph. The

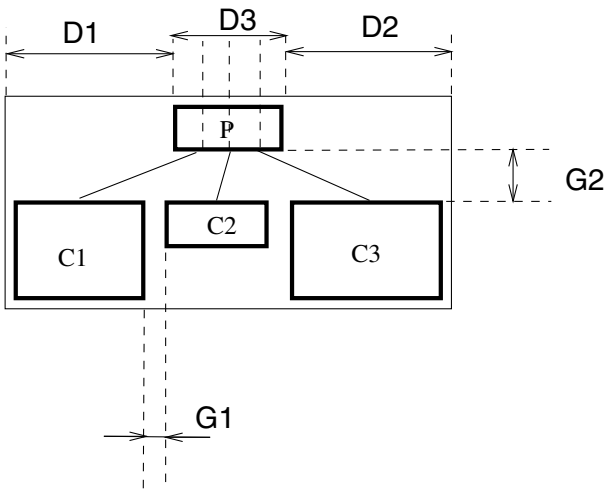


Figure 6: *Layout algorithm for directed acyclic graphs*

ObjectMath diagram is not acyclic, but it has an acyclic subgraph. We also discuss how users modify the default layout.

The editor uses an automatic layout algorithm for acyclic directed graphs. In this algorithm the root of the tree is placed at the middle on the top border of the diagram window. The area reserved for the parent node (`P` in Figure 6) has the same width as the sum of all the widths of the child nodes (`C1`, `C2`, `C3`) and the gaps of width `G1` between those. The child nodes are placed below the parent node at a distance of `G2` from each other. The parent node is placed in the middle of the space above the child nodes so that `D1=D2`.

The lines from the parent node `P` to the child nodes start at evenly spread points (the width `D3` is divided by 4).

The graph of relations in ObjectMath diagrams is not acyclic since multiple inheritance may occur and since parts inherit some classes. In the diagram (see Figure 1) a cycle consisting of six nodes (*Global Container*, `Body`, `Cylinder`, `Wheel`, `Metal`, `Material`) occurs. For layout computation we find an acyclic subgraph. For each class or instance the algorithm chooses the deepest superclass. Only this inheritance relation is chosen. The depth of the class is the depth of its deepest superclass plus one. The depth of the *Global container* is zero. The acyclic subgraph contains all the nodes of the original diagram, but not all connections are included.

When the layout of nodes for an acyclic graph is computed, all the connections of the diagram are placed between the nodes. The inheritance almost always corresponds to connectors placed in top-down direction.

The default layout can be used as a convenient starting point for the user to modify. Any node can be shifted, together (or without) its descendants. The displacement between the default and modified node position is stored in a file (as comments) and this information is preserved from one editing session to the next.

7 Conclusions

Given a grammar for textual representation of class relationships and syntax rules we have developed a mapping from textual to graphical, representation. A more complicated task has been to design rules for diagram editing consistent with the syntactic rules. Three types of rules were identified, and they are explained in the report. A well designed mapping will have most rules belonging to the first group that does not require additional checks during the operation of the diagram editor. The rules of the second group may lead to difficulties for the user. The rules of the third group make difficulties both for the user and for the integrated environment (e.g. the diagram cannot be saved unless the errors are fixed).

Based on this mapping, a class diagram editor with facilities for multiple inheritance and composition relationship was developed. These two features differ this editor from a number of tree diagram editors which are used traditionally. A simple but efficient and user-friendly automatic diagram layout facility is implemented in the editor.

References

[1] Peter Fritzson, Vadim Engelson, Lars Viklund. Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment, In *Proceedings of the Conference on Design and Implementation of Symbolic Computation Systems (DISCO 93)*, vol. 722 of Lecture Notes in Computer Science, pp. 145–160. Springer-Verlag, 1993.

[2] *Mathematica*, Wolfram Research Inc., http://www.wolfram.com

Paper 3

Automatic generation of user interfaces from data structure specifications and object-oriented application models

Vadim Engelson, Dag Fritzson* and Peter Fritzson†

Abstract

Applications in scientific computing operate with data of complex structure and graphical tools for data editing, browsing and visualization are necessary.

Most approaches to generating user interfaces provide some interactive layout facility together with a specialized language for describing user interaction. Realistic automated generation approaches are largely lacking, especially for applications in the area of scientific computing.

This paper presents two approaches to automatically generating user interfaces (that include forms, pull-down menus and pop-up windows) from specifications.

The first is a semi-automatic approach that uses information from object-oriented mathematical models, together with a set of predefined elementary types and manually supplied layout and grouping information. This system is currently in industrial use. A disadvantage is that some manual changes need to be made after each update of the model.

Within the second approach we have designed a tool, PDGen (Persistence and Display Generator), that automatically creates a graphical user interface and persistence routines from the declarations of data structures used in the application (e.g., C++ class declarations). This largely eliminates the manual update problem. The attributes of the generated graphical user interface can be altered.

Now structuring and grouping information is automatically extracted from the object-oriented mathematical model and transferred to PDGen.

This is one of very few existing practical systems for automatically generating user interfaces from type declarations and related object-oriented structure information.

Published in Proceedings of European Conference on Object-Oriented Programming (ECOOP96), Linz, Austria, 8-12 July 1996, Pierre Cointe (ed.); Lecture Notes in Computer Science, vol. 1098, Springer-Verlag, pp. 114-141, ISSN 0302-9743, ISBN 3-540-61439-7

*SKF ERC B.V., Postbus 2350, 3430 DT Nieuwegein, The Netherlands, adsdtf@skferc.nl

†Dept. of Computing and Information Science, Linköping University, S-58183, Linköping, Sweden, {vaden,petfr}@ida.liu.se

1 Introduction

Almost all applications include some kind of user interface. Graphical user interfaces (GUI) provide the opportunity to control an application's execution, to modify the input data and to inspect the results of computations.

Application programs have different data structures. Each application domain puts special requirements on visual presentation of data. Therefore, graphical interfaces are traditionally designed individually for each application.

The following properties are expected from applications with graphical user interfaces:

- The data must be presented to the user in a well-structured way. The graphical user interface should be consistent with the computational part of the application (for example, elements of the graphical user interface for data input should correspond to components of the application data).
- The user interface should satisfy style guidelines, conventions and standards. The compromise between large amounts of information and limited screen space can be achieved if the graphical user interface allows the user to choose only interesting information and ignore all else.
- The user interface software should be portable and not be dependent on a specific operating system or compiler.
- Entered data should be persistent: it should be possible to store entered data outside the program memory and reload it again later.

For realistic applications the design and implementation of a graphical user interface often become rather laborious, expensive and error-prone. Currently available toolkits are very powerful. Unfortunately, they are also very complicated and not user-friendly enough. In order to obtain some result the programmer often has to take too many implementation details into account. The high cost of implementing user interfaces can be partly reduced by the use of *user interface generation tools*. Such tools usually include a WYSIWYG layout definition tool that helps the programmer to design the layout of windows, menus, buttons and other user interface items. The graphical user interface code is generated automatically.

However, every time the application code is updated or the layout is changed, the interface code between them has to be updated manually.

1.1 User interface generation based on data declarations

In this paper we propose a different approach, based on the automated generation of user interfaces from data structure information. As a preliminary we present some terminology.

Data structures in traditional languages (such as Pascal or C) are described by variable and type declarations. In object-oriented languages (e.g. C++) data structures are defined using classes, objects and relations (inheritance, part-of) between objects.

The *structure of a graphical user interface* can be described in terms of graphical elements such as windows, menus, dialog boxes, frames, text editing boxes, help texts, etc., and layouts that define how these elements are placed on the screen.

In which *context* is the interface used ? The main purpose of a graphical user interface is to let the user inspect and modify some data. The input data can be edited by a stand-alone graphical tool, saved in file and then loaded by the computing application. The output data can be saved by the application and inspected by a separate tool. The application may suspend computations, initiate a graphical interface in order to allow data editing, and then resume the computations again. *We consider graphical interfaces that can be used in all these cases.*

The data has some structure and it is used to control the application functionality. Therefore the structure of the graphical user interface should be similar to the structure of the application data.

Typically there is an implicit or explicit correspondence between the structure of a program and the structure of data. On the other hand, there is a correspondence between the structure of the interface and data structures of the program. This means that the way the programmer perceives the structure of the implementation is close enough to the way the end-user perceives the structure of the application area.

The basic idea of our approach is to *generate the graphical user interface automatically from the application data structures*.

The similarity between the data structures and the structure of the graphical user interface is characteristic for a wide spectrum of applications, including simulation tools and information systems.

Data persistence is a generic property that includes saving data structures on permanent storage such as a file system and being able to restore this data next time the application is executed. To implement persistence, we need routines that can save or load all the application data (or some part of the data). *Such persistence routines can be generated automatically from the application data structures*.

Automatic support for data persistence as well as generation of graphical user interfaces will allow designers to concentrate on the main goals of the applications rather than on mundane tasks such as implementing a graphical user interface and input/output.

We applied the method of generating user interfaces from data structure declarations to two object-oriented languages: ObjectMath (an object-oriented extension of Mathematica [Wolfram91]) and C++.

In Section 1.1 we have discussed some reasons and motivation for the design of a user interface generator according to these principles.

The rest of the paper is organized as follows:

First we describe relevant features of ObjectMath, an environment for scientific

computing (Section 2.1) and consider a *semi-automatic* approach to the creation of user interfaces from application data structures, which has also been tested in industrial applications. A new *automatic* graphical user interface generation approach is based on our PDGen tool (Section 3) that automatically generates persistence and graphical user interface code from given data-type declarations. This tool is applied to ObjectMath models.

In Section 4 we describe how the PDGen tool can be applied to ObjectMath models.

Section 5 discusses related work on persistence and display generation and we conclude with proposals for future work. More details can be found in [PDGen96, E96].

2 The Semi-automatic GUI Generating System

2.1 The ObjectMath Environment

Applications in scientific computing are often characterized by heavy numerical computations, as well as large amounts of numerical data for input and output.

The data often have a complicated structure including objects with fields of various types, vectors and multidimensional arrays. This structure often changes during the course of program design.

An important application area in scientific computing is the simulation of various mechanical, chemical and electrical systems. These applications can be described by mathematical models of the physical systems to be simulated. Additionally, routines for numerical solution systems of equations are needed, as well as routines for input/output and routines and tools for user interfaces.

The process of manually translating mathematical models to numerical simulation programs in C or Fortran is both time-consuming and error prone. Therefore, a high-level programming environment for scientific computing, ObjectMath [Fritzson95, Viklund95, Fritzson93], has been developed that supports the semi-automatic generation of application code from object-oriented mathematical models.

The ObjectMath programming environment has been applied to realistic problems in mechanical analysis. ObjectMath class libraries describing coordinate transformations and contact forces have been developed. They are used for mathematical modeling of rolling bearings by our industrial partner, SKF Engineering and Research Center.

In ObjectMath formulae and equations can be written in notation that is very similar to conventional mathematics. The ObjectMath language is an object-oriented extension of the Mathematica computer algebra language, in a similar way as C++ is an extension to C.

The ObjectMath language includes object-oriented structuring facilities such as *classes*, *instances*, single and multiple *inheritance* (for reuse), and the *part-of* relation (to compose new classes from existing ones).

2.2 The simulation environment for ObjectMath models

First, an *ObjectMath model* is specified with the help of a class relationship editor and class text editor. The *ObjectMath code generator* generates parallel or sequential programs for systems of equations expressed in ObjectMath. Typically a system of ordinary differential equations is considered.

The generated code is linked with model-independent run-time libraries. The *executable code* requires a large number of input values (such as start values, limitations, model geometry and conditions, solver parameters) in order to start a simulation.

The *input data editor* is designed for input data inspection and update. It has a window-based graphical interface for ObjectMath variable editing and can load and save a file with variable values. In this paper we present two graphical user interface generation systems that can create an input data editor from model specifications. The first system is described here, the second in Section 4.

The simulation program reads the data prepared by the input data editor and computes a large amount of output data for every simulated time step.

This data can be explored with the help of an *output data browser*. This browser can create graphs that illustrate how the variables change during the simulation.

The *animation tool* reads the output data step by step and shows the model geometry in motion.

2.3 An ObjectMath example: a Bike model

In this section we present an ObjectMath model example, a mechanical model of a bicycle (Fig. 1) in order to explain the relations between classes and instances in ObjectMath[1].

Every model specification consists of *classes* and *instances*. The textual part of classes and instances contain variable declarations, formulae and equations. This is the way the formulae and equations related to the same phenomenon are grouped. Classes and instances inherit variables, formulae and equations from one or several (multiple inheritance) classes. The classes serve as templates for instances.

The class `Bicycle` in Fig. 1 inherits all variables, equations and formulae from the class `Body`. The instance `bike` inherits everything from the class `Bicycle`. An instance or a class can also contain its *own* variable declarations and formulae. Every instance is created *statically* and its variables (both its own and inherited ones) can be referenced in formulae and equations of other classes and instances.

A `Bicycle` consists of three *parts* in this model: the `front` wheel, the `rear` wheel and the `frame` consisting of several `tubes`. The number of `tubes` is equal to the value of some variable, in our example it is `framesize` (this is not shown in the diagram).

[1] We discuss the constructs relevant for graphical user interface generation only.

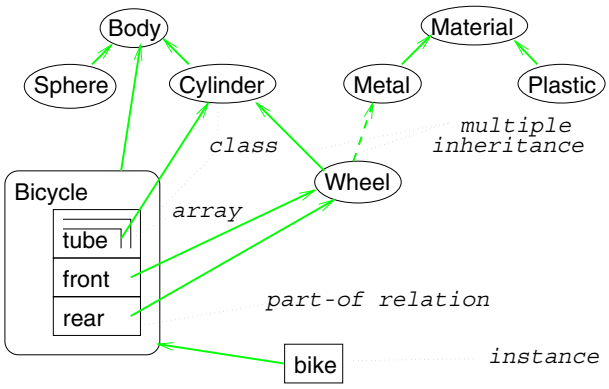


Figure 1: *An ObjectMath class diagram for a bike model. Arrows denote single or multiple inheritance. The* `bike` *instance contains the parts* `tube` *(array of tubes),* `front` *and* `rear`*, which inherit from the classes* `Cylinder` *and* `Wheel`*, respectively. Such diagrams are editable with a graphical class relationship editor.*

2.4 Variables and built-in data types

Let us assume that there are several ObjectMath variables[2] declared in the class definitions:

```
In Body:    Declare [angle, "doubleVec3", "rad", uII]
In Cylinder:Declare [radius, "double", "m", uII]
In Wheel:   Declare [pressure, "double", "H/m^2",uII]
In Bicycle:Declare [framesize," int", "-", uII]
```

In the general form variable name, type, physical unit and persistence status are specified:

`Declare[`*name*`,` *type-name*`,` `"`*unit*`",(uII|uOO|uL)].`

Types. In ObjectMath there is a fixed set of twenty primitive data types that can be used for variables in the model. Some of the types have complex structure and may contain up to 100 double precision real numbers, integers and strings.

A variable of type `double` has a double precision floating value. The type `doubleVec3` is a 3-element vector of `double`.

Units. A string such as `"H/m^2"` contains the name of the physical unit of the value this variable represents. This unit name is used as part of the prompting information for the relevant input field in the input data editor that is generated from the declarations above.

[2]This declaration syntax is for ObjectMath version 3.0. The latest version 4.0, fall 1995, has a different declaration syntax.

Persistence status. The variable declarations provide the persistence information: whether a variable should be initialized by the input data editor, should be output and stored as a computed result, or is simply a local variable for intermediate results.

Here `uII` means "to input from the input file", `uOO` means "to output to the output file", and `uL` means "local variable, neither input, nor output".

From this information a window with text input boxes (Fig. 2) is generated. The variable instance identifier, text input area for value editing and "units" are shown for each variable component.

2.5 Generation of input data editor

The *basic idea* of this approach is that part of the code necessary for graphical user interface creation is *automatically* generated from ObjectMath variable declarations. Then the *layout* information is manually inserted into this code.

In order to create the input data editor we have to create the hierarchy of windows and variables; this hierarchy is created half-automatically and combines display routines provided that create widgets for every ObjectMath variable type.

When model specification code is analyzed, the names of variables such as `angle`, `radius` and `pressure`, are converted to unique names (within the model) by adding prefixes (part and object names). This is the list of *all* the variables available in this model, where the notation *name* `[`*n*`]` denotes an array with *n* elements:

```
bike‘front‘radius
bike‘front‘pressure
bike‘front‘angle
bike‘rear‘radius
bike‘rear‘angle
bike‘rear‘pressure
bike‘tube‘radius[bike‘framesize]
bike‘tube‘angle[bike‘framesize]
bike‘framesize
```

A specially designed filter reads the model specification and generates a comma-separated list of function calls:

```
var_double_array("bike‘tube‘radius","m"),
var_doubleVec3_array("bike‘tube‘angle","rad"),
var_double("bike‘front‘radius","m"),
...  ...
```

The calls of `var_...()` functions above register the variables as members of the list of relevant variables and return a frame handle which is used for constructing corresponding windows. The rest of the code needed in order to display these

variables in a separate window (see Fig. 2) is inserted *manually* (manual part is shown in *italic* font):

```
make_dialog(
    layout_vertical(
      layout_horizontal(
        layout_vertical(
          var_double("bike`front`radius","m"),
          var_doubleVec3("bike`front`angle","rad"),
          var_double("bike`front`pressure","H/m^2")),
        layout_vertical(
          var_double("bike`rear`radius","m"),
          var_doubleVec3("bike`rear`angle","rad"),
          var_double("bike`rear`pressure","H/m^2"))
        )
      layout_frame(
        layout_vertical(
          var_int("framesize","-"),
          array("framesize",
            layout_vertical(
              var_doubleVec3_array("bike`tube`angle","rad"),
              var_double_array("bike`tube`radius","m")
))))));
```

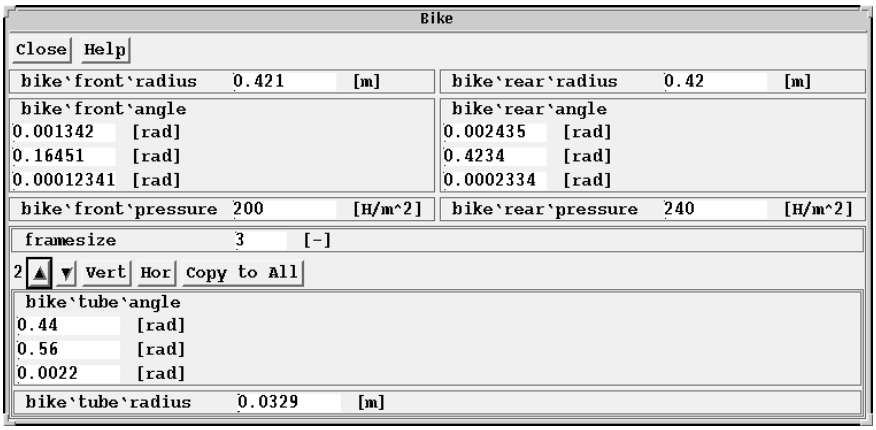


Figure 2: *Example of variable display in the input data editor.*

2.6 Presentation of arrays.

Two variables (`bike`tube`angle` and `bike`tube`radius`) represented in the example (see Fig. 2) are arrays of `doubleVec3` and `double`. They have the

same length; therefore they may be grouped together. These arrays share common control buttons (`Vert`, `Hor`, `Copy to all` etc.) in the upper part of their frame. (It is also possible to build displays where every array has a separate control panel.)

As shown in Fig. 2 only one element (currently the 2nd element) of each array is visible, as indicated by the label "`2`" in the upper left corner. This is the compact presentation of the array. The buttons with the triangles ("`Up`" and "`Down`") switch the current element to the previous or the next, respectively. Then the label may change to "`3`" ("`Up`") or "`1`" ("`Down`"). The button "`Copy to all`" copies all the values from the visible element of the arrays to all other elements.

The buttons "`Vert`" and "`Hor`" change the presentation of the array: they spread its elements vertically or horizontally, respectively. Then the button "`Collapse`" appears that changes the presentation back to compact form.

2.7 Frame hierarchy definition functions

Every application window contains a hierarchy of *frames*. Each frame is a rectangular area that contains graphical user interface elements (widgets) such as labels, text input boxes, buttons, as well as compositions of other frames in the vertical or horizontal direction. A number of functions are needed in order to specify this hierarchy of frames:

- The function `make_dialog(`*frame*`)` specifies the top frame of the window.
- The function *frame*`=layout_vertical(`$frame_1$`,`$frame_2$`,...,`$frame_n$`)` specifies that the frames $frame_1, frame_2, ..., frame_n$ are allocated in a vertical direction.
- The function `layout_horizontal` allocates them in the horizontal direction.
- The function *frame*`=array("`*bound-variable*`",` $frame_1$`)` specifies that length of all arrays within $frame_1$ is equal to the current value of the variable *bound-variable* and they are controlled all together by the buttons in the upper part of the frame. The control buttons for array variables can change the index of the currently displayed element (see Figure 2).

There are several functions for additional help texts and decorations.

- The function *frame*`=layout_frame(`$frame_1$`)` draws a rectangle around $frame_1$;
- The function *frame*`=layout_label("`*text*`")` specifies a label containing the *text* string.

The description this hierarchy is stored as a tree. When necessary, the tree is traversed, relevant Motif API functions are invoked, and the windows are displayed on the screen.

2.8 Description of variables

For every displayed variable a function for a corresponding data type should be called. These calls are automatically generated from the list of model variables. There is a separate function for each data type used in the ObjectMath language: `var_double`, `var_int`, `var_doubleVec3` and all others (totally, twenty) ObjectMath basic data types.

For example, *frame*=`var_double("bike'front'radius","m")` specifies that a text input box is constructed for the variable `bike'front'radius` of type `double` and that the physical unit is `"m"`.

Every such call registers a variable and arranges for the value of this variable to be displayed at an appropriate place in the layout. For every type a certain specific layout has been designed and hard-coded. For example, for the type `doubleVec3` (a structure with three double values) the layout is three vertically aligned text input boxes. Arbitrary `double` values can be entered here. For `integer`, `double` and `string` variables the layout is a single input box with the variable name to the left and the unit to the right (see Fig. 2). Arbitrary `integer`, `double` and `string` expressions can be entered into the input boxes respectively. .

Persistence. When the button `Save` is pressed, the persistence function is called and all the registered variables are written to the input data file. When the button `Load` is pressed all the variables in the list receive their values from the input data file. Both the input data editor and the application program should register the variables with the same name and type.

2.9 Evaluation of the first generation system

If the ObjectMath model changes, the graphical user interface programmer has two ways to solve the update problem. If many changes are introduced, the graphical user interface code should be generated again and the programmer has to insert the layout functions manually. If the changes are small and local (such as renaming some variables), the variable registration function calls (`var_...(...)`) should be manually updated.

The first generation system described so far in this paper has several *disadvantages*:

The update problem. If the model is changed, the new code that is automatically generated from the variable list must be manually merged with the layout description. Every small change in the list of variables from the ObjectMath model may lead to inconsistency between the generated application and the input data editor. Therefore the inherent flexibility of the ObjectMath environment cannot be used to full advantage.

Insufficiency of the basic type set. Only a limited number of basic data types are supported. These data types are either primitive ones or are specially designed for a particular application domain. There is no way to specify other types than these and there are no new type declaration constructs. The persistence routines and the layout routines are designed for the fixed set of types only.

Variable grouping. There is no automatic graphical user interface generation for distributing variables between different windows. Moreover, there is no automatic generation of the menu structure. However the structure of the model (i.e. the names of classes, objects and parts) can be used for this purpose.

Practical application of the system has also shown its *positive* features.

The first generation system has proved quite effective in producing practical user interfaces for specialized application domains such as bearing simulation. Recently SKF ERC researchers used the system to produce user interfaces for 6 new variants of similar bearing models requiring only 3-4 days of work. The difference between the variable sets in these models were rather limited and all the adjustments of the graphical interface for the input data editor were done manually.

3 The Persistence and Display Generating tool (PDGen)

The basic idea of PDGen is that display layout for every data item exactly corresponds (by default) to the type structure of this item.

Through the *display* for a given variable the user can inspect and update all the data items that can be reached from the variable by recursively traversing its structure. In the same way, the *persistence* routines save and load all the data items that can be reached from a given variable by recursively traversing its structure.

Traversing all of a complex data structure is a non-trivial task if we want to provide this automatically. Special complications arise in languages with pointers and dynamic data structures. Code necessary for this purpose can be automatically generated from data type declarations of the variables we are going to traverse.

We primarily orient PDGen to handling C++ data types. This tool can analyze almost any C++ data type and class declarations and add graphical user interface and persistence routines to an *arbitrary* C++ program. The manual efforts necessary for this are minimal.

The creation of the PDGen tool has been inspired by the PGen (see Section 5.1) approach from which we cite Walter Tichy et al.:

> *The class and type declarations can be used to generate browsers and editors. For instance, a class variable can be presented as a dialog box that contains sub-windows for all members to be inspected or edited. Pointers could be drawn as arrows to other variables. [...] The browsers and editors could be used to inspect or modify persistent*

data on files. More importantly, they could become the default graphical interfaces for all applications. The difference with other interface construction tools is that they require absolutely no programming. Debuggers are another application area. [Tichy94]

In Section 3.1 a graphical user interface generation example is given and in Section 3.2 display generation for every C++ data type is presented . In Section 3.4 we discuss window display issues; the generation process is analyzed step by step in Section 3.5 and the use of the generated code is discussed in Sections 3.6 and 3.7.

The tool is based on the C++ language (Section 5.3) and the Tcl/Tk toolkit [Ou94].

3.1 Example of graphical user interface generation

Let us consider some type declarations that can appear in a header file (see Fig. 3(a)) of some application.

The PDGen tool analyzes these data type declarations, recognizes the C++ class hierarchy, and generates necessary code for creation of a graphical interface. If the application calls the function `show(bike)` (see Fig. 3(b)) then the dialog window shown in Fig. 3(c) appears on the screen. The specification of physical units (`rad`, `H/m^2`, `m`) is performed with the help of an attribute specification script (see Section 3.10).

All the data items that belong to `bike` are shown and they are available for editing. This example illustrates the display for *classes*, *arrays* and elementary data items of types `int` and `double`.

The array `tube` is shown at the bottom part of the window. The user can press the buttons `"+"` and `"-"` in order to change the index of the currently displayed element of the array `bike.tube`. In the window shown in the picture the index is equal to 2, i.e. `bike.tube[2]` is displayed.

3.2 Graphical presentation of variables

Every window may contain one or several variables. The graphical presentation of every variable depends on its type and it is combined from graphical presentations of its components.

**Types `char`, `char*` and `char[`*n*`].** The types `char*` and `char[`*n*`]` are typically used for `0`-terminated strings. A text input box is constructed for such variables and the string can be edited. Scrolling of the text is always provided so that character strings longer than the text box can be inspected and edited.

The display for variables of type `char` is similar to `char[1]`.

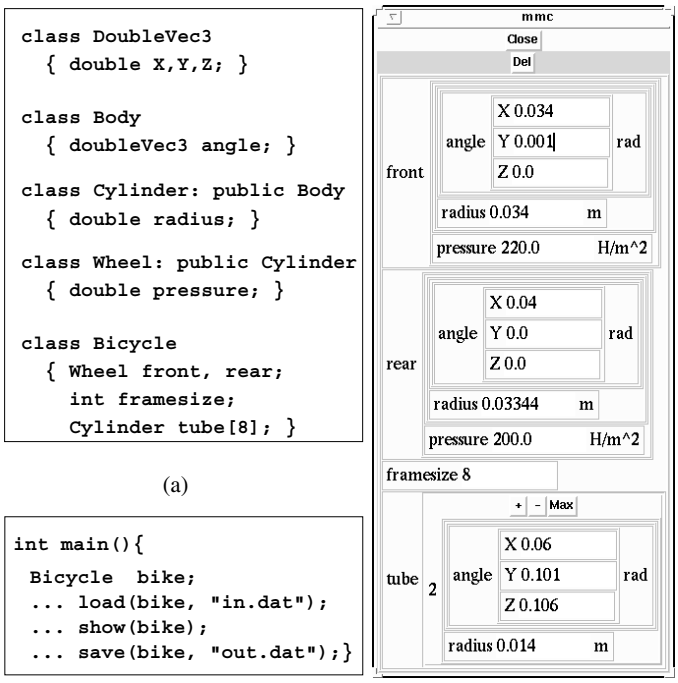


Figure 3: *(a) Data type declarations. (b) Function call. (c) The window for editing the variable* `bike`.

Types `integer`**,** `float` **and** `double`**.** Variables of these types are displayed as text editing boxes (see Fig. 3(c)). Only numbers or expressions (consisting of numbers and arithmetical operations) can be entered. The range of permitted values can be specified (see Section 3.10).

Structures and classes. These are represented as horizontal or vertical[3] combinations of the components. The names of the data members of a structure or class are used as labels that appear to the left of corresponding components (see Fig. 3(c)).

Pointers. In our initial approach a pointer variable is represented by the referenced variable if the address is not `NULL`. There is a button `Delete` that deallocates the memory and sets the pointer to `NULL`. If the address is `NULL`, then there is a `New` button that creates a new variable in the dynamic memory, initializes it if it is an object and sets the correct value for the pointer variable.

[3]In order to choose between horizontal and vertical combinations we use some heuristics. For example, we choose one that makes the resulting frame more similar to a square i.e. the ratio between the height and the width of the frame is closer to 1. With the help of customization options this default layout can be altered.

Let us specify the class `Tree`:

```
class Tree
 {  Tree * right;
    int elem;
    Tree * left;
 };
```

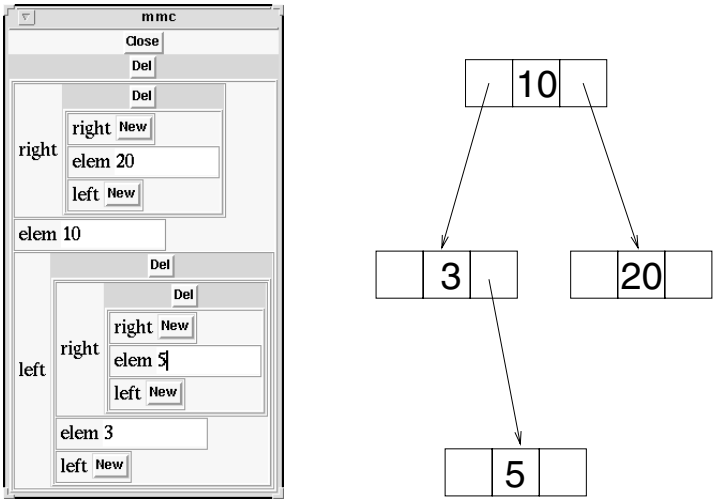


Figure 4: *The window for editing the pointer structure of type* `Tree*` *and memory diagram of this structure.*

A variable of class `Tree` is displayed as a structure with three components (`right`, `elem` and `left`). We can also display a variable that contains a pointer to `Tree`. A variable of type `Tree*` is visualized as shown in Fig. 4.

This is a simple and quite straightforward approach if we are not concerned about the cases when two or more pointers refer to the same address.

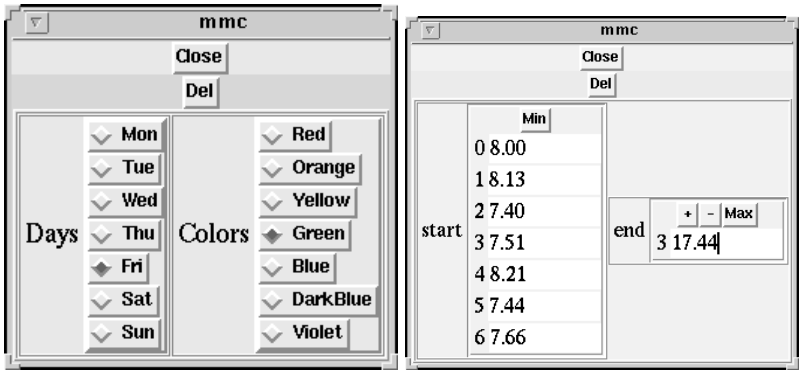
In the *alternative* representation every dynamically allocated object (that has two or more references) is shown as a separate sub-window and arrows are drawn from the pointers to these objects in order to indicate the references.

Enumeration. The enumerations are represented as a group of radio buttons (or, as an alternative, as a pop-up menu). Enumerator names are written beside the buttons and only one of them may be selected at a time.

An object of class `Foo` is shown in Fig. 5(a).

```
enum weekday
 { Mon, Tue, Wed, Thu, Fri, Sat, Sun} ;
enum colors
 { Red, Orange, Yellow, Green, Blue, DarkBlue, Violet};
```

```
class Foo
   { weekday Days;
     colors  Colors;};
```



(a) (b)

Figure 5: *The windows for editing variable with (a) enumerators and (b) arrays.*

One-dimensional array. Fig. 5(b) shows how an object of class `Foo` is visualized.

```
class Foo
{ double start [7];
  double end[7];
};
```

The elements of `Foo::start` are shown in the *complete presentation*, i.e. all of them are available for browsing. In the *compact presentation* of the array `Foo::end` only one element (currently it is the element `end[3]`) is shown at a time. By using the buttons `+` and `-` we can increase or decrease the index of the currently visible element. The button `Max` switches the display to the *complete presentation*; the `Min` button changes the display back to the compact presentation.

An array of dimension larger than one is represented as a combination of one-dimensional arrays. This is not very convenient for browsing. A special browser [FWHSS96] has been designed for two-dimensional arrays. We are working on an universal array browser for an unlimited number of dimensions.

A special interface is provided for dynamically allocated arrays. These can grow and shrink dynamically. For this purpose the buttons `Insert` (insert new element after the current one) and `Remove` (remove the current element) are added above the presentation of the array values. This option has some limitations and requires some additions in the description of data structures: the dynamic array (A) must belong to some class and an additional integer variable (A`_length`) should store its length:

```
class Test
 { Element * array_foo;
   int     array_foo_length; }
```

3.3 PDGen restrictions

There are some restrictions in the PDGen system that are partly caused by the restrictions of the PGen tool.

- References, constants, bit fields, unions, pointers to functions, pointers to members, `void*` pointers are skipped and ignored, because they cannot be persistent or are compiler-dependent, or there is no sense in keeping them persistent.
- Virtual base classes are supported for persistence only.
- Pointers to memory inside an object are supported for persistence only.

3.4 Hiding and detaching windows

The number of data items that can be displayed on the screen simultaneously is limited. Normally we cannot show a hierarchical layout of more than approximately one hundred text editing fields. We propose a window handling scheme where every displayed data item can be in three states (see Fig. 6):

- **hidden**: only a button with the data item identifier is shown in its default place;
- **normal**: the data item is displayed as usual in its default place;
- **detached**: the button with data item identifier is shown in its default place; the item is shown in a separate (top-level) window.

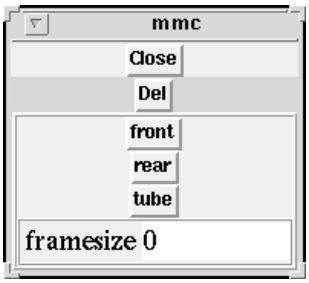


Figure 6: *The variables* `front`, `rear` *and* `tube` *are hidden in the window for editing the variable* `bike`.

Switching between normal, hidden and detached state is performed by the mouse buttons. In each case the user has to click on the name of the item.

The *default* status is "hidden" for all non-elementary data elements and "normal" for elementary ones. The user can specify the default status with the help of attributes discussed in Section 3.10.

The buttons have the same function as pull-down menu items. The end-user has complete control over the information layout on the screen and there is no problem with the windows occupying all the display space. This way the user can hide unnecessary information and select interesting data for display in separate windows. Since the buttons have almost the same behaviour as pull-down menus, this approach is rather close to graphical user interface standards and conventions.

3.5 Data type analysis and code generation.

This section discusses the generation process in detail, phase by phase.

The stages of graphical user interface generation are shown in Fig 7.

The PDGen tool reuses some ideas and essential code fragments from the PGen tool [Tichy94, PGen94, Paulisch90].

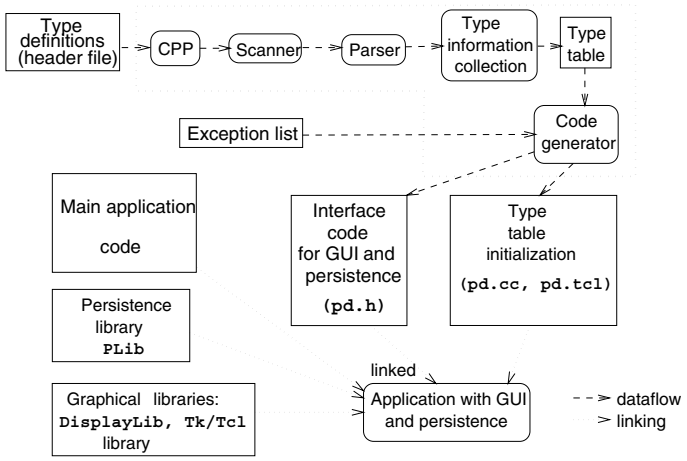


Figure 7: *Phases of graphical user interface generation from C++ code and generation results.*

The basic source of the graphical user interface generation is a file with data type declarations. In C++ applications it involves one (or several) *header* files.

Parsing. The file is preprocessed by the standard C preprocessor `cpp` and analyzed by the C++ scanner and parser.

Together with the PGen analyzer (see Section 5.1) we reuse the C++ grammar parser with semantic actions for syntax tree construction.

The syntax tree contains nodes of different kinds and references between them. The collection phase traverses all the nodes describing `typedef`, `enum`, `struct` and `class` declarations and produces the data-type table.

The syntax tree contains all the syntactical elements that appear in the input file. For our purposes we use `typedef`, `enum`, `struct` and `class` declarations only. In the `class` declarations we are interested in data members and constructors only.

Data type table. This table contains the names of all types, the names and types of class data members, inheritance information, the element type and the length of arrays, as well as information on elementary data types.

The analyzer assigns a unique type number (used for references); the numbers are assigned in increasing order: the first few numbers are reserved for fundamental types such as `int` and `double`.

The exception list. This list is optional and helps to prevent inclusion of unnecessary classes and data members to the type table.

Generation. The generation phase creates **type information** and **overloaded access routines**. All the generated code belongs to the class `PD` (**P**ersistence and **D**isplay) defined in the header file `pd.h` , with member functions defined in the file `pd.cc`

Generation of data type information. The `PDGen` code generator writes to `pd.cc` a code fragment that initializes the data type table.

For the example given above (Fig. 3(a)) a fragment of relevant code is:

```
initSimpleType(3,"int",sizeof(int));
....
initClassType(33,"Wheel",sizeof(Wheel),0);
initBaseClass(33,32,0, "Cylinder");
initMember(33,12,offsetof(Wheel,pressure), "pressure");

initClassType(34,"Bicycle",sizeof(Bicycle),0);
initMember(34,33,offsetof(Bicycle,front),  "front");
initMember(34,33,offsetof(Bicycle,rear),   "rear");
initMember(34, 3,offsetof(Bicycle,framesize),"framesize");
initMember(34,35,offsetof(Bicycle,tube),   "tube");

initArrayType(35,"Cylinder[8]",sizeof(Cylinder[8]),0,8);
```

The standard C macro `offsetof(Bicycle,rear)` calculates the offset (position) of data member `rear` within objects of class `Bicycle`[4].

This code fragment is later compiled and linked with the application. When the application starts, the data type table is initialized. This table is available to the persistence and display routines at the run time. When the data are saved,

[4]This approach is more portable and safe than to sum up the sizes of every data member.

loaded, or shown, appropriate routines use this table in order to recursively traverse the data structures.

Generation of overloaded access routines. For every data type or class T that is defined in the header file the appropriate instances of the overloaded functions `PD::show(`T`&p)`, `PD::load(`T`&p)`, `PD::store(`T`&p)` are constructed.

These functions can take the variable of type T as an argument.

3.6 Input and output procedures

When the functions `load` and `store` are called, a variable of the corresponding data type is passed as a parameter.

The function `load` reads the variable value from the file and restores it in the memory. The function `store` saves the value on disk.

These functions traverse the application data that can be reached from the passed argument variable by recursively following data members, including pointer references. Every step of this process is controlled by the data type table. When data items of an elementary type are reached, the functions load/store the data in some format (textual or binary); see Section 3.8 where formats are discussed.

When the data is *loaded*, memory is dynamically allocated if a pointer variable is visited and its value is not *NULL*. When memory is allocated for class instances, the class constructor is called without parameters. It is assumed that every class has a constructor without parameters.

3.7 Data display procedure

The function `show` activated from the application program displays the required variable.

For data display we have designed a universal data browser which can show and edit the data when the data-type table is given. We use the `Tcl/Tk` graphical library [Ou94]. First, the C++ variables are associated with Tcl variables. It produces the following effect: if a Tcl variable changes, the C++ data change automatically. If a Tcl variable value is requested, then it is taken from the C++ variable.

We recursively traverse all data members, including pointer references. The algorithm that builds the window as a hierarchy of frames recursively traverses Tcl variables with the help of the data type table (Tcl script `pd.tcl`) which is generated automatically.

When some value is *updated* by the user, the corresponding Tcl and C++ variables are automatically updated; when it is updated by the application, its text presentation is changed, too.

If necessary (i.e. when the `New` button is pressed in the display of a pointer variable), memory is allocated and a class instance is initialized by the constructor.

3.8 Data storage formats

Complex data items are traversed recursively when loaded or stored. The original PLib library includes `load` and `store` routines for two machine-independent formats, ASCII and XDR [SUN90]. XDR is a data representation format used in remote procedure call. These formats are not self-describing formats, i.e., there is no possibility of discovering mismatches between the loaded data and the program data structures.

In the PDGen tool we extend this format by simply adding the type table information. When the `store` procedure writes some data, it also writes the type table. When the `load` procedure reads data from file, it also reads the type table and verifies that it is identical to the original one (for all data types that actually appear in the loaded file).

Difficulties can arise if old data are `load`ed by a program with new data structure. In our approach some basic data scheme correction is provided. If the old data contain classes with permuted order of data members, the correction works automatically. Extra members in the old data are ignored, the missing ones are not initialized. Finally, the user can explicitly specify the old and the new name for *renamed* class data members and renamed class names in the exception list.

3.9 Universal browser design

Since the data table is stored together with data (i.e. in a self-describing data format), a stand-alone universal browser can easily be designed. This is one of directions of our future work.

This browser automatically adapts the interface to the structure of loaded data. It works independently on underlying C++ data type declaration files and can browse and edit a file with arbitrary data structures if it is prepared by the PDGen persistence routines or by the universal browser.

It should be noted that there will be limitations for dynamic memory allocation during editing, because the C++ code (where necessary constructors without parameters are defined) is not available to the universal browser.

A semi-universal browser may contain some application-specific C++ classes and adapt itself to data structures constructed from these classes and elementary types.

3.10 Attributes

Attributes are used for additional control over class instances, type components and single data items in such cases when we want to alter the default behaviour of the PDGen tool when traversing the data elements.

The attribute information is *orthogonal* to the type structure declaration. Therefore it should normally be described outside the code containing the data types.

The graphical user interface designer (or generator) writes the attributes in a separate script file (the attribute definition file) which can be unspecified until the

application program starts. It allows altering many preference settings and options without recompilation and even during the runtime.

Each attribute specification has syntax:

`set_attr` { *component1, component2, ...* } { *attribute1, attribute2, ...* }

Component is specified as *path* or *Class-name*`::`*path* where the *path* has the same syntax as C++ qualified names. This means that the data members are selected with dot (`.`), and array elements are specified in square brackets `[`*index*`]`.

The use of patterns and regular expressions (within quotation marks `" "`) is allowed instead of standard C++ path syntax. In this case the attribute specification applies to all paths that match the pattern.

The attributes are specified as *attribute=value*.

Example: The attribute specification script

```
set_attr { Bicycle::front.pressure
           Bicycle::rear.pressure }
         { postfix = "H/m^2" }
```

states that for these variables the postfix (area normally used for physical units) should have the value `H/m^2`. The script is checked for correctness of the syntax; e.g. the system verifies that `pressure` is defined as a member of the class `Wheel`.

The same effect can be achieved by specifying a pattern:

```
set_attrp { "*pressure" } { postfix="H/m^2" }
```

The complete attribute specification necessary for the window in the Fig. 3(c) is:

```
set_attrp { "*angle" }  { postfix = "rad" }
set_attrp { "*radius*" } { postfix = "m" }
set_attrp { "*pressure*" } { postfix = "H/m^2" }
```

Several other available attributes are mentioned here:

- `validate` specifies a Tcl function that will be called each time when the input text area is altered.
- `hidden` specifies how and whether the item value is shown at the beginning. It can be shown, hidden or detached (Section 3.4).
- `load` and `save` specify whether the value is loaded from disk file and saved. By default it is both loaded and saved. `required` specifies that the user *must* enter some value; `read-only` specifies that the user cannot update it.
- `layout` specifies whether the array or structure should be displayed in vertical or horizontal layout. By default a heuristic is applied.

- Finally, `hook` gives the designer "free hands"; it specifies a Tk/Tcl function that is responsible for complete graphical representation of the value. A Tcl variable name with the value and Tk window name is given. The function is written by the designer and it has to create a window with the given name.

4 Automatic Generation of GUI from ObjectMath Models

The basic idea behind the second generation system is to generate all components of the graphical user interface from the application model, and to avoid manual editing when the model is updated and the user interface code is re-generated.

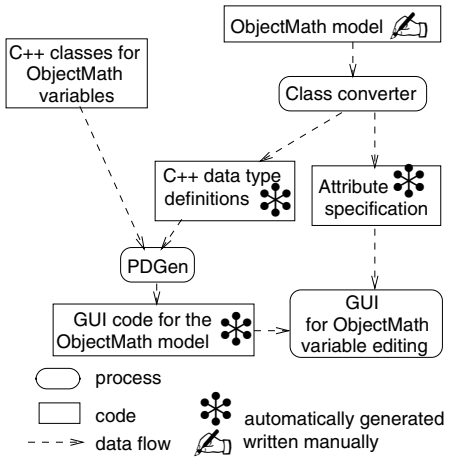


Figure 8: *Second generation graphical user interface generating system.*

The phases of the generation process are depicted in Fig. 8. First the ObjectMath model is analyzed by the *class converter*. All the data necessary for the class converter are contained in the class hierarchy diagram and the ObjectMath variable declarations. The class converter translates the ObjectMath class hierarchy to the relevant C++ class hierarchy.

The ObjectMath variables can be of twenty different predefined types which are implemented as C++ classes. For example, the ObjectMath type `DoubleVec3` (contains three double precision real numbers and can serve as operand in various ObjectMath arithmetic expressions) corresponds (in the simulation code) to the C++ class `DoubleVec3`. The data members of these C++ classes can represent the corresponding ObjectMath variables in the graphical interface. For example, the class `DoubleVec3` is defined as

```
class DoubleVec3
 { double X,Y,Z; // data members
   ....          // member functions,
```

```
                    // friend functions etc.
    }
```

and the `DoubleVec3` type in ObjectMath can be presented in a graphical user interface as three text input boxes marked with `X`, `Y` and `Z`.

The C++ class hierarchy (generated from the model) and C++ classes (that correspond to ObjectMath variable data types) are merged together and passed as the input for the PDGen tool. This tool generates code for a graphical user interface for the corresponding ObjectMath model.

Attribute specifications are necessary for the application with graphical interface at run-time. They contain some information absent in the C++ class declarations, such as physical units and persistence status. These attribute specifications are generated separately for the members of every class by the class converter.

4.1 Translation of an ObjectMath model to a C++ class hierarchy

Every single ObjectMath class gives rise to a C++ class. The ObjectMath inheritance means that all variables are inherited by the subclass. The same happens in the C++ class inheritance.

ObjectMath instances correspond to C++ classes, too. Such ObjectMath instances inherit all variables and formulae from the superclasses and, in addition, they may declare their own variables. ObjectMath *parts* serve for aggregation of class instantiations. They cannot specify their own variables. The parts can be modeled by C++ class data members.

ObjectMath variables become C++ data members.

The ObjectMath model (as a whole) corresponds to a single C++ class that includes one data member for every instance in the model.

4.2 Translation example

For the purpose of illustration we take our basic example (Fig. 1), a bicycle model. When the conversion described above is applied, the C++ type declarations shown in Fig. 3(a) are generated.

The attribute information is generated from the parameters in `Declare` statements. The attribute information is created according to the syntax rules described in the Section 3.10:

```
set_attr {Body::angle       } {postfix="rad"}
set_attr {Cylinder::radius  } {postfix="m"}
set_attr {Wheel::pressure   } {postfix="H/m^2"}
set_attr {Bicycle::framesize} {postfix="int"}
```

The graphical interface generated for this example is identical to Fig. 3(c).

4.3 Advantages of the second generation approach.

The new approach successfully solves the problems arising in the first generation approach (Section 2.9). There are no *update problems* because the application and its graphical user interface are generated simultaneously. The set of supported types can include arbitrarily complex types because we analyze all type declarations and derive the graphical user interface from them. The *variable grouping* and menu structure are automatically derived from the class structure of the ObjectMath model.

An additional advantage of the new approach is the automatic generation of persistence routines for arbitrarily complex data types.

5 Related work

5.1 Persistence generation systems

There are various ways to make objects persistent in object-oriented database management systems. In [BB88] such objects must be instances of special classes. In [Deux91] and [LLOW91] objects are assigned to "persistent variables" or "placed into persistent sets". In the *OBST* system [OBST94, CRSTZ92, ACSST94] application developers can program in an object-oriented extension of C. There are no pointers; unique object identifiers are used for references instead. The language includes an option to create objects as persistent data. Primitive data components are not lightweight, therefore high performance necessary for scientific computations and fitting memory constraints is hard to achieve.

Typically OODBMS automatically provide persistence for a specific language with the help of an OODB language compiler.

The *PGen* system [Tichy94, PGen94, Paulisch90] analyses C++ header files and generates C++ code for reading and writing variables of arbitrary types and classes defined there. This way persistence of data can easily be achieved. Traditionally, to make C++ objects persistent the user has to write the `Store()` and `Load()` functions for every class in the application. The PGen tool generates appropriate functions automatically. In most cases hardly any modifications are needed in the C++ header files.

One of the difficult problems with persistent data is how the data should be converted if the type declarations change. This is difficult to do automatically with C++ header files. A special type declaration editor can be designed to trace down all the changes in the type definitions. Then a conversion program can be generated.

Our system reuses the ideas and the code of PGen and extends it for variable display generation.

5.2 Display generation systems

In the *OBST* system a graphic shell visualizes the OBST objects and is used for debugging the data.

The systems *DOST* and *SUITE* [Dewan87, Dewan91, Dewan90, Suite91] generate variable displays from C header files. The translator analyzes specially annotated header files and generates C code that controls message-based communication between the application and a universal display manager. The generated code is linked with the original application. A variable of arbitrarily complex type can be displayed. The display manager can show various C data structures (including pointers and dynamic arrays coded as pointers). The layout can be customized by a large number of attributes (such as colors, help texts, constraints and validation functions) that can be adjusted interactively (with the help of a special preference setting dialog) or in the code. In these systems a single description of data types can specify the internal data, the data used for communication between the processes and the data for structure displays. Despite the large number of attributes associated with every type, variable or variable element, there is no possibility for the programmer to construct new graphical elements when necessary. Persistence can be implemented outside the system.

The *SmallTalk visualization system* [Dewan90A] uses the fact that all the objects in the program have an ultimate ancestor, `Object`, that has access to meta-information about the objects, e.g. a description of its structure. The display of any object is based on this meta-information. The user can change the attributes of object display by adding some extra SmallTalk code. Persistence can be supported by other SmallTalk methods and it is not part of the visualization system.

Some modern *debuggers* [Debug92] show displays with selected C, C++ or Pascal data structures for data inspection. They are based on symbol tables and dynamically change during program execution. The displays appear automatically in a manner similar to our approach, but they *cannot* be customized by the designer. The user can modify the values, but the validation procedures cannot be specified.

5.3 The C++ language and access to meta-information

C++ is a high level object-oriented language. Nowadays it is widely used in industry for scientific software design, including scientific computing.

C++ supports many ways to simplify the work of the application programmer and to avoid writing unnecessary code. Macro definitions, templates, operator overloading, class inheritance and standard class libraries cover almost all typical needs of application designers. They allow code to be written at a very high level and its size is close to the minimal possible if accurate design is applied. Therefore C++ code analysis and generation is not applied very often. One case where this is necessary is automatic code generation for persistence and displaying data for arbitrary C++ data types.

C++ is a hybrid language in the sense that it operates both with objects and non-

objects. This creates difficulties when applying a uniform approach to all values. C++ is a strongly typed language. Therefore when we create code for universal operations that could be applied to many types, code for every type should be written.

It is possible to design persistence and display routines for C++ manually. The problem is that *for every data type* separate routines should be written. Unlike a SmallTalk object, a C++ object cannot automatically provide (or inherit) meta-information about its structure (declarations of types, component names, sizes and types) during run-time. Therefore it does not know how to read, write or display itself.

Unlike SmallTalk, in C++ we cannot state simply that the variable `foo` should be stored, loaded or shown. For this purpose a relevant function must be declared and defined. The argument type for this function must be the same as `foo` has, and this function must access the internal structure of `foo`. Obviously, the code and control information for such a function should be created in advance. Such information can be extracted from C++ data type declarations. This is what the PDGen tool does.

6 Conclusions and Future Work

There is a substantial need for automatic generation of graphical user interfaces for many applications. The first generation system for generating user interfaces described in this paper has been in industrial use for more than two years. Experience shows that the model changes tend to require a number of manual changes to the user interface. We have provided a more flexible system that can automatically cope with model changes. Therefore, the more universal second generation system has been designed. The user interface constructed in a partly manual way using the first generation tool can now be generated completely automatically.

The PDGen tool is applied to ObjectMath, C and C++ programs, but it can also be adapted to other languages. Type definitions can be extracted in several ways:

- the source code is parsed and analyzed (in our tool we reuse the analyzer from PGen (see Section 5.1) and apply it to the C++ code),
- analysis of the symbol tables generated by some standard compiler (this is the approach implemented in Suite, (see Section 5.2),
- extracting type definitions from the model description, if the application is generated from this model (we apply this to the ObjectMath models),
- creation of the type table with the help of a special data-type definition editor.

The last approach can be combined with the customization tool. In this way both data-type definitions and information about interface details (attribute values)

will be integrated under the strict control of one tool. This reduces the possibility of data type mismatches and update problems.

In some languages the type information can be available at run-time with the help of built-in functions; in this case there is no need for it in code analysis and generation at all.

We are currently working on several extensions to the basic idea implemented in the automatic graphical user interface generator. Some interesting questions that could be considered include:

- the application of the PDGen tool to programs in other languages;
- integration of the tool with (extensible) symbolic debuggers;
- automatic generation of a graphical user interface for member functions (not only for data members). For example, if a member function has no arguments it is displayed as a button. When the button is pressed, the function is invoked.
- a more general array browser implementation;
- integration of ObjectMath with tools for data visualization, as it is implemented in the output data browser;
- implementation of the universal browser (Section 3.9) that adapts the graphical user interface according to the type tables given together with the input data;
- the design of a meta-editor that can edit data type definitions.

Remote data editing with the help of widely distributed WWW browsers is another application area. The data resides on the server and can be updated by the clients with the help of HTML interactive forms. One of our future research topics is automatic generation of HTML-based editing tools from data structure specifications.

Finally we would like to mention that a WWW site devoted to the PDGen tool has been organized [PDGen96]. More details about the systems discussed in this paper are available in [E96].

Acknowledgments

Lars Viklund and other members of the PELAB group contributed to the design of the ObjectMath language and its implementation. Ivan Rankin improved the English in this paper.

This project is supported by the Swedish Board for Industrial and Technical Development, the Swedish Board for Technical Research and SKF Engineering and Research Centre and the PREPARE Esprit-3 project.

References

[ACSST94] J.Alt, E. Casais, B. Schiefer, S. Sirdeshpande, D. Theobald. *The OBST Tutorial*, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, FZI.049.2, 15nd December, 1994

[BB88] A. Bjrnerstedt, S. Britts, "AVANCE: An Object Management System", *SIGPLAN Notices*, vol. 23, pp. 206-221. Nov. 1988. In *Proceedings of the OOPSLA'88 Conference*, San Diego, CA, 25-30 September 1988.

[CRSTZ92] E. Casais, M. Ranft, B. Schiefer, D. Theobald, W. Zimmer. *OBST - An Overview*, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, FZI.039.1, June 1992

[Debug92] Debugging a Program. *SparcWorks documentation.* SunPro, October 1992

[Deux91] O. Deux et al., "The O2 System", *Communications of the ACM*, vol. 34, pp.34-48, Oct. 1991

[Dewan87] P. Dewan, M. Solomon. "Dost: An Environment to Support Automatic Generation of User Interfaces". In *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, *SIGPLAN Notices* Vol. 22, No. 1, January 1987, pp. 150-159.

[Dewan90] P. Dewan, M. Solomon. "An Approach to Support Automatic Generation of User Interfaces". *ACM TOPLAS*, Vol. 12, No. 4, pp. 566–609 (October 1990)

[Dewan90A] P. Dewan. "Object-Oriented Editor Generation". *Journal of Object-Oriented Programming*, vol. 3, 2 (July/Aug 1990), pp. 35-49

[Dewan91] P. Dewan. "An Inheritance Model for Supporting Flexible Displays of Data Structures". *Software - Practice and Experience*, vol. 21(7), 719-738 (July 1991)

[Dewan91A] P. Dewan. *A Tour of the Suite User Interface Software*. Included in [Suite91]

[E96] V. Engelson. *An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing*. Linköping Studies in Science and Technology. Licentiate thesis No 545. Department of Computer and Information Science, Linköping University, 1996.

[Fritzson93] P. Fritzson, V. Engelson, L. Viklund. "Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment". In *Proc. Of DISCO'93 - Conference On Design and Implementation of Symbolic Computation Systems*, LNCS 722, Sept. 1993

[Fritzson95] P. Fritzson, L. Viklund, J. Herber and D. Fritzson. "High-level Mathematical Modeling and Programming". *IEEE Software*, July 1995, pp. 77-86.

[FWHSS96] P. Fritzson, R. Wismüller, Olav Hansen, Jonas Sala, Peter Skov. "A Parallel Debugger with Support for Distributed Arrays, Multiple Executables and Dynamic Processes". In *Proceedings of International Conference on Compiler Construction*, Linköping University, Linköping, Sweden, 22-26 April, 1996, LNCS 1061, Springer Verlag.

[LLOW91] Ch. Lambs, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, vol. 34, pp. 50-63, Oct. 1991

[OBST94] OBST, a persistent object management system. Available at `ftp://ftp.ask.uni-karlsruhe.de/pub/education/computer_science/OBST`, see also `http://www.fzi.de/divisions/dbs/projects/OBST.html`

[Ou94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994

[Paulisch90] F.N. Paulisch, S. Manke, W.F. Tichy. "Persistence for Arbitrary C++ Data Structures". In *Proc. of Int. Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, FRG, May 1990, pp. 378–391.

[PDGen96] V. Engelson. *The PDGen tool.* Information available at `http://www.ida.liu.se/~vaden/pdgen`.

[PGen94] *PGen, a persistence facility.* This software is available at `ftp://ftp.ira.uka.de/systems/general`.

[Tichy94] W.F. Tichy, J. Heilig, F.N. Paulisch. "A Generative and Generic Approach to Persistence". *C++ report*, January 1994. Also included in [PGen94].

[Wolfram91] S. Wolfram, *Mathematica - A System for Doing Mathematics by Computer* (second edition), Addison-Wesley, Reading, Mass., 1991.

[Viklund95] L. Viklund and P.Fritzson, "ObjectMath: An Object-Oriented Language for Symbolic and Numeric Processing in Scientific Computing", *Scientific Programming*, Vol. 4, pp. 229-250, 1995.

[Suite91] SUITE, user interface software. This software is available as `ftp://ftp.cs.unc.edu/pub/users/dewan/suite`. Some information available as `http://www.cs.unc.edu/~dewan/`

[SUN90] SUN Microsystems Inc. *Network Programming Guide (Ch. 5,6)*, 1990

Paper 4

USING THE *MATHEMATICA* ENVIRONMENT FOR GENERATING EFFICIENT 3D GRAPHICS

Vadim ENGELSON Peter FRITZSON Dag FRITZSON
Department of Computer and Information Science, Linköping University
S-58183, Linköping, SWEDEN
{vaden,petfr,dagfr}@ida.liu.se
http://www.ida.liu.se/~vaden

ABSTRACT

Mathematica is an integrated environment for symbolic transformation of mathematical formulas. This environment has applications in scientific computing, scientific visualization and education. *Mathematica* provides the ability to describe visualized objects in form of mathematical formulas and expressions. Such descriptions are more clear and concise than low-level C or C++ code. Many visualization systems require input in the form of (sometimes huge) data files, which is a disadvantage for highly interactive and animated 3D graphics applications. This is also the case for graphics expressed in *Mathematica* which are computed interpretively and saved in a static data form before display. This causes low graphic performance. In this paper we describe an approach which uses object geometry descriptions in the form of efficient program code instead of huge data files. We have built a tool that produces 3D visualizations of geometrical objects and object trajectories from mathematical specifications expressed as parametric functions in *Mathematica*. A compiler has been developed which generates efficient C++ code from such functions and symbolic expressions. This code is linked together with a powerful 3D browsing environment and uses OpenGL with possible hardware support. All the computations are performed *within* the visualizing application. Object geometry, color, etc. can be changed dynamically during animations. Thus the flexibility of interactive exploration of 3D scenes and animation become available for the end-user.

Key Words: Mathematica, Compilation, Surface geometry, Three-dimensional visualization.

Published in *Proceedings of EduGraphics/ CompuGraphics-97*, Vilamoura, Portugal, December 15-17, 1997, Graphic Science Promotions and Publications, (Harold P. Santo, ed.), pp. 222 – 231.

1 Introduction

Data to be visualized is traditionally prepared outside of the visualization tools. However in some cases the opposite approach is more practical. In this paper we address the problem of visualization of mathematical models, particularly models expressed in the symbolic mathematical modelling and programming language *Mathematica* [Wolfram96] and an object-oriented extension of this language called *ObjectMath* [Fritzson et al. 95, Viklund-Fritzson95, ObjectMath97].

The functions describing surfaces and trajectories can be very complicated. Therefore it is easier to specify those in the form of traditional mathematical formulas than to write low-level C or C++ code.

Our current prototype implementation, called *MAGGIE* (Mathematical Graphical Generated Interactive Environment) has shown the feasibility of this ap-

proach. In this implementation *Mathematica* functions are translated to optimized C++ code. An OpenGL-based graphical environment has been constructed, for efficient visualization of such functions.

Currently we work with industrial applications such as mathematical modelling and simulation of rolling bearings in cooperation with SKF - the world's leading bearing manufacturer. Advanced 3D visualization and animation is very important for the end-users to help in interpreting the results. This animation tool [ObjectMath97] has demonstrated the usefulness and feasibility of high performance visualization/animation.

In the following sections we explain the role of *Mathematica*, discuss visualization techniques as well as illustrate the use of our *MAGGIE* tool. It should be noted that we primarily consider visualizations of dynamically changing surfaces. Volume visualization problems are outside the scope of this paper.

1.1 The *Mathematica* Environment

The *Mathematica* system (we use version 3.0) is an integrated environment which supports a wide spectrum of activities. The user works within a *notebook* (a live document) which consists of *cells* (Fig. 1). The *input cells* contain expressions and function definitions in traditional mathematical notation. When they are evaluated, *output cells* appear, which contain transformed expressions, numerical results and 2D or 3D plots.

The system is widely used in education, particularly in Calculus, Algebra, Geometry and Physics. The students modify expressions in the notebooks, apply algebraic transformations and investigate numerical and graphical results. Currently 2D plots are created quickly, whereas rendering more complicated 3D plots, complex pictures and particularly animations may take several minutes.

Notebooks are also used for publications and interactive presentations. The formulas and equations can be mixed with ordinary text, such as explanations to formulas. In this way traditional mathematical papers or books can be written. The notebooks can be scrolled up, down, and displayed via a video or overhead projector, e.g. during lectures. Expressions in the input cells can be evaluated during presentations.

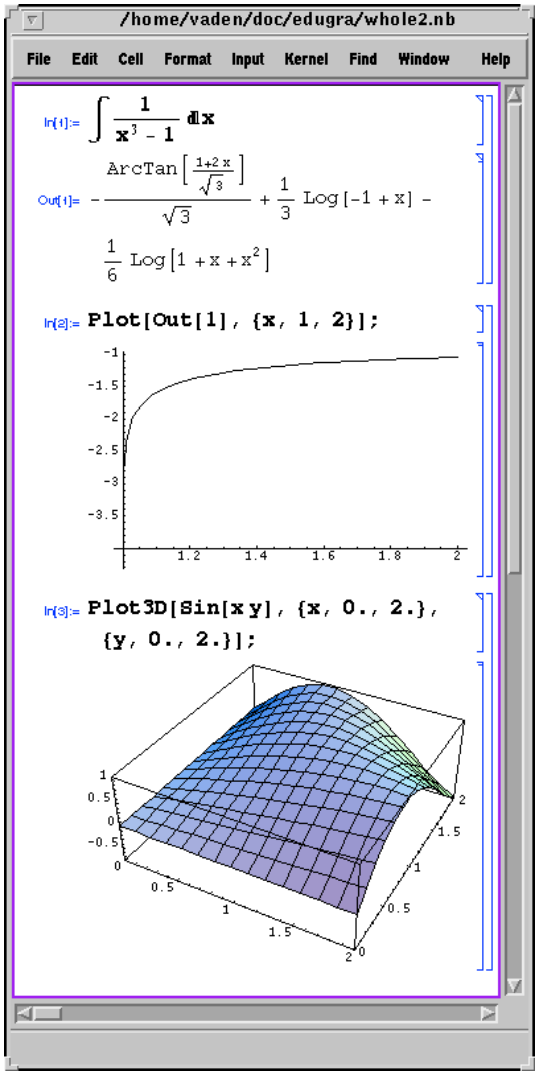


Figure 1: A *Mathematica* notebook with results of a symbolic integration, 2D and 3D plots. The notebook cell structure is made visible via brackets on the right side.

Finally, notebooks are used as a format for Internet document exchange between mathematicians since mathematical notation is shown in its traditional form and can be directly edited.

1.2 Graphical Output In The *Mathematica* System

The *Mathematica* system includes several built-in functions for visualization.

The function `Plot[`f`[`x`], {`x`, `x_{min}`, `x_{max}`}]` generates 2D plots of the function $f(x)$. The function `Plot3D[`g`[`x`,`y`], {`x`, `x_{min}`, `x_{max}` }`,

$\{y,y_{min},y_{max}\}$] generates 3D plots of the function $g(x, y)$. The functions $f(x)$ and $g(x, y)$ can be defined symbolically as shown in Fig. 1

The performance of `Plot` and `Plot3D` is currently rather low for the following reasons: first, the functions f and g are computed symbolically and interpretively. They are not compiled to efficient machine code. Also, graphics are rendered through a universal interface (PostScript format). Hardware graphic acceleration is not used at all.

Creating a single 3D picture of realistic application models, e.g. of a rolling bearing, takes far too long (more than an hour).

1.3 The Visualization Problem

Scientific visualization is necessary for interpretation and making use of data produced from numerical experiments and measurements. When designing scientific visualization tools, several kinds of input and output format need to be considered. The inputs are:

- data stored in an external file (in tabular form)
- data specified directly by the end-user via dialogs and graphical controls
- mathematical formulas (functions) that can compute or process numerical data
- configuration options that specify the structure and attributes of visualization

The traditional means of result visualization would be:

- compute all results and store them on disk
- create a geometry model
- visualize results using a separate visualization package
- change input parameters
- repeat all the steps again

Instead, our tool automatically produces graphical output from the inputs. Specifically, the outputs are:

- vector and surface 2D and 3D graphics
- animation of graphics
- constructions that combine elementary graphical objects into more complex objects.

In our approach the numerical data computed by functions given as graphic input, generally describe the 3D geometry of the parametric surfaces, i.e. a set of points $(x, y, z) = f(u, v)$ for $u \in [u_{min}, u_{max}]$, $v \in [v_{min}, v_{max}]$. Here (x, y, z) are Cartesian coordinates, f is a mathematical function. The rectangle formed by $u_{min}, u_{max}, v_{min}, v_{max}$ is the area of the grid.

In addition to the coordinates for a point on the surface, auxiliary properties can be defined:

- surface color according to a color scale
- color intensity
- direction of a vector field
- trajectory of the surface or an object in space defined by changing object coordinates as a function of time.

Example 1. Assume that there exists a table which contains the position of a point (x_i, y_i, z_i) relative to coordinate system C for every time t_i in the interval $[t_0, t_n]$. We might need to visualize the acceleration vector movements for the time period [t_0,t_n] in another coordinate system D.

The scientific visualization software should perform several steps:

- read the positions of the point from an external file
- perform mathematical computations to calculate acceleration from position for given time t_i
- transform acceleration vector from the coordinate system C to the coordinate system D.
- check configuration options which specify that in the current case the results should be visualized as a vector
- perform rendering of a vector.

2 A Code Generation Approach

The approach proposed in this paper uses *code generation* technique to produce efficient implementation of visualization code.

The input to the code generator is a mathematical model expressed in *Mathematica*. In the simplest case such a model contains a set of parametric functions that describe the graphical objects to be visualized. Every function defines the geometry of the object surface.

Example 2 We consider a single surface defined by the parametric function[1]

$$F(x,y,t) =$$

$$= \iint \frac{y}{x^2+1} + x^2\cos(x-ty) + \sin(y)\,dx\,dy$$

The surface is the set of points (x,y,z) where z changes according to additional parameter t, i.e. $z(t) = F(x,y,t)$ for $x \in [x_{min}, x_{max}]$, $y \in [y_{min}, y_{max}]$, $t \in [t_{min}, t_{max}]$.

In *Mathematica* notation this function is written as

```
F[x_, y_, t_] =
  ∫∫ (y/(x^2 + 1) + x^2 Cos[x - t y] + Sin[y]) dy dx
```

All the steps described below are performed *automatically* by our *MAGGIE* tool. The result of the processing is a stand-alone application (Fig. 2). This application allows browsing the surface, rotating it and manipulating it in various ways.

There are two ways to manipulate the time parameter t: First, it can be changed manually by the user (via a scrollbar). The user views the objects at a certain time instant. Second, it can be used as an argument for animation. The parameter t changes gradually from t_{min} to t_{max}. For every step the picture is updated, creating the animation effect.

Application generation. Mathematical formulas like the double integral above are processed by *MAGGIE*, giving symbolic expression as the result. For the double integral the result is

```
F[x_, y_, t_] = 1/2 y^2 ArcTan[x] +
  1/t (-t x Cos[y] - 2 Cos[x - t y] +
    x^2 Cos[x - t y] - 2 x Sin[x - t y])
```

Note that in this case the integral can be evaluated with traditional functions (cos, sin etc.). Otherwise, *Mathematica* represents the result with the help of special functions.

We have built a *Mathematica* to C++ compiler called *MathCode* [Fritzson97]. The symbolic expression is translated into the C++ code (Fig. 3). Note that common subexpression elimination (CSE) is applied so that every expression is evaluated only once. This significantly improves performance of complex expression evaluation[2].

The function `F` is linked with OpenGL library and our browsing package. This way a stand-alone application is created. It does not require connection to *Mathematica* anymore, and all the data is calculated on demand within the application. The libraries GLUT [Kilgard97] and Tcl/Tk with Togl [Paul-Bederson97] are used for programming the control dialog windows [Engelson et al. 97].

Browser configuration. Since all the calculations are performed during the visualization the tool allows

— modify surface grid size, i.e. values of x_{min}, x_{max}, y_{min}, y_{max}

— modify the value of t interactively or gradually (perform animation of dynamically changing surface in 3D)

— modify the level of detail u, i.e. number of grid points used for rendering in each direction[3]. The function `F` is evaluated for

$$x = x_{min} + \frac{k}{u}(x_{max} - x_{min}),$$

$$y = y_{min} + \frac{l}{u}(y_{max} - y_{min}),$$

[1] This particular function has been chosen for the purpose of a small demonstration.

[2] CSE is a standard technique used by many optimizing compilers. However, its implementation is limited to small expressions and cannot be performed if external functions (like sin) are called.

[3] In future the level of grid granularity should be automatically adapted: a high zooming rate will automatically switch the tool to higher resolution.

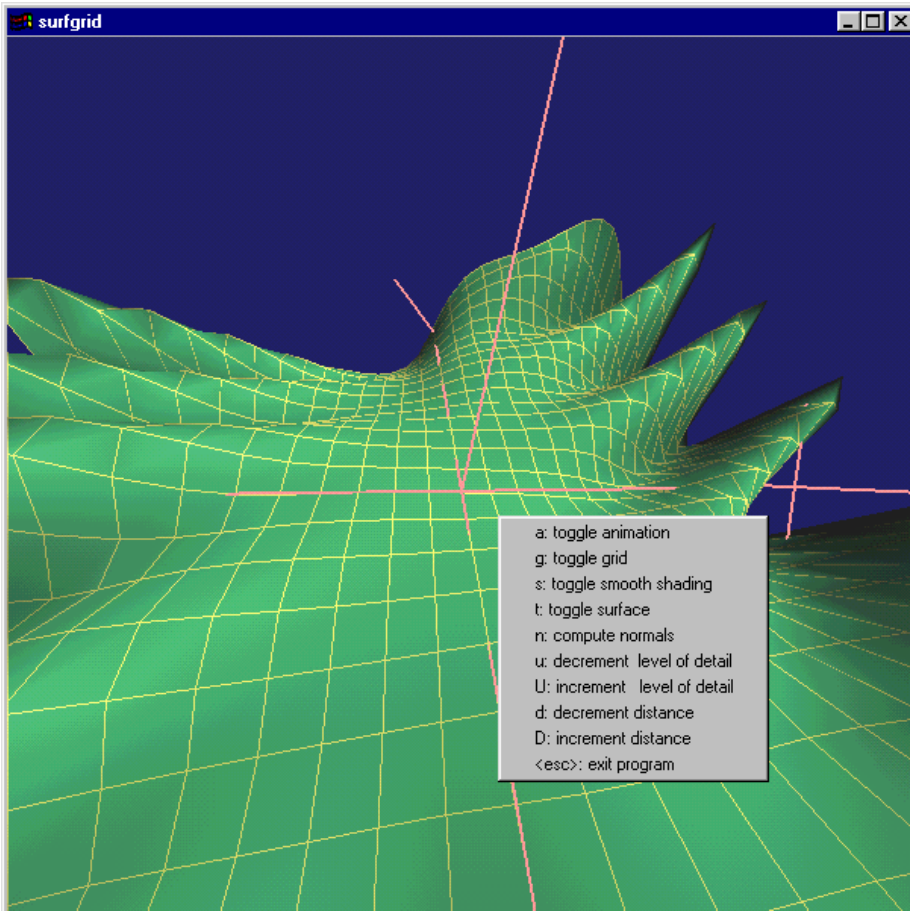


Figure 2: The screen shot of the *MAGGIE* tool with animation of a parametric surface.

$$k, l \in \{0..u\}$$

Also the following parameters can be modified, as it is usually done in 3D browsers: grid, surface, evaluation of normals, smooth shading can be toggled on and off. The browser has traditional tools for navigation, including zooming, scaling, viewpoint and object rotation, manipulating cameras and lights, and showing several views simultaneously from different camera positions.

3 Introducing the Hierarchy of Surfaces and Objects

Visualizations may require rendering of several surfaces simultaneously. A set of surfaces may form geometrical objects. Some objects can be attached to other objects, some of them can be in movement.

In order to describe more complex visualizations we introduce the notion of *scene*. A *scene* is a hierarchy composed from *objects*. Every *object* can be composed from one or several surfaces or objects. Every surface is described by a parametric function, as discussed above. An object can have a set of attributes that describe it, specifically mathematical functions that determine position of the object.

An instance of a hierarchy is shown in Fig. 4.

The output of our visualization generator is an application that contains a graphical interactive environment which allows the user to explore the objects and functions described in the model.

3.1 Syntax of Hierarchical Scene Description

According to *Mathematica* syntax we use the form $f\,[p_1, p_2, \ldots,\ option_1 \text{->} value_1, option_2 \text{->} value_2, \ldots]$ to specify the call of the function f with arguments p_1, p_2 etc. and optional arguments

```
double F(doubleN &P_D_Txyt)
/* P_D_Txyt contains x,y,t */
{double om_T1,om_T2,
om_T3,om_T4,om_T5,om_T6,
om_T7,om_T8,om_T9,om_T10,
om_T11,om_T12,om_T13,om_T14,
om_T15,om_T16,om_T17,om_T18,
om_T19,om_T20,om_T21,om_T22;
om_T1 = P_D_Txyt(3);
om_T2 = P_D_Txyt(2);
om_T3 = om_T2*om_T1;
om_T4 = -om_T3;
om_T5 = P_D_Txyt(1);
om_T6 = om_T5+om_T4;
om_T7 = sin(om_T6);
om_T8 = -2*om_T5*om_T7;
om_T9 = cos(om_T2);
om_T10 = om_T9*om_T5*om_T1;
om_T11 = -om_T10;
om_T12 = om_T5*om_T5;
om_T13 = cos(om_T6);
om_T14 = om_T13*om_T12;
om_T15 = -2*om_T13;
om_T16 = om_T15+om_T14+om_T11+om_T8;
om_T17 = om_T16/om_T1;
om_T18 = om_T2*om_T2;
om_T19 = atan(om_T5);
om_T20 = om_T19*om_T18;
om_T21 = om_T20/2;
om_T22 = om_T21+om_T17;
return om_T22;
}
```

Figure 3: The C++ code generated for the function F, the double integral in Example 2.

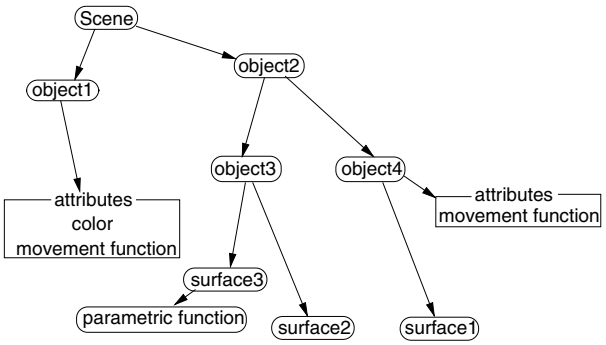


Figure 4: A hierarchy of scene, objects, its attributes and surfaces.

(i.e. *named* parameters) with their values. The form $\{e_1, e_2, ...\}$ is a list of elements e_1, e_2, etc.

The *scene* is described as `Scene[`$\{o_1, o_2, ...\}$`, InputParameters -> {` $ip_1, ip_2, ...\}$`]` where o_i is an *object* described below, ip_i is a name of a scalar *input parameter*. The *objects* are visualized in the scene. The user can manipulate the values of *input parameters* via dialog boxes.

Objects. There are three kinds of objects:

1. `Surface[`f`, Grid->`$\{u_{min}, u_{max}, v_{min}, v_{max}\}$`, ExtraParameters-> {`$ip_1, ip_2, ...\}$`]` defines an object which is visualized as a parametric function f with arguments u, v on the grid and additional arguments $ip_1, ip_2, ...$ These additional arguments can be input parameters of the scene. The function returns coordinates and color for every grid point: x, y, z, h, where x, y, z are Cartesian coordinates relative to the object coordinate system; h is intensity which can be visualized with some color from the color spectrum.

2. `StandardSurface[(Cube | Sphere | Polygon), Size->`s`]` defines one of many standard surfaces, such as cube, sphere or polygon and its size (radius, length and width, if necessary).

3. `Compound[{` $o_1, o_2, ...\}$`]` defines an object that is not visualized. However, it serves as a container for several other objects.

The objects have additional options that control their position and orientation.

`TranslateFunction->`f`, TranslateParameters->`$\{ip_1, ip_2, ...\}$ specifies that the coordinate system of the object is shifted from the coordinate system of the parent object. This shift is described by the function f which takes arguments $ip_1, ip_2, ...$ and returns Cartesian coordinates x, y, z relative to the parent object.

`RotationFunction->`f`, RotationParameters->`$\{ip_1, ip_2, ...\}$ specifies that the coordinate system of the object is rotated in relation to the coordinate system of the parent object. The rotation is described by the function f which takes arguments $ip_1, ip_2, ...$ and returns matrix of rotation relative to the parent object.

Generally, all configuration options that are not specified in the scene description can be adjusted by the user during visualization. For instance all *input parameter* values can be changed manually, or used for animation. Also, the minimal and maximal value of the parameters can be specified.

External data access. There are several ways to access necessary data:

— There are special functions that we designed for reading numerical data from an input file. The user can update the file manually or generate (compute) it using some program. This way the visualization uses a simulation trace, results of other computations or measured data.

— External C++ and Fortran functions can be called.

— An extensive collection of built-in *Mathematica* functions (the system contains almost 1000 functions) can be called.

4 Example of a Scene

We illustrate diversity of visualizations by a scene which consists of two spherical stones falling into the water. The surface of the water illustrates a dynamically changing surface. The stones are rigid bodies, moving according to some trajectory. They simultaneously start falling vertically from different positions above the water. The observer can see the animation of falling stones and waves on the water surface. He or she can also modify the starting positions of the stones.

The model is described in Fig. 5. This description shows two spheres (`stone1` and `stone2`) which fall down from their respective positions (`x1`, `y1`, `z1`), (`x2`, `y2`, `z2`).

The formula of the surface together with information about the structure of the model is compiled to C++ code. It is linked with the OpenGL library and common routines for browsing. The result can be seen in Fig. 7.

The observer can manipulate all input parameters via an automatically generated dialog window (see Fig. 6). The time parameter t is used for animation. Other scalar parameter values are set via respective sliders. The limit values of the sliders can be changed interactively.

For other observations we could leave t fixed and animate the picture by gradually changing one of other parameters. Such a change of the roles of the parameters can be done interactively. In this case the dialog window contents changes automatically.

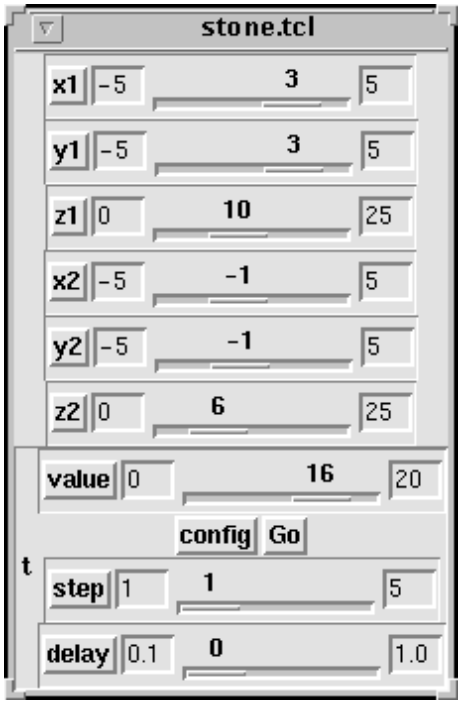


Figure 6: A dialog for interactive control of the stone and water visualization.

5 Related Systems

Universal visualization and animation tools available on the market are often insufficient for high performance industrial applications. Performance is not sufficient, options to call user-specified functions are limited or absent, options to configure visualization are limited. In practice, for most simulation application domains, specific application dependent visualization support is usually required.

A *Mathematica* to VRML compiler. The *Mathematica* to VRML (Virtual Reality Modelling Language) compiler [Fisher97] allows generation of files with point coordinates that describe sets of vectors and polygon surfaces. Such files describe sets of

```
fall[t_,x_,y_,z_]={x,y,z-(9.8/2)*t^2}; (* Position of falling stone *)
tfall[zs_]=Sqrt[2*zs/9.8];(* How long it takes to fall from the height zs *)
dist[x_,y_,xs_,ys_]=Sqrt[(x-xs)^2+(y-ys)^2]; (* Distance between two points *)
waterh3[m_]=Which[m<0,0, m<2, (m-1)^2-1, m<4, 1-(m-3)^2, m>=4 , 0]; (* Wave height at the point of fall
at time m. We compose the wave from two parabolic curves *)
waterh2[d_,t_]=If[t<0,0,waterh3[t-d]/(d/5.0+1.0)]; (* Waves distribute with fading and with speed 1 m/s *)
waterh1[x_,y_,t_,xs_,ys_,zs_]=waterh2[dist[x,y,xs,ys],t-tfall[zs]]; (* only distance and time after fall
is important *)
waterh[x_,y_,t_,x1_,y1_,z1_,x2_,y2_,z2_]=waterh1[x,y,t,x1,y1,z1]+
+ waterh1[x,y,t,x2,y2,z2]; (*effects of two stones are summed up *)
stone1=StandardSurface[Sphere,Size->{0.1},TranslateFunction->fall,
                       TranslateParameters->{t,x1,y1,z1}]; (* Stone 1 *)
stone2=StandardSurface[Sphere,Size->{0.2},TranslateFunction->fall,
                       TranslateParameters->{t,x2,y2,z2}]; (* Stone 2 *)
water=Surface[waterh,{t,x1,y1,z1,x2,y2,z2}]; (* water *)
Scene[{stone1,stone2,water},InputParameters->{t,x1,y1,z1,x2,y2,z2}]; (* time and initial positions of
the stones are controlled by the user *)
```

Figure 5: A simplified model of falling stones in *Mathematica* syntax.

graphical objects that can be explored using a VRML browser. The performance is relatively good because VRML browsers often use hardware graphic accelerators.

This approach has the disadvantage of being rather static and not allowing for the dynamic aspects of graphics. Animations of realistic size cannot be implemented this way since a sequence of VRML files will occupy large amounts of space. Reading these files slows down the animation. If picture resolution (the level of detail) needs be changed, the plotted function must be recomputed on a more fine grained grid. This requires the generation of a new VRML file.

MathLive and similar systems. In these systems the graphics created by the `Plot3D` is converted to sets of 3D vectors and polygons and sent (via `MathLink`) to a graphical 3D browser. By using such a browser the users can rotate, zoom, change textures of the surfaces etc.

This approach supports more interactivity between the *Mathematica* system and the browser. However, function computation is still performed within *Mathematica* and the performance is not sufficient.

Examples of these systems are the MathLive and MathLive Pro tools [MathLive97] and Dynamic Visualizer [Dynamic Visualizer97].

Other graphical systems. Why not just buy an existing visualization package? Unfortunately, most available software known to us, e.g. GLView [GLView97] and AVS [AVS97], require that the geometry of visualized objects is specified outside the program and is read from an external file. For instance, the geometry of complex bearing models, especially with structural deformations, may consist of more than 10 - 50 thousand polygons. Their coordinates should be rapidly computed and it would be highly inefficient to store them outside the visualizing program. Therefore these systems do not match our requirements.

The animator/visualizer should also include high-performance functions for reading data files produced from simulations and for computation of polygons. This option is absent in available systems (or is available, but inefficient).

Another problem is that no available visualization package accepts mathematical models as input.

AVS supports construction of process networks. These networks may contain geometry viewers as well as components written by the user in C which, e.g., perform computations and fast I/O. For the moment the speed of communication between the processes is not high enough for our visualization needs.

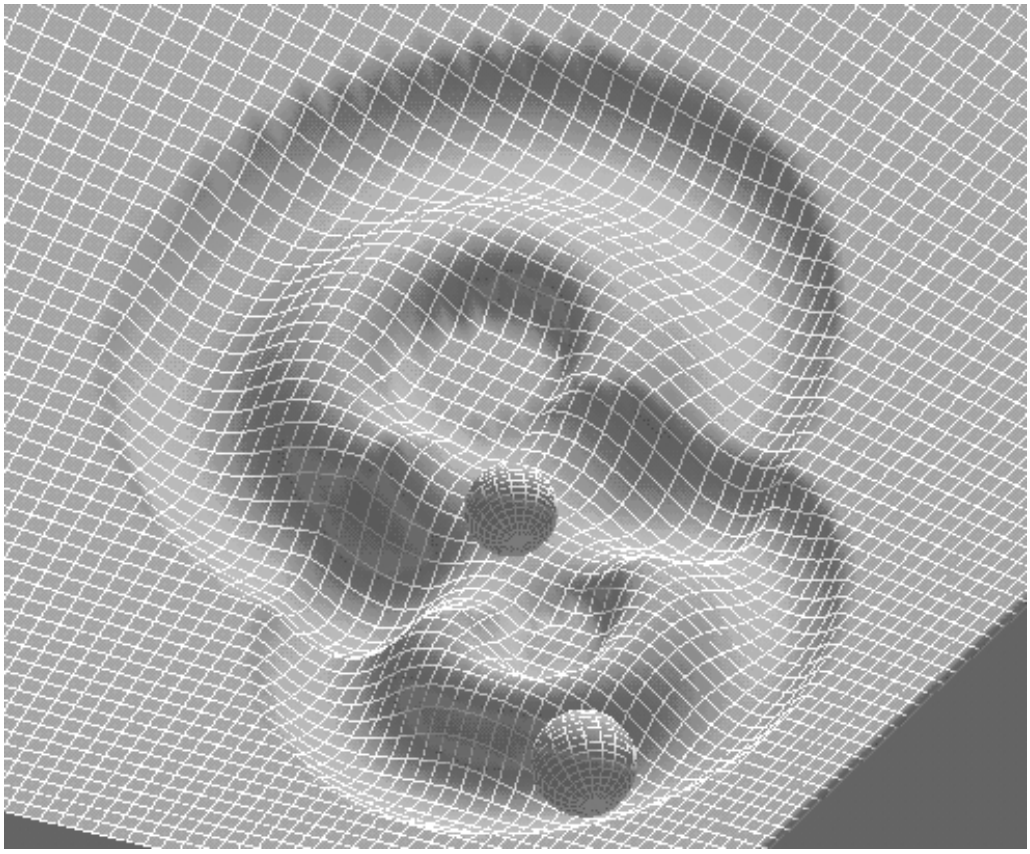


Figure 7: Water surface after the stones fell down and the waves appeared. The stones are below the surface and we look at them from below. This is a screen shot from the animation sequence.

6 Conclusions and Future Work

This paper presents ongoing work to give more efficient and flexible ways to visualize massive amounts of numerical data from simulations. We are looking for a compromise between the needs of high performance graphics and our desire too keep down the complexity of models described in *Mathematica*.

In a previous project part of simulation software for rolling bearings was automatically generated by the *ObjectMath* system from object oriented mathematical models at a high level of abstraction. A 3D animation tool (called *BEAUTY*, Bearing Animation Utility) has been implemented for several bearing application models.

The unique advantages of the *ObjectMath* system in producing both simulation code and visualization from the same mathematical models have been demonstrated on industrial bearing simulation applications, and have potential applicability in several additional application areas. This approach ensures correctness and consistency between the simulation and the visualization, and saves substantial programming effort and speeds up development considerably.

Our new system, *MAGGIE*, can be used both for small models and complex industrial scale applications.

Our future research will be directed to:

— Closer integration of the 3D graphical browser with the *Mathematica* environment.

— More automation and flexibility at code compilation stage, e.g. incremental compilation would be useful.

— Using the tool with various graphical environments, e.g. virtual reality modelling systems.

— Using the tool for computational steering[14], online, interactive control of the simulation via graphical environment.

More details can be found on the author's WWW page.

REFERENCES

AVS 1997 http://www.avs.com

Dynamic Visualizer 1997 http://www.wolfram.com/applications/visualizer/index.html

Engelson, V., Fritzson, P., Fritzson, D. 1996 Automatic Generation of User Interfaces From Data Structure Specifications and Object-Oriented Application Models, *Proceedings of European Conference on Object-Oriented Programming (ECOOP96)*, Linz, Austria, 8-12 July, Lecture Notes in Computer Science, Vol. 1098, Pierre Cointe (Ed.), pp 114-141. Springer-Verlag.

Engelson, V., Fritzson, P., Fritzson, D. 1997 Generating Efficient 3D Graphics Animation Code with OpenGL from Object Oriented Models in Mathematica, *Innovation in Mathematics, Proceedings of the Second International* Mathematica *Symposium*, Rovaniemi, Finland, 29 June - 4 July, V. Keränen, P. Mitic, A. Hietamäki (Ed.), pp 129 - 136.

Fisher S., WebMath 1997 http://www.mathsource.com/cgi-bin/msitem?0208-336

Fritzson, P., Viklund, L., Herber, J., Fritzson, D. 1995 High-level Mathematical Modeling and Programming, *IEEE Software*, July, pp 77-86.

Fritzson, P. 1997 Static and String Typing for Extended Mathematica, *Innovation in Mathematics, Proceedings of the Second International* Mathematica *Symposium*, Rovaniemi, Finland, 29 June - 4 July, V. Keränen, P. Mitic, A. Hietamäki (Ed.), pp 153 - 160.

GLView 1997 http://www.viewtech.no/GLview.html

Kilgard, M. 1997 OpenGL Utility Toolkit, GLUT, http://reality.sgi.com/mjk_asd/glut3/glut3.html

MathLive and MathLive Pro Tools 1997 http://www.milohedge.com/products/mathlive/mathlive.html

Neider, J., Davis, T., Woo, M. 1993 *OpenGL Programming Guide*, Addison-Wesley.

ObjectMath Home Page 1997 http://www.ida.liu.se/labs/~pelab/omath

Paul B., Bederson B. 1997 Togl — a Tk OpenGL Widget http://www.cs.unm.edu/~bederson/ Togl.html

Viklund, L., Fritzson, P. 1995 ObjectMath: An Object-Oriented Language for Symbolic and Numeric Processing in Scientific Computing, *Scientific Programming*, Vol. 4, pp 229-250.

Wolfram, S. 1996 *The Mathematica Book*, Wolfram Media Inc., Champaign, IL, USA.

Paper 5

Tools for Design, Interactive Simulation and Visualization for Dynamic Analysis of Mechanical Models

Vadim Engelson, PELAB, IDA, Linköping University

Abstract

The complexity of mechanical and multi-domain simulation models is rapidly increasing. Therefore new methods and standards are needed for model design. A new language, Modelica, has been proposed by an international design committee as a standard, object-oriented, equation-based language suitable for description of the dynamics of systems containing mechanical, electrical, chemical and other types of components. However, it is complicated to describe system models in textual form, whereas CAD systems are convenient tools for this purpose. Therefore we have designed an environment that supports the translation from CAD models to standard Modelica representation. This representation is then used for simulation and visualization. Assembly information is extracted from CAD models, from which a Modelica model is generated. By solving equations expressed in Modelica, the system is simulated. We have designed several interactive 3D visualization tools which display expected and actual model behavior, as well as additional graphical elements for the purpose of engineering visualization. We applied this environment for robot movement and helicopter flight simulation.

Keywords: CAD, SolidWorks, Mechanical modeling, Simulation, Animation, Visualization, Modeling languages, Modelica, OpenGL .

1 Background

The use of computer simulation in industry is rapidly increasing. Simulation is typically used to optimize product properties and to reduce product development cost and time to market. 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 mechanical, electric, hydraulic, thermodynamic, and control system components.

In this chapter we concentrate on simulation of mechanical models, in particular, systems of multiple rigid bodies, as well as simulations where such models are major components in the simulated system.

This means that flexible bodies, fluid and gas mechanics, molecular physics and some other simulation and visualization areas are outside the scope of this work.

We define a simulation for our models as a particular execution of the software that given initial conditions and other input uses physical laws in order to reproduce the behavior of idealized physical models during some time span.

1.1 Visualization Requirements Induced by Simulation Goals.

When discussing design, simulation and visualization issues it is useful to be goal oriented, since different goals can be set up. The goals can be either application- or tool-driven.

The purpose of an application can for example be to optimize product properties and to reduce product development cost and time. This means that the simulation should be able to predict behavior of the system, or analyze what happened with some system in case of an unexpected behavior.

The purpose of simulation tools is to provide adequate technology necessary for applications. In particular, tools should be tested for capabilities and performance requirements – whether these can handle complex models and demanding simulations.

Of course, products will not become better and cheaper if they are just simulated. Attention should be paid to quality of simulation and availability of results. If there are too many inadequate assumptions and simplifications, the simulation will be cheap, and may be fast, inaccurate and useless. If the assumptions are adequate, all relevant details are taken into account and everything possible is measured with highest degree of accuracy, simulation requires very complicated mathematical analysis, complex programming models, and is able to produce numerically correct results. Such simulations may require extremely high computational power and proceed very slowly; often it is very hard to predict the time needed for computation. It might happen that these simulations are useless anyway - if the goal of simulation has been set incorrectly, or if results cannot be visualized in a comprehensible way.

The efforts of simulation software designers are focusing on finding a golden middle way between the two mentioned extremes, finding the trade-off between the computing resources and computation accuracy.

Several categories of general simulation goals are established; these goals can be described in a mathematical notation. Requirements for visualization of simulation results are depending on the goals of the simulation. Assume that x is some input data for a simulation function F, that produces simulation results $F(x)$. The simulation results for mechanical system are movement trajectories (position, rotation, velocity, acceleration) of bodies as well as force, torque, pressure and other dynamically changing values. The categories of simulation goals are the following:

Design optimization problem. For instance, the optimal size of the balls in bear-

ings is searched in order to minimize friction. The function F simulates movement of some mechanism with a parameter vector x. The function E estimates how good the movement trajectory is. The goal of simulation series is to find such x_m that $E(F(x_m))$ achieves its maximum. The function E has many parameters. Therefore engineers use interactive environments and visual aids in order to find the appropriate x_m. In applications for mechanical models it is very important to display forces, velocities and accelerations that occur in the simulated world. The simulation is not affected by the user after it starts, and usually the trajectories are analyzed *after* the results are computed.

Control system design. Assume that a robot that finds, grabs, moves and releases a detail should be designed. A control system for this robot should be developed. This control system should operate so that the robot performs the mission in minimum time and with maximum accuracy. The function E is an overall estimation of the quality of robot performance. A simulation function F for the robot includes F_m (a mechanical component) and F_c (a control component). The goal of the simulation is to find an algorithm F_c such that $E(F(x))$ is maximal. Visualization of such simulations should include display of trajectories of movements and comparison tools for such trajectories. A simulation can be affected by the user after it starts; in particular the user can feed different inputs (mission descriptions) to the control system. If the control system is designed so that it is able to compensate for errors in the movements of the machine elements, the numerical accuracy of computations can be reduced without excessively affecting the overall precision of the simulation.

Simplification problem. Quite often there exists a numerical method to find $f(x)$ which can be used as an approximation of $F(x)$, i.e. $f(x) \approx F(x)$, and $f(x)$ can be computed much faster. In particular, linearized results from finite element model computations are often used in order to reduce computation time. It is important to visualize the differences between $F(x)$ and $f(x)$, and to investigate (e.g. using interactive visualization) how these can be reduced.

Presentation. Artistic, emotional and educational side effects of simulation, i.e. evaluation of $F(x)$, is useful in many cases, such as computer games, movie industry, digital art, and human operator training. Numerical accuracy of computations can be reduced unless deviations between the simulated world and the real one can be perceived by the human during the simulation. However, color and texture choice is important in visualization. In order to use simulation interactively fast response time should be achieved. There is a trade-off between response speed, model complexity and accuracy.

Assessment of correctness. Often the simulation is performed just in order to check the correctness of $F(x)$. In certain simple cases there exist analyti-

cal solutions to mechanical problems, and $F(x)$ can be compared with this solution. Another variant of verification is comparison between results produced by two different software tools that perform simulation of the same mechanical model. Finally, it is possible just to observe dynamic model visualization and check whether it matches our intuition about model behavior. In an early stage of debugging of mechanical models this is a typical way of correctness assessment. Visualization requires forward and backward animation, trajectory visualization, rendering with varying level of details as well as interruption of running simulation. Simultaneously with assessing the correctness of $F(x)$ the user can check the correctness of the simulation tool: the correctness of simulation libraries, the correctness of the translation from the simulation language to machine code, as well as the correctness of the numerical solver.

1.2 External Factors Important for Simulation Software and its Life Cycle

A simulation life cycle is often extremely complex because it includes the life cycles of many software components, and the simulation goal might change during or after initial design. Here we consider several steps of the life cycle of dynamic model simulations, as well as assumptions used for these steps.

First a virtual prototype of a simulated product (robot, toy model, car, aerial vehicle) should be designed. It might be based on an existing product, or it can be result of an engineer's creative imagination. An informal application model is then created by removing unnecessary details and adding important features of the prototype. During this stage the objectives of the simulation should be taken into account. For instance, when some machine element is represented as a rigid body, we can obviously ignore its color and heat conductivity. However, if a simulation engineer expects that the model will be visualized for presentation purposes, color information should be preserved. If in some simulation thermodynamics is computed, the heat conductivity and heat capacity of the rigid body is quite important.

The informal application model can be expressed in the form of requirement specifications written in natural language, as sketches of model geometry, and collections of mathematical formulas that can be potentially useful for describing the considered phenomena (e.g. Newton's laws).

During the next stage the informal application model is converted into a formal model expressed in a modeling language. This language can be more general (a general purpose programming language, or an equation-based language), or less general and more application specific (a language for the description of models in a specific application area). The distinction between these tools is discussed in Section 2 in more detail.

It is important to know the model of the environment for the particular simulation. For instance, whether weather conditions should be taken into account in flight simulation. Or whether torques applied to joints of a virtual robot are directly

obtained from the virtual control system (electrical motors are not modeled) or the electrical properties of the robot motors are simulated too. It might happen that modeling the environment is much more complicated that modeling of the base product.

The I/O aspects, e.g. for sensors, are important too. For instance, during design of a control system the question ”how much the control system knows” should be clarified. For instance, whether the system or subsystem (e.g. an autonomous robot) has a positioning system (knows its location), vision (knows the positions of other objects) and communication (other components know about this subsystem).

During the next stage the formal model (written in a modeling language) can be translated into efficient code and simulated for certain starting conditions, parameters, and input data which become available after simulation starts.

As a rule the result of the simulation is visualized online (during simulation) or offline (after simulation), necessary assessment is performed by the user, certain starting conditions, parameters, and input data are modified, and after that the simulation is repeated many times with the ”human in the loop”.

1.3 Structure of the report

Simulations of multidomain physical systems are performed for many different purposes. Assuming that mechanical components are at the focus of attention of a multidomain model, *multibody system analysis* becomes one of the most important applications. In particular, *dynamic analysis* answers the question *how* the mechanical components move within a given (possibly, interactively steered) environment. *Static analysis* computes static equilibriums of mechanical systems.

In this work we concentrate on tools for *dynamic analysis* of mechanical models and visualization of results of such analyses. We are reviewing existing tools for this purpose as well as presenting our solution to this problem. We propose that mechanical models are designed in geometry modeling tools, and that these models are converted into a high level object oriented modeling language.

Therefore this paper includes an overview of existing techniques for dynamic analysis of multidomain systems, as well as some details of our approach. Sections 2, 3, 4, 5.1–5.3 and 7 describes the *background* of our work, including existing techniques, tools and requirements from the practice. Three main categories of tools are considered: simulation tools (Sections 2 and 3), tools for design of mechanical models (Sections 4 and 5), and visualization tools (Sections 7, 8 and 9). Sections 5.4–5.7, 6, 8 and 9 describe *our* techniques, tools and applications of these. Section 6 describes the overall architecture of our integrated system. Sections 2, 4 and 7 contain application requirements and an overview of existing techniques. Sections 3, 5, 8 and 9 describe our solutions to arising problems.

2 Overview of Approaches to Dynamic Simulation of Mechanical Models

Mechanical models, assuming certain approximations, are described as collections of rigid bodies with mass and inertia, connected with constraints that limit the freedom of movement as well as motors, springs and dampers that force bodies to move. The behavior of such bodies is described by Newton's laws which can be mathematically expressed as a system of ordinary differential equations (ODE) for every rigid body:

$$\dot{v} = \frac{F}{m}$$

$$\dot{x} = v$$

where v is a velocity vector, x is a position vector, m is mass and vector F is the sum of forces applied to the body.

The motion of the bodies is determined by starting positions and applied forces. The acceleration is defined by forces acting upon the bodies. Forces might be induced by gravity, mechanical constraints, drives (motors), springs, dampers, collisions, and user-defined active forces.

When the motion of the system is computed the results are the positions, velocities and accelerations of each body in the model for each time step.

Assuming ideal conditions, the actual motion trajectories of the bodies described by the model can be found as a solution of the corresponding system of (non-linear) algebraic and ordinary differential equations.

It is too hard, however, to find analytical solutions for non-trivial mechanical systems. Therefore, approximate numerical methods are used to solve the system of equations.

The problem of dynamic analysis of mechanical systems therefore consists of three stages:

- Translation of an idealized mechanical model into some computer readable format that can define the computations to be performed by the computer.
- Applying numerical methods to solve the equations.
- Representing (i.e. visualizing) the solution in a comprehensible form. The visualization consists of two-dimensional plots of variables and three-dimensional scenes with the mechanical model in motion as well as other required information.

The systems which are able to translate mechanical models into machine readable formats can be divided into two categories:

- MBST – multibody simulation tools, using a mechanical model description as input.
- EBST – equation based simulation tools, using a collection of algebraic and ordinary differential equations as input.

The main difference between these two group of tools is the level of abstraction used when input is given to the tools. MBST requires descriptions of mechanical bodies and their properties (see Section 2.1). EBST requires the specification of collections of equations (see Section 2.2). Modelica has the benefits of both approaches since mechanical elements are represented as class instances, and arbitrary auxiliary class instances can be added to the mechanical parts of a model (see Section 3).

There exists a spectrum of various methods to perform dynamic simulation and analysis of motion for mechanical models. Many of these methods have difficulties to interact with model components from other application domains, such as electrical systems, control systems, hydraulic systems, etc.

Since the goals of simulation are broader than just testing a mechanical model, some other potential properties of simulation tools become very important:

- Integration of mechanical systems with control systems and other multidomain components.
- Performing visualization during simulation and steering the simulation via the user's input.
- Using experiment descriptions (experiment setup scripts), e.g. for running series of simulations in batch and comparing the results from the simulations.

2.1 Multibody Simulation Tools

The purpose of multibody simulation tools is to perform various kinds of static and dynamic analyses of mechanical systems. Mechanical elements are fetched from libraries of ready components and their position and orientation is defined via a CAD-like 3-dimensional user interface. The connections between the elements are set up interactively using a CAD-like tool. Such tools are usually tightly coupled with the simulation tool, and a uniform graphical user interface and three-dimensional representation of mechanical parts is used both at the modeling stage and during visualization of simulation results.

2.1.1 ADAMS

Adams[2] is the world's most widely used multibody mechanical simulation software. It can be used in different configurations: as a full simulation package (Adams/View, Adams/Solver and other components) or as Adams prototyping capabilities integrated within CAD/CAM environments.

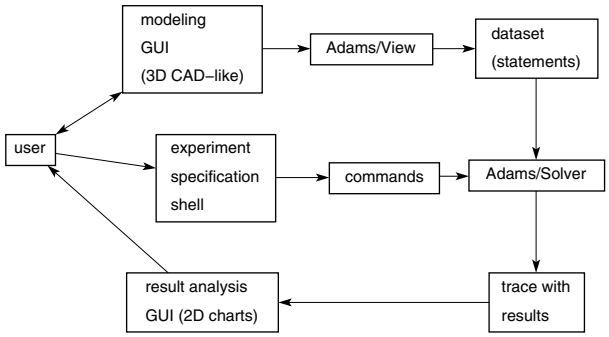


Figure 1: Cooperation between simulation engineer, Adams/View and Adams/Solver

Full simulation package In the full simulation package (see Figure 1) 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.

The initial model can be refined in order to apply more realistic features of the model.

- By default all the bodies are assumed to be rigid. Using a finite element modeling program and the Adams/Flex package, flexible body deformations can also be modeled. The flexibility of bodies and the dynamic behavior of the total system affect each other.

- Automatic collision detection between parts can be performed. For this purpose descriptions of part geometries are supplied.

When a series of computational experiments should be organized, parametric properties can be swept through a range of values, and parametric design relationships can be set up. The results of the model definition phase are stored as an *Adams dataset file*, essentially Adams program code in a proprietary declarative language. If necessary, external functions in FORTRAN can be specified in such definitions, and these functions are linked to the application when the models are simulated.

The Adams program code is normally generated via a graphical user interface and the code is not intended to be very "user-friendly". However, it is easy to modify already written code. The four statements described in Table 1 give some clues about the Adams program notation (words in *italics* are replaced by relevant numbers). Adams notation is based on statements, where integers are used to refer to other statements.

The expressions in Adams may contain any geometric, kinematic and dynamic values available in the model, logical conditions (`IF`-functions) as well as the current modeled time. External functions written in FORTRAN can be called within

PART/*partid*, QG=*x,y,z*, REULER=e_1, e_2, e_3, MASS=*m*, CM=*markerid*, IP=$i_1, ..., i_6$	Each *marker* in Adams represents a local coordinate system, i.e. a point in 3D space serving as an origin of a coordinate system rotated in a certain way. The specification above describes a rigid body of mass m located at *x,y,z* in the world coordinate system, and rotated by angles e_1, e_2, e_3. The center of mass of this body is a marker *markerid*; the inertia matrix is given as $i_1, ..., i_6$.
MARKER/*markerid*, PART=*partid*, QP=*x,y,z*, REULER=e_1, e_2, e_3	The reference point *markerid* is attached to the part *partid* at a certain position (in the coordinate system of the part) *x,y,z* and it has rotation e_1, e_2, e_3.
JOINT/*jointid*, REVOLUTE, I=*marker1id*, J=*marker2id*	A revolute joint is attached to two markers, which apparently belong to different parts and are joined at the same location in the 3D space.
MOTION/*motionid*, I=*marker1id*, J=*marker2id*, B3, FUNCTION=*expr*	There is a prescribed motion of the joint around the Z axis (i.e. the 3rd axis – therefore the keyword B3 is used), and the joint angle during this motion is defined by the expression *expr*.

Table 1: Four typical ADAMS statements.

these expressions. Also these functions can obtain values from "input channels" in order to cooperate with a control system. Such control systems are described outside Adams, using, e.g., the MATLAB tool.

The notation provides a relatively high flexibility of Adams models. The same model can be used for three different kinds of simulations:

- *Kinematic simulation:* All motions are already prescribed by the user. The system has zero degrees of freedom. All part positions can be computed from the motions. Forces are ignored.
- *Static simulation:* This simulation re-positions parts so that all forces are balanced. It finds the so called equilibrium configuration.
- *Dynamic simulation:* This simulation computes the combined effect of forces and constraints. It can be used for any number of degrees of freedom. The dynamic simulation package contains four different integrators. The user should tune these integrators by giving appropriate accuracy, integration step minimum and maximum, as well as other tuning parameters.

Adams in Control System Context. Integration of model simulations specified in Adams with external input and output (such as control systems or operator con-

trol graphical user interface) requires some special adjustments in the model structure. In order to integrate Adams with control systems, the mechanical model is exported from Adams into a special module. Then this module can be used in SIMULINK, and control system components should be designed and coded in SIMULINK.

Visualization. During simulation or after the simulation terminates (in Adams/Solver) the user can see dynamic visualizations of machine elements. However there are no options to steer an ongoing simulation. Adams has a rich set of constructs helping to run a series of simulations as a batch. In the 3D visualization the results of two (or more) simulations can be displayed and compared. However, this is only possible if all the simulations have the same number of steps and the same step size. This limitation makes it difficult to visualize the results from simulations that require varying time steps.

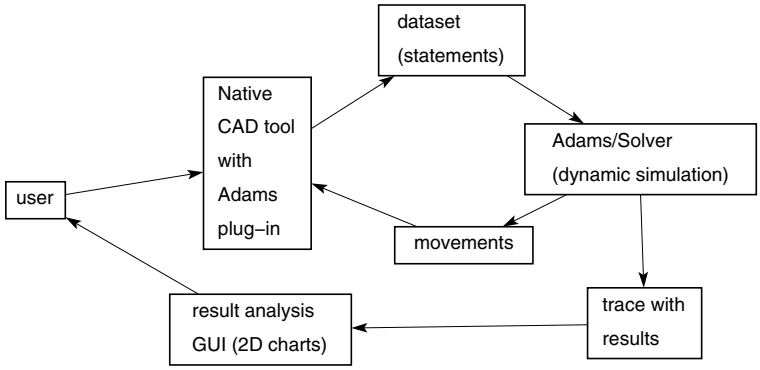


Figure 2: Cooperation between a simulation engineer (user), and Adams plug-in embedded in a CAD application.

Adams Plug-in Integrated with CAD Environment. When Adams is integrated with CAD tools (see Figure 2), the user works in his or her native CAD environment where parts and assemblies are normally designed. 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.

Integration of Adams with CAD tools from Autodesk, Unigraphics, SolidWorks and many other products is available.

The Mechanical Designer simulates the model and the model movement is displayed online, during the simulation. When more detailed modeling is necessary, the Adams program code can be saved, modified and later used with the full Adams simulation package.

Unfortunately, the integrated variant of the Adams package has some limitations regarding the complexity of the simulation system. In particular, expressions that are used to specify the force values cannot call external functions, and cannot interface with external programs.

Example. An experiment has been done with the Mechanical Designer, a plug-in utility for the SolidWorks CAD tool. A double pendulum has been constructed (see Figure 3). The pendulum consists of two bars and one box in a fixed position; these are connected by a revolute joint, i.e. a joint with one rotational degree of freedom. Such revolute joints are labeled as door hinges in the schematic presentation of this mechanism. The tool generates code in Adams dataset format:

```
ADAMS/View model name: Designer
UNITS/, FORCE=NEWTON, MASS=KILOGRAM,
  LENGTH=MILLIMETER, TIME=SECOND

PART/1, GROUND
MARKER/54, PART=1, QP=-248.06, 147.27, -162.19

PART/2, QG=-595.51, -50.91, -162.19,
  REULER=119.700423D, 90D, 180D, MASS=0.64, CM=10000,
  IP=8618.6, 8618.6, 170.66, 0, 0, 0

MARKER/50, PART=2, QP =0,0, 0,  REULER=0D, 90D, 60.29D
MARKER/53, PART=2, QP =0,0, 400, REULER=0D, 90D, 60.29D
MARKER/10000, PART=2, QP=20, -20, 200

PART/3, QG=-611.96, -14.45, -122.19,
  REULER=-65.71D, 90D, 0D, MASS=0.64, CM=10001,
  IP=8618.66, 8618.66, 170.66, 0, 0, 0
MARKER/49, PART=3, QP=40, -40, 0, REULER=180D, 90D, -114.28D
MARKER/10001, PART=3, QP=20, -20, 200

JOINT/1, REVOLUTE, I=49, J=50
JOINT/2, REVOLUTE, I=53, J=54
ACCGRAV/, JGRAV=-9810

END
```

2.1.2 Working Model 3D

Working Model 3D [29] is an advanced tool providing dynamic analysis within an integrated modeling and simulation environment. It supports design of simple mechanical models, as well as import of CAD models from SolidWorks, SolidCad and Mechanical Desktop. 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 (Figure 4). 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.

Models in Working Model 3D (version 4.0) cannot interact with any external software during simulation (see also Section 4.2.2). There is no way to attach mul-

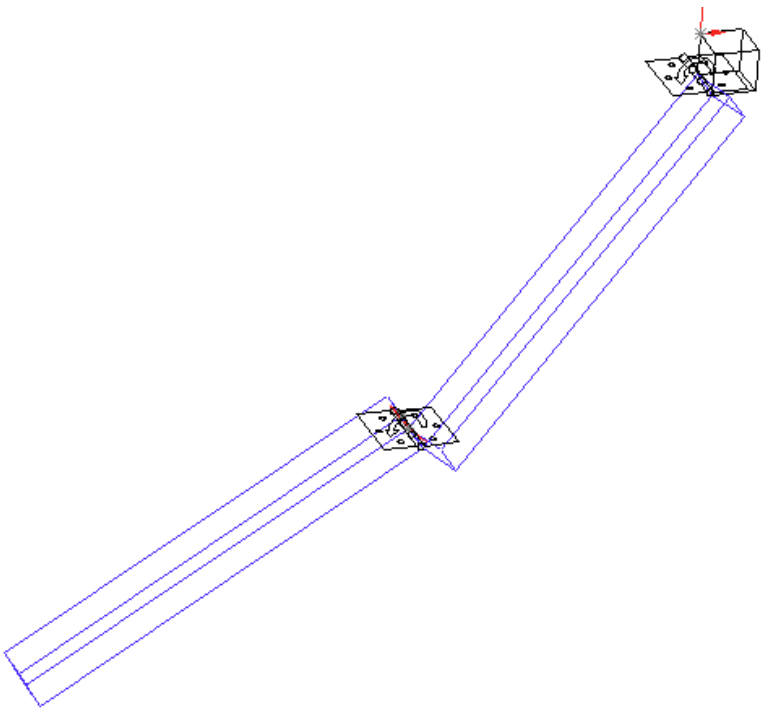


Figure 3: Pendulum model designed in SolidWorks with Adams plug-in.

tidomain components, and no options to attach controls to steer the model during running simulation. There is no experimentation language and no batch simulation is possible.

Prescribed motion, or prescribed external forces can be described in the form of mathematical expressions. Such expressions can be arbitrarily complex and nested. The expressions can include other variables of the system (position, velocity of the bodies) as well as the current value of time.

In the documentation of Working Model 3D the numerical results obtained for several simulations are compared with results computed by analytical methods as well as the results of identical mechanical systems constructed in different tools, e.g. ADAMS. The comparison shows a high accuracy of the results.

2.1.3 Integrated Environments for Computer-Based Animation (3D Studio Max)

Computer animation software (such as 3D Studio Max and Maya) provides a large variety of features in order to make computer-based animations more realistic in movements. For this purpose these tools can model mechanical systems (see Figure 5) and compute trajectories of rigid and flexible bodies according to physical laws. These trajectories can be saved in the proprietary format of the computer animation tools. Normally there are no options for external steering of the mechanical model

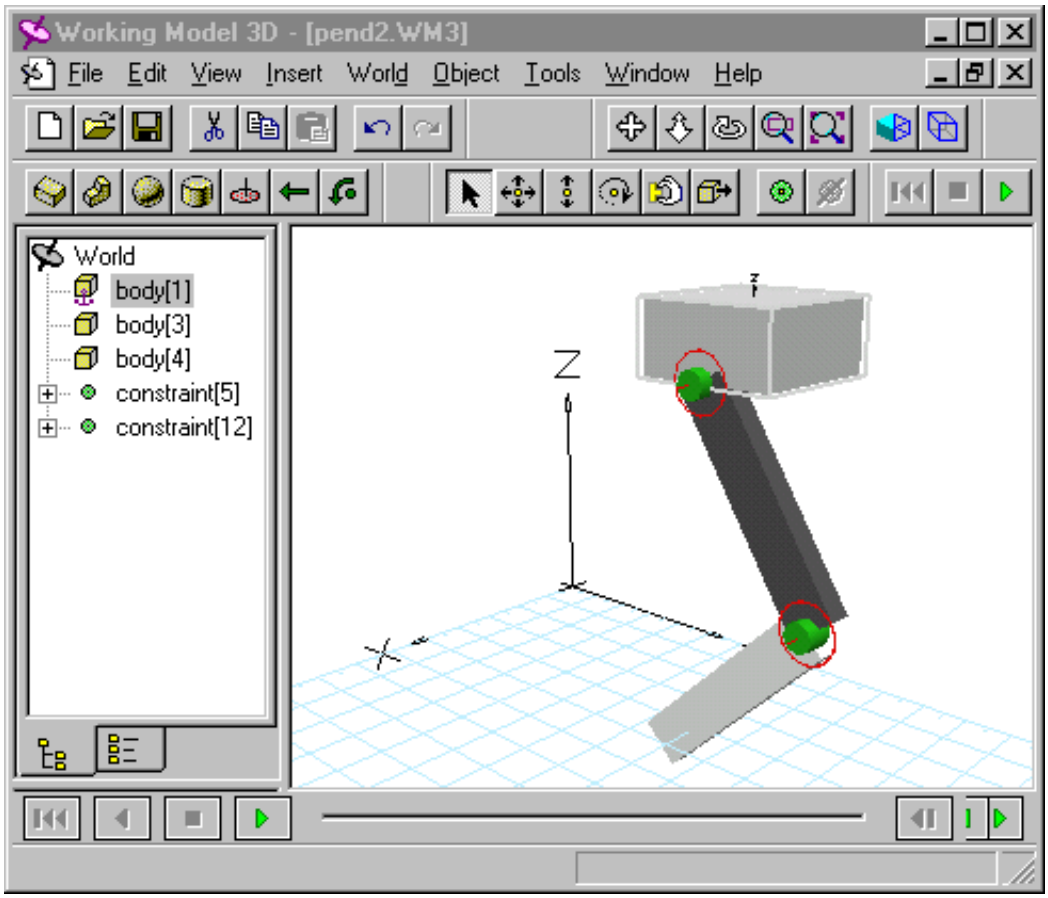


Figure 4: Double pendulum model in Working Model 3D

during computation of the trajectories.

Instead, computer animation packages can quickly compute rough approximations of dynamic movements.

In 3D Studio Max ([5] , see also 4.2.3) , mechanical or similar models with moving parts are constructed in a hierarchical fashion. All the objects of a mechanism should belong to a directed acyclic graph (see Figure 6). The edges of the graph define relations between parents and children and there is one and only one parent for every node (except the root of the hierarchy which has no parents).

The lock (constraint rules) of the children define how it can be translated and/or rotated with respect to the parent (see Figure 7). If, for instance, `Rotate/X` and `Rotate/Z` are selected, the child object can only rotate around its local `Y` axis. If `Move/X` and `Move/Z` are selected, the child can only move along the Y axis.

By combining different degrees of freedom all typical joint types can be modeled (revolute, prismatic, spherical, cylindrical, etc.). The joint properties are not clearly visualized during model editing, and it can be difficult to choose correct

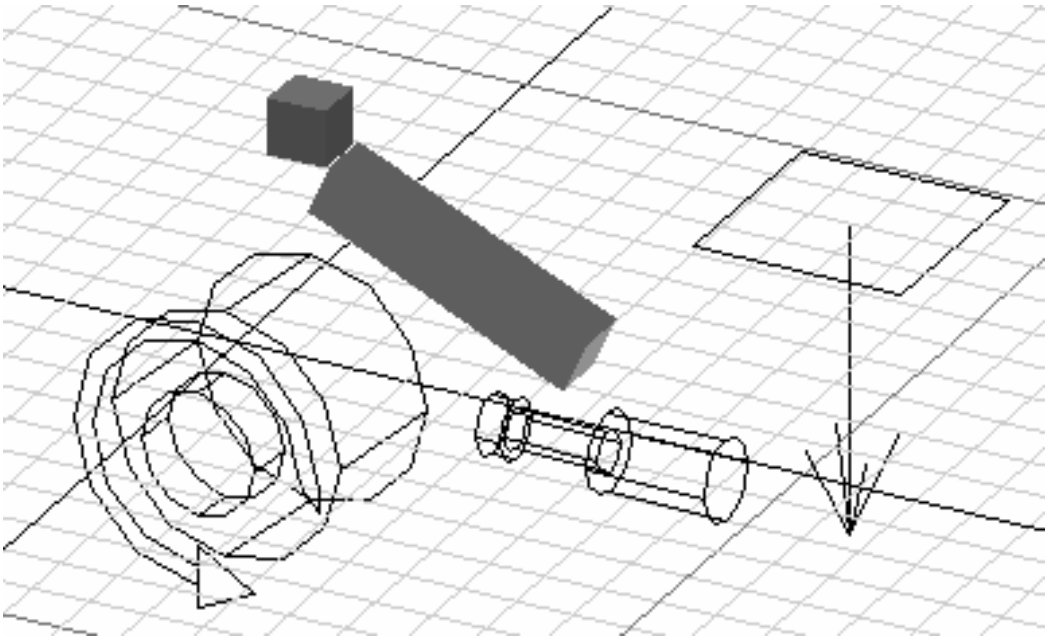


Figure 5: A pendulum model in 3D Studio Max

directions of constraints. External force (*Push* option) and torque (*Motor* option, as well as environment settings (*Gravity* and *Wind*) can be applied to the objects. These objects are included in the hierarchy of a 3D Studio Max scene (see Figure 6)

The 3D Studio Max *Dynamics* subsystem analyses collisions and generated motion taking collision response into account. Collisions can be computed at different level of accuracy (using just bounding box, bounding sphere, or using a detailed mesh of the object). There exist material properties, which are applied to the object: collision restitution coefficient (bounce coefficient), static and sliding friction coefficient. These numbers are in the [0.0,1.0] interval.

The dynamic capabilities of 3D Studio Max do not include springs and dampers. Kinematic loops cannot be constructed because the kinematic skeleton always has tree structure. Tests we have run show that computations are very approximate and sometimes they produce non-adequate results (objects penetrate, or locks are broken).

2.2 Equation-Based Simulation Tools

Equation based simulation tools are intended for modeling and simulation of almost arbitrary dynamic systems. Such tools are able to integrate mechanical systems with, for example, control, electrical, hydrodynamic and user-interface system components. It is not very feasible, however, to perform modeling of complex systems by writing equations directly. Instead, a structuring approach should be

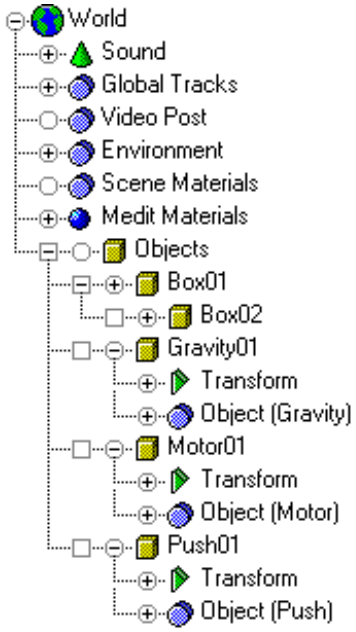


Figure 6: Hierarchy of objects in the model of a pendulum in 3D Studio Max

used. The modeled systems are designed in several layers. Physical phenomena are usually described by collections of variables and equations. Such a collection can be reused. Therefore, the block approach (replaced by object-oriented approach in modern systems) should be used. A block contains a collection of equations which is reused many times in the same model or in different models. Blocks can be connected, and connections specify additional equations. Furthermore, blocks can be parameterized, and the parameters can serve as coefficients in equations. Finally, blocks can be nested.

Here we discuss three tools for equation based modeling: SIMULINK (Section 2.2.1), Mathematica (Section 2.2.2) and Modelica (Section 3). These tools use different level of abstraction for equation blocks.

2.2.1 SIMULINK/Systembuild

SIMULINK is a graphical interactive environment integrated with the MATLAB tool. MATLAB is an interactive numeric programming tool with command shell based interaction.

SIMULINK is a graphical environment that exclusively uses input/output blocks for model specification. There is no specific support for mechanical modeling. Multibody systems when translated to SIMULINK lose their "kinematic structure" and the designer must provide the whole chain of mathematical transformations needed for computations.

For instance, the double pendulum becomes a very complex model in SIMULINK

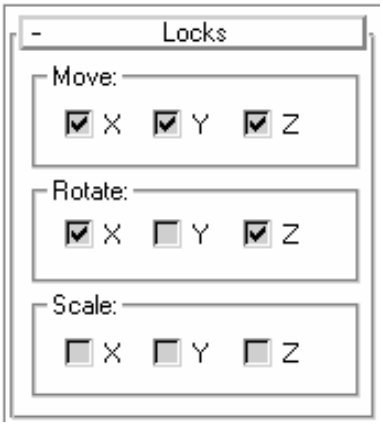


Figure 7: Lock (i.e. degrees of freedom) specification for a pendulum model in 3D Studio Max. The child object can rotate around the `Y` axis only. Scaling is not locked, but it cannot happen in dynamic analysis.

(Figure 8). It can be seen that the two pieces of the double pendulum cannot be identified as two separate independent blocks in this diagram. The same problem is revealed when SIMULINK is used for simulation of electrical circuits. Therefore it is difficult to use SIMULINK for physically-based simulations: the model structure does not in general correspond to the physical structure of the system.

In order to visualize computation results as 2D plots some default MATLAB graphical functions can be used. MATLAB also supports 3D plotting and display of 3D scenes. However there is no specific support for visualization of simulated mechanical models.

2.2.2 Mechanical Packages for General Purpose Computer Algebra Systems

Computer algebra systems such as Mathematica[57] are intended for expressing mathematical function definitions and equations, and providing symbolic and numerical tools for solving such systems. There exist libraries of functions and equations designed for various application areas. These packages are able to integrate mechanical models with control system models and models from other application domains. However it is difficult for the user to describe mechanical models since the tools do not provide any graphical user interface to specify the body geometry and assembly information. Also, rules for constructing an application for dynamic analysis of mechanical systems are quite complex.

Mechanical Systems for Mathematica Mechanical Systems is an application package written in the Mathematica language that can be used within the Mathematica environment. There exist subpackages for two- and three-dimensional kinematics. Mathematica function calls are used in order to describe a mechanism consisting of bodies and constraint specifications. The tool can compute both

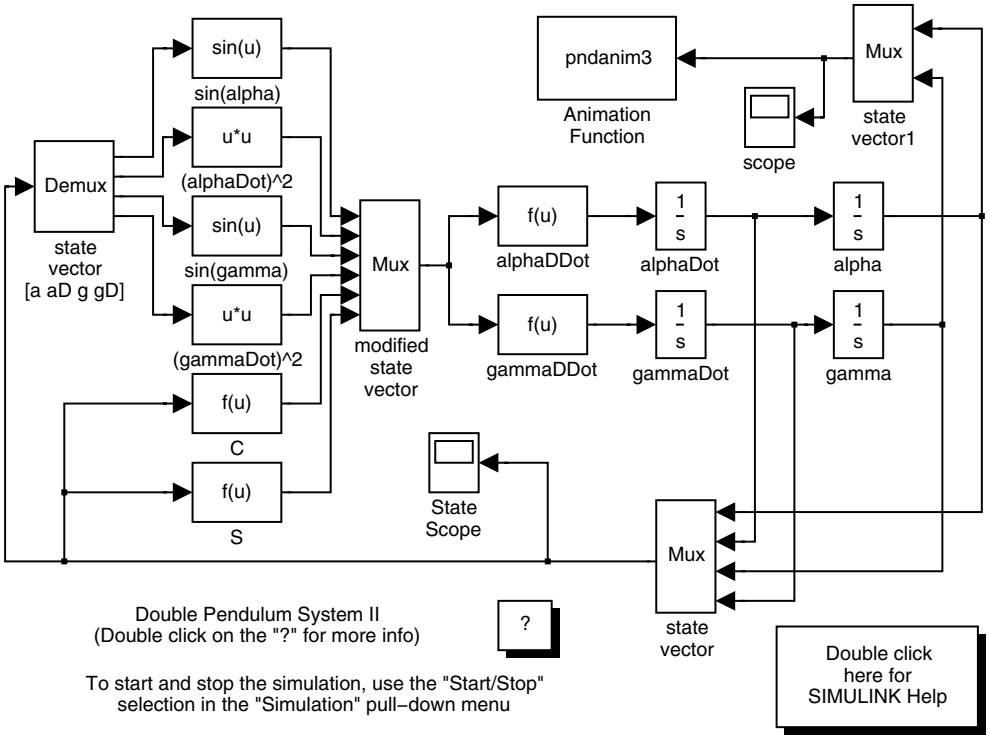


Figure 8: Sample mechanical model (double pendulum) described in SIMULINK notation. This example is taken from the SIMULINK demo collection.

forward and inverse kinematics. Positions, velocities and accelerations can be obtained from the simulation.

Mechanical Systems does not contain any specific support for friction, collision detection and collision response. The system does not take the surface geometry into account. These details are left to the user who can specify the values and directions of external forces applied to the bodies. An advantage of this package is that functions of arbitrary complexity available in Mathematica can be used for description of external forces.

The tool creates symbolic representations of all equations in the mechanical simulation model. There equations are, however, solved numerically, using an adaptive ODE solver.

The solution cannot be affected interactively during simulation. Since Mathematica is an interpreted language the performance of such simulations is relatively low.

Visualizations of mechanical model simulations can be designed by the user. For this purpose standard Mathematica 3D graphics functions are used. The output can be accumulated and transformed into animation sequences.

3 Using the Modelica Language for Dynamic Analysis

In Section 2 we discussed properties of multi-body simulation tools and equation-based simulation tools. Modelica is an equation based system. However, it can accept descriptions of mechanical models. In this work we have designed an interface between a CAD tool and Modelica that allows simulating arbitrarily complex mechanical models by translating them to Modelica equations. Therefore we can claim that now, with these additional technologies, Modelica integrates the best properties of multi-body simulation and equation-based simulation approaches.

3.1 Modelica

Modelica [35] is a new object-oriented equation-based modeling language. Models described in this language are well-structured collections of variables, ordinary differential and algebraic equations, and functions.

The methodology of object-oriented equation-based modeling have been invented by Hilding Elmqvist in his dissertation [14] and initially implemented in the Dymola language and tool [15]. This methodology has been further developed by the international Modelica Design Group [35] which includes participants from both universities and industry. The result of this development has been reflected in Modelica language specification [36]. Tutorial and rationale for the language can be found in [37].

The first version of Modelica, version 1.0, was announced in September 1997. The most recent, version 1.3, was released in December 15, 1999.

Two papers written by Martin Otter, Hilding Elmqvist and Sven Erik Mattsson [17, 43] cover the major properties of Modelica and hybrid modeling technique invented by these authors. The hybrid modeling in Modelica integrates discrete and continuous modeling methods in the same language. Object-orientation techniques used in Modelica are discussed in [25]. Our current Modelica activities are reflected in [40].

The language unifies and generalizes previous object-oriented modeling languages, e.g. such as Dymola [15], Omola [4], ObjectMath [24, 41], Smile [22], NMF [49], etc. Compared with the widespread simulation languages available today this language offers three important advances:

- non-causal modeling based on differential and algebraic equations;
- multidomain modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model;
- a general type system that unifies object-orientation, multiple inheritance, and templates within a single class construct.

Modelica is a general standard notation which can be used for standard application domain libraries and for applications that use these libraries. Tools and

environments are built to comply with this standard.

Currently there are two design, simulation, and visualization environments for Modelica:

- *Dymola tool with Modelica Language Support*, developed by Dynasim [15] and
- *MathModelica*, developed by MathCore[31].

Our *contributions* to using the MBS library are the following:

- CAD to MBS conversion (Section 5) that enables creating complex mechanical models quickly and correctly;
- MBS visualization (Sections 8 and 9) that makes it possible to perform detailed verification of the movements of modeled bodies.
- Collision detection and response [20] that makes it possible to conveniently produce more realistic models than before.

3.2 Basic Features of the Modelica Language

The Modelica language is primarily intended for the description of systems that have variables with continuously changing values. Additionally, discrete variables are supported giving the language hybrid modeling capabilities. Models written in Modelica specify which variables exist in the mathematical model, and how the variable values are related. Values of continuously changing variables are usually represented as floating point values. The floating point variables are usually involved in such models. The type of these is denoted as **`Real`**. Relations between the variable values are described by differential-algebraic equations (DAE), such as

$$x + 1 = \cos(Time)$$

or

$$y' = x$$

The DAE of Modelica system are formulated, according to [36] as system

$$0 = f(\dot{x}, x, y, u)$$

where x are unknowns that are differentiated, y are variables that are not differentiated, u is known input data. In Modelica DAEs may have discontinuities and the structure of a DAE system may change at certain points in time.

In the general form such hybrid DAE system are mathematically described by a set of equations:

- Residue Equations: $0 = f(\dot{x}, x, y, t, m)$
- Monitor Functions: $z = g(\dot{x}, x, y, t, m)$

- Update Equations: $0 = h(\dot{x}, x^{known}, x^{reinit}, y, t, m, pre(m))$

Additionally these functions depend on simulation parameters p which are constant during simulation and input data $u(t)$. The different classes of variables are described below:

- t – independent time variable,
- $x(t)$ – differentiated variables, x^{known} – known part of vector x, x^{reinit} – reinitialized part of x,
- $y(t)$ – non-differentiated (algebraic) variables
- $u(t)$ – known input data
- m – discrete variables, $pre(m)$ – values of discrete variables before update at most recent event.

Mathematical models of complex physical system usually contain many variables and equations. Since most physical systems have a natural object structure this fits well with the object-oriented programming paradigm. All the equations, variables, and constants related to some class of objects can be encapsulated in the same class. By applying inheritance, other classes can reuse equations, variables and constants. Variables can be instances of other classes.

The execution of a Modelica model results in a solution of each unknown variable (e.g. x) as a function of time, $x(t)$, where $t \in [t_{start}, t_{end}]$. In order to find $x(t)$, usually the initial value $x(t_{start})$ has to be supplied.

Examples of problems that can be formulated in the form of DAEs and solved by Modelica are:

- Positions of point masses in space can be found when acceleration is given.
- Control system behavior can be computed when continuous input is given.
- Dynamic motion of mechanical systems can be computed when the masses of included bodies as well as constraints and forces acting on them are given.

The distinctive feature of Modelica is its non-causal programming model. The programmer does not specify the order of evaluation of elements in the model. The Modelica execution semantics does not depend on the order in which equations are written. Instead, before the simulation starts, the compiler finds the appropriate order of evaluation for every instantiation of each equation (if such an order exists).

For instance, the equation

$$a = b + c$$

can be used as

$$a := b + c$$

or

$$b := a - c$$

or

$$c := a - b$$

depending on the data flow context where the corresponding variables a, b, and c are instantiated. Certain equation systems cannot be converted into assignment statements since they are mutually dependent on other equations, forming a system of simultaneous equations.

It should be noted that an equation belongs to some class. Each class can be instantiated several times within the same application. Therefore the same equation can be instantiated several times. Depending on the context an equation can be converted into different explicit forms, as for the three examples above. Furthermore, several equations can occur in a system of linear or non-linear differential-algebraic equations (DAEs). A relevant method for equation system solution is chosen, and a, b and c are found using this method.

Modelica classes are intended for reuse. The classes are usually collected into libraries. There exist libraries for electrical, mechanical, thermodynamics, hydraulic and other application domains. Libraries serve as collections of knowledge for each application domain. Equations based on the laws of physics, as well as rules describing how the classes can be applied are stored in the libraries. Unfortunately not all information describing the rules for correct library usage can be stated formally within a model. Some rules are described in documentation (as plain text), and it is the user's responsibility to follow them.

3.2.1 Implementation of Model Simulation

Instances of classes in a model, including equations, are translated into a flat set of equations, constants and variables. After flattening, all the equations are sorted according to the partial order of data dependence.

The symbolic solver/simplifier performs a number of algebraic transformations to simplify the dependencies between the variables. It can also solve a system of differential equations if it has a symbolic solution. Finally, C code is generated which is linked with a numeric solver (Figure 9).As the result of executing this code functions of time (t), e.g. `R2.v(t)`, can be computed during a time interval [t_0,t_1] and displayed as a graph or saved in a file. This data presentation is the final result of system simulation.

3.3 Introduction to Modelica Syntax

Despite the fact that our discussion in the coming sections is devoted to mechanical models, we have chosen an electrical model example to introduce Modelica since this example is very simple and easy to understand but still demonstrates major Modelica features.

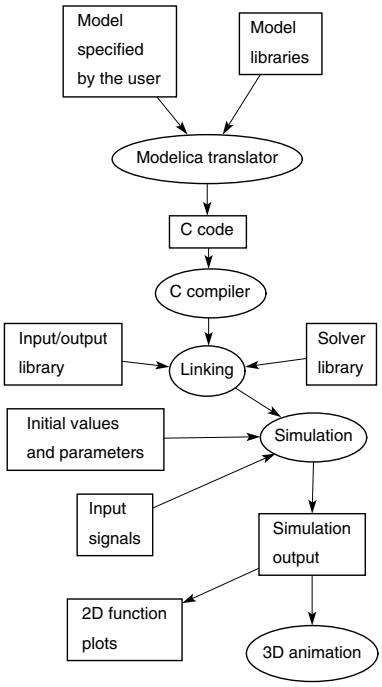


Figure 9: Components of the Dymola design, simulation and visualization environment.

As an introduction to Modelica we will present a model of a simple electrical circuit. Our goal is to describe major syntactic features of the Modelica language, which can be used in applications in various domains (such as e.g. electric, mechanical or chemical). A detailed description of this example can be found in [37].

3.3.1 Introduction to the library of Electrical Components

An electrical circuit model consists of electrical components. Most components have one or two pins. All components with two pins can inherit variables and equations from a general class `TwoPin`. Such components are e.g. `Resistor`, `Capacitor`, `VoltageSource` and `Inductor` (see Figure 10).

Each class has a corresponding icon (see Figure 11) designed to be easily recognizable by engineers in the corresponding application area.

In addition to basic elements of electric circuits there also exist Modelica classes for other elements, e.g. switches, diodes, current sources, voltage sensors, transformers, and electromotoric sources.

3.3.2 Example

Assume that the example model (Figure 11) consists of a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of such components are

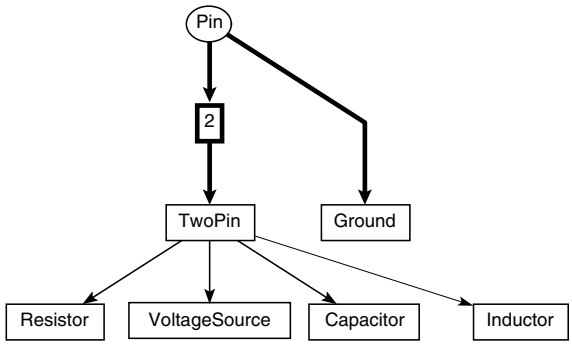


Figure 10: Inheritance diagram for some electric components. The graphical notation is explained in Figure 14.

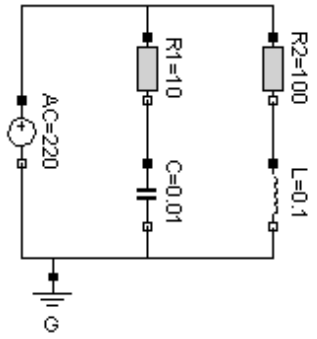


Figure 11: Example circuit structure using Modelica graphical notation associated with Modelica classes.

available in the Modelica standard class library for electrical components.

A declaration like the one below specifies `R1` to be an instance (i.e. an object) of the standard library class `Resistor`. This class contains several variables, constants (having the prefix `parameter` in Modelica), and equations. The equality `(R=10)` sets the default value of the resistance parameter, `R`, to 10 (i.e. `R1.R` is `10`).

```
Resistor R1(R=10);
```

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); connect(R1.n,C.p);
connect(C.n,AC.n); connect(R1.p,R2.p);
connect(R2.n,L.p); connect(L.n,C.n);
connect(AC.n,G.p);
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 the **`connect`** statements above `.n` means the *negative* pin of the relevant component; `.p` is the *positive* pin of the component. Each **`connect`** statement corresponds to an electric wire between the specified pins.

3.3.3 Using connectors

In this section we consider definitions of the Modelica language constructs used above.

The components (`Resistor`, `Capacitor`, etc.) are subclasses derived from the class `TwoPin` which in turn contains two `Pin` objects:

```
class Voltage = Real; //Voltage, Current and Resistance are used
class Current = Real; //the same way as Real
class Resistance = Real;

connector Pin
 Voltage  v;
 flow Current i;
end Pin;

class TwoPin
 Pin p, n;       //positive and negative pin
 Voltage v;      //voltage difference between the pins
 Current i;      //current through the element
equation
 v = p.v - n.v; //voltage difference
 p.i = - n.i;
 i = p.i;        //current i going inside
end TwoPin;

class Resistor extends TwoPin
   // inherits p, n, v, i and all equations from TwoPin.
  parameter  Resistance R=1.0;
equation
     v=R*i; // Ohm's law
end TwoPin
```

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 an equality for v and a flow balance for i, since there is a prefix **flow** in the declaration of i. The two equations resulting from the **connect** statement are:

```
Pin1.v = Pin2.v
```

and

```
Pin1.i + Pin2.i = 0 .
```

Only connector instantiations can be connected. If several connectors are connected by **connect** statements, it implies equality of all non-flow variables:

```
AC.p.v=R1.p.v=R2.p.v
```

and balance of all flow variables:

```
AC.p.i+R1.p.i+R2.p.i=0.
```

Modelica and its standard libraries for electrical models provide short, clear, extensible and concise notation for such models.

During system simulation the variables i and v evolve as functions of time. The solver of algebraic and differential equations computes the values of all variables in the model during the specified simulation time interval.

3.4 Introduction to the Modelica Multibody System Library

To facilitate mechanical system modeling we use one of the existing Modelica component libraries, the Multibody System (MBS) Library. This object-oriented library has been invented by Martin Otter in his dissertation [42]. Overviews of the library are given in [44, 45]. This library has been first implemented in the Dymola language. Later this library has been translated to Modelica.

This section, as well as Section 3.5 aim at giving a gentle introduction into using the MBS library for simple mechanical models. More details can be found also in MBS documentation, included in [15]. Some MBS terminology will be used later in Section 5 as well as in our reports [19, 20].

The purpose of the MBS library is to simplify modeling and dynamic analysis of mechanical models. Given an idealized mechanical model and external forces, a Modelica simulation can compute the position of each body as a function of time. For this purpose each physical body, joint, external force, etc. of a multibody system is modeled as an object, i.e. an instance of the corresponding library class. The object contain equations of motion, as well as equations for conversion of physical quantities (position, speed, etc.) between different coordinate frames used in a multi-body system. Since the library works with bodies that have some extent, their dynamics cannot be simplified to the dynamics of point masses. The rotation of bodies, as well as the speed and acceleration of rotation are taken into account.

The MBS library operates on idealized rigid bodies described by mass and inertia tensors. Massless joints and bars can be set up between bodies with mass.

Motors and other driving and braking forces and torques can be applied to the joints. Also, springs, dampers and external forces can be modeled.

The major areas that MBS does not handle (and is not intended to handle) are flexible bodies (bodies with deformations). This requires solution of partial differential equations which currently cannot be formulated in Modelica. Furthermore, these equations require very high efficiency in computation, as well as very detailed control of solution method choice, which is missing in current implementations. However there is ongoing research in this area[26, 51].

The MBS library is a collection of Modelica classes. In order to do modeling of a mechanical system certain classes need to be instantiated in the user's model. These should be properly connected, according to certain rules. Mechanical system models can be used for several purposes, since there are different ways to analyze dynamic systems. However, the most typical use of such models is to analyze the movements of bodies resulting from the application of certain forces and torques, as well as other changes in the environment. The forces and torques can have different origins: these can produced by drive trains, electrical motors, controlling devices, or combination of those. Additionally, external forces and torques can be directly applied to bodies in mechanical models.

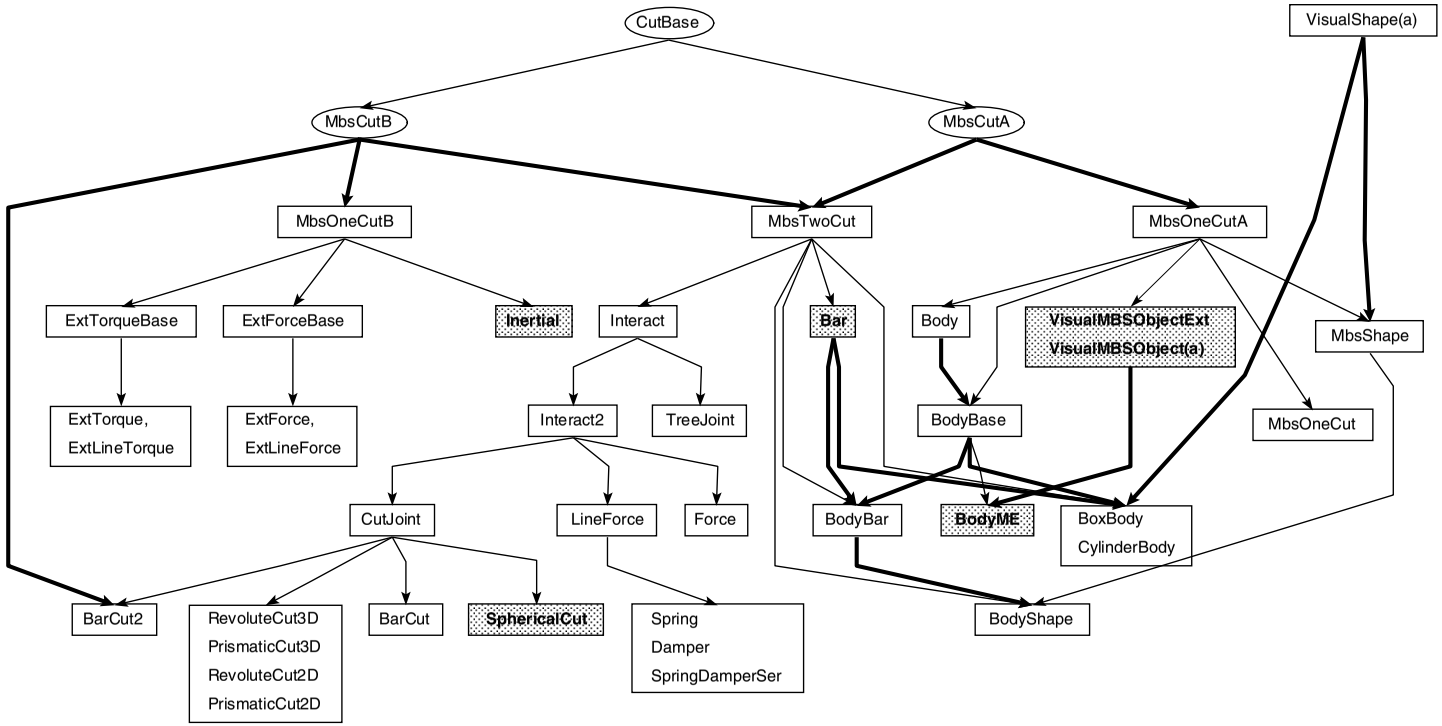


Figure 12: Classes of the MBS library (except descendants of `TreeJoint`). The graphical notation is explained in Figure 14.

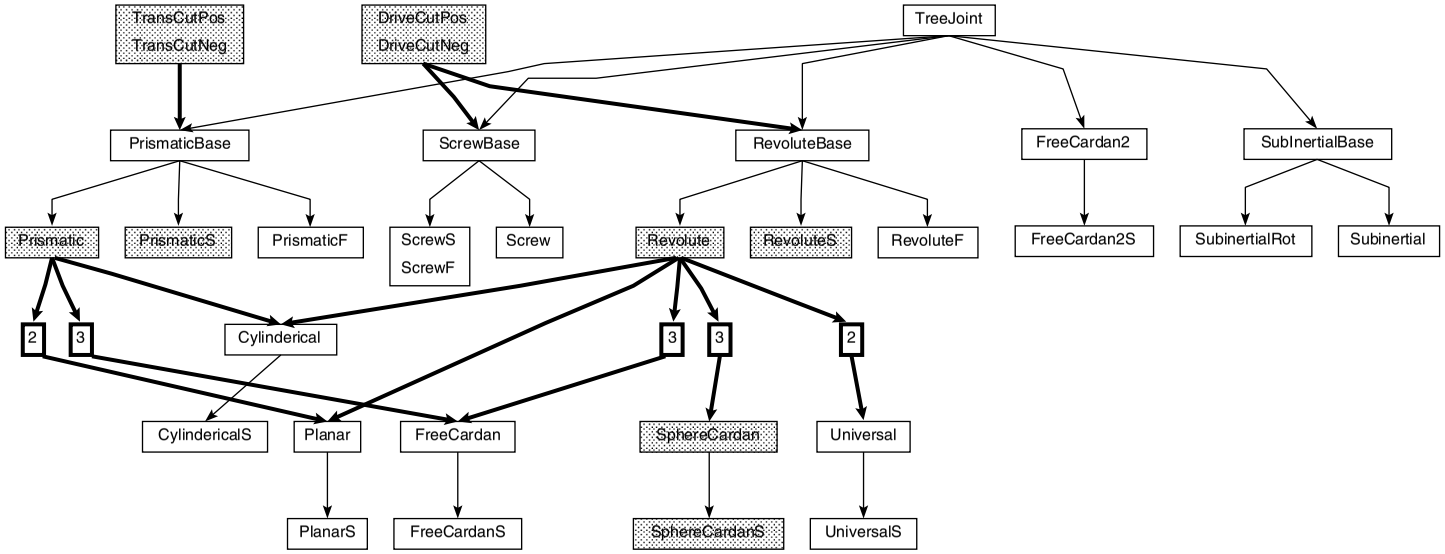


Figure 13: Classes of the MBS library: `TreeJoint` and its subclasses. The graphical notation is explained in Figure 14.

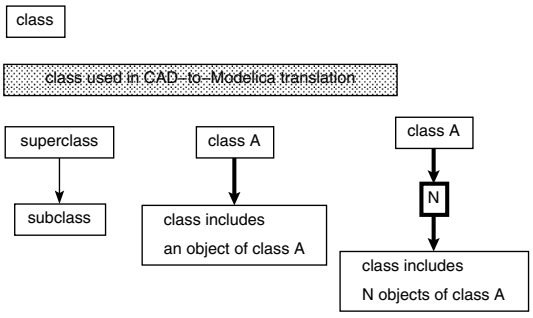


Figure 14: Notation for class diagrams (Figure 12 and 13)

The figures 12 and 13 contain the inheritance and aggregation hierarchy of the 73 classes of the library. We use non-traditional notation for inheritance and aggregation relations in these diagrams. This notation is explained in Figure 14. The links used in the diagrams are more convenient for libraries with multiple levels of inheritance. In aggregation we emphasize how many subcomponents are used in the class. Figure 13 is continuation of the Figure 12 and it displays all classes that inherit from the class `TreeJoint`.

Our detailed description covers ten major classes only. Most mechanical systems can be constructed from these. Many other classes are variations and optimized combinations of the basic classes.

3.5 Using the MBS library

3.5.1 Kinematic outline

There can be different approaches to constructing an MBS model for a mechanical model using a collection of Modelica classes. If a CAD-to-Modelica translator is used, there is no need for manual construction of the MBS model. Otherwise, the construction process depends on the context, in particular what type of analysis that will be applied to the model, and whether simulation efficiency should be taken into account. Here we suggest here the process of stepwise refinement of an MBS model first using so called kinematic outline (kinematic skeleton), and later introducing details of model dynamics and visualization based on this outline.

The kinematic outline is a graph consisting of nodes, connectors belonging to the node, and edges between the connectors. The *nodes* of the outline correspond to mechanical bars and joints. The *edges* of the outline are connections between them. The edges are attached to the nodes in connectors that correspond to mechanical connectors.

In many cases the graphs are acyclic. However there exist mechanisms that correspond to graphs with loops. Such loops are called *kinematic loops* and require special treatment when MBS library classes are used. More details are discussed in Section 5.4.1.

Mechanical connector `MbsCut`[1] (also referred as `MbsCutA` or `MbsCutB`) specifies a local coordinate system (a local frame of reference).

A bar (**`class`** `Bar`) describes the (static) position of one mechanical connector with respect to another. A joint specifies how one mechanical connector can move with respect to another. The possible movements are constrained by certain degrees of freedom: rotational (**`class`** `Revolute*`[2]), translational (**`class`** `Prismatic*`) or their combinations.

The kinematic outline consists of the inertial system, the bars, and the joints that define the kinematic features of the system. Given angles of rotation for revolute joints and the length of translation for prismatic joints, the MBS system can compute the position of any mechanical connector. For this purpose it uses the coordinate transformation matrix between adjacent mechanical connectors.

The following classes are used for constructing the kinematic outline:

`connector MbsCutA` Each instance of `MbsCutA` describes a local coordinate system (also called *frame A*). This connector contains several non-flow variables:

- **`Real`** `Sa[3,3]` - rotation matrix describing frame A with respect to the inertial frame.
- **`Real`** `r0a[3]` - vector from inertial frame to A.
- **`Real`** `va[3], wa[3], aa[3], za[3]` - translational and angular velocities and accelerations

According to Modelica semantics, if connectors from several objects are connected, the non-flow variables with the same name become constrained to have identical values through equality equations. This corresponds to a common local coordinate system used by several model components.

Connectors also contain two flow variables:

- **`Real`** `fa[3]` - resultant force acting on A.
- **`Real`** `ta[3]` - resultant torque acting on A.

According to Modelica semantics, if n connectors from several objects are connected together, the sum of all **`flow`** variables is equal to zero, i.e. $F_1 + F_2 + ... + F_n = 0$.

`connector MbsCutB` This connector has exactly the same variables, but uses the *B frame*, and the names of its components are: `Sb`, `r0b`, `vb`, `wb`, `ab`, `zb`, `fb` and `tb`.

[1] The MBS library has been translated from Dymola to Modelica. A Modelica connector corresponds to the **`cut`** construct in Dymola.

[2] The notation `Revolute*` means that there are several similar classes: `Revolute`, `RevoluteS` and `RevoluteF`.

The Connection Rules There is a rule regarding connections between the MBS library classes discussed further. Connector `a` can be connected only to connector `b` of other object[3].

This rule implies the tree-like structure of the kinematic skeletons. When a connection between two connectors is set up, the coordinate frames that the two connectors represent become identical: the position, rotation, velocities and accelerations are equal. However, the sum of all acting forces (plus the resulting force taken with negative sign) and the sum of all torques (plus the resulting torque taken with negative sign) are zeroes. This equation is implied by the fact that force and torque are **`flow`** variables of the connector.

`class Inertial` An MBS model should contain one (and only one) object (**`class`** `Inertial`) representing the global coordinate system and force of gravity. It has a connector `b`. All other components in the model should be directly or indirectly connected to the connector `b` (`MbsCutB`) of the `Inertial` object.

`class Bar` This class has connectors `a` (of connector class `MbsCutA`) and `b` (`MbsCutB`), as well as parameter **`Real`** `r[3]`. This parameter describes a fixed translation between the frames A and B in the coordinates of A. An object of this class represents a massless bar between these two points.

`class Revolute (RevoluteS)` This class has connectors `a` (`MbsCutA`) and `b` (`MbsCutB`), as well as parameter **`Real`** `n[3]`. This vector defines the direction of the axis of rotation when the B frame rotates around the A frame. There is a variable `q` which contains the current angle of the joint. There is also a **`connector`** `pDrive` which allows connecting the driving force (i.e. motors) to state variables for objects of class `RevoluteS`. The class `Revolute` is used when motion is predefined, or within kinematic loops.

`class Prismatic (PrismaticS)` This class has connectors `a` (`MbsCutA`) and `b` (`MbsCutB`), as well as parameter **`Real`** `n[3]`. This vector defines the direction of translation when the B frame moves relatively to the A frame. There is a variable `q` which contains the current relative distance of the joint. There is also a **`connector`** `pTrans` which allows connecting the driving force (i.e. motors) to state variables for objects of class `PrismaticS`. The class `Prismatic` is used when motion is predefined, or within kinematic loops.

3.5.2 Example: Kinematic Outline of Double Pendulum

In this section an idealized model of a double pendulum is considered. This pendulum consists of two boxes linked by a hinge. The upper box is lined by a hinge to a non-movable platform, i.e. an inertial system. Both the boxes can move in the

[3]This rule, however can be violated when closed kinematic loops and external forces are used.

XY plane. The corresponding kinematic outline consists of two revolute joints, one bar and one inertial system. The structure is displayed in Figure 15.

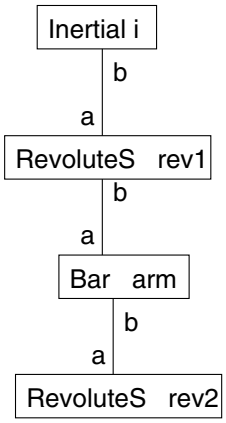


Figure 15: Connection diagram of a kinematic skeleton of a double pendulum without masses.

3.5.3 Adding masses.

The motion of dynamic system cannot be defined just by its kinematic outline, since an outline does not contain mass objects yet. The class `BodyBase` represents mechanical objects with mass. This class has a connector `a` (`MbsCutA`) as well as the following parameters supplied by the user:

- **`Real`** `m` - mass of the body.
- **`Real`** `rCM[3]` - position of center of mass with respect to frame `a`.
- **`Real`** `I[3,3]` - inertia tensor with respect to the position of the center of mass. This 3 by 3 tensor is defined by six numbers since it is symmetric:

$$\begin{pmatrix} I_{11} & I_{21} & I_{31} \\ I_{21} & I_{22} & I_{32} \\ I_{31} & I_{32} & I_{33} \end{pmatrix}$$

3.5.4 Adding Masses to the Double Pendulum Example.

In the double pendulum example two bodies can be added to the kinematic outline.

Since the description of class `BodyBase` contains the distance between its frame A and its center of mass, such objects can be directly connected to connector `b` of revolute joints `rev1` and `rev2`. The kinematic skeleton with masses is displayed in Figure 16.

The instances of Modelica classes (such as `P1`, `P2`, `rev1` etc.) have attributes that can be modified. For instance, `Body` has attributes that define mass, inertia

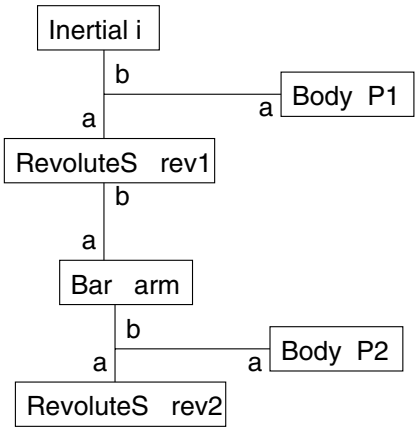


Figure 16: Connection diagram of a kinematic skeleton of a double pendulum with masses.

tensor and the location of its center of mass relative to the local coordinate system (`rCM`). Instances of `RevoluteS` (revolute joint) have an attribute `n` that define the direction of axis of rotation. Instances of `PrismaticS` (prismatic joint) have an attribute that specify the direction of the allowed translation. For a `Bar` the coordinates of its end are specified by the vector `r`.

Several important parameters should be defined for a double pendulum: L_1 and L_2 are the lengths of the boxes. Their masses are m_1 and m_2. The center of mass of the first box is located at the distance $L_1/2$ from the first rotation point. Rotation axes of the revolute joints are directed along the Z axis. Therefore these are described as the vector $\{0,0,1\}$.

The textual representation of the Modelica model is as follows:

```
class Pendulum
  parameter Real L1;
  parameter Real L2;
  Inertial I;
  Body P1(rCM={L1/2,0,0});
  Body P2(rCM={L2/2,0,0});
  RevoluteS rev1(n={0, 0, 1});
  RevoluteS rev2(n={0, 0, 1});
 Bar arm(r={L1, 0, 0});
equation
 connect(I.b, rev1.a);
 connect(rev1.b, P2.a);
 connect(rev1.b, arm.a);
 connect(arm.b, rev2.a);
 connect(rev2.b, P1.a);
end Pendulum;
```

The corresponding mechanical structure is shown in Figure 17.

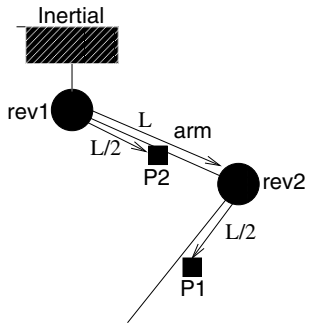


Figure 17: Mechanical structure of the pendulum.

3.5.5 Adding Geometrical Shapes

There are two ways to create 3D visualizations of mechanical models for systems constructed from the MBS library. The information about object position and rotation can be saved in a file and read back later for visualization. Alternatively, this data can be sent to external functions for visualization.

Objects of MBS classes can be connected with special classes intended for saving position and rotation of objects during simulation. In particular, in the Dymola tool[15], class `VisualMbsObject` can be used for this purpose[4]. The objects of this class require the following parameters and variables for visualization:

- Connector `a` (`MbsCutA`) which is normally connected to connector `a` of the corresponding body.
- **`Real`** `nx[3], ny[3]` specify the orientation of the visualized shape (direction of its axes `X` and `Y`) with respect to the frame of the connector `a`.
- The vector **`Real`** `r0[3]` specifies the point of insertion of the shape with respect to the rotated frame of the connector `a`.
- The array **`Real`** `Material[4]` specifies the color of the shape.

Other variables specify a predefined shape[5] (and its size) or the custom shape. The detailed description of geometry and other graphical properties of custom shape can be stored outside Modelica code in some standard format, e.g. DXF ([15]), STL or VRML (see Section 8), or can be stored inside the Modelica code as annotations, as proposed in [21].

[4]This class does not belong to the MBS library. Interpretation of the data as graphical information is not part of Modelica semantics and is specific for the particular tools that implement visualization, e.g. DymoView[15], or MVIS (Section 8).

[5]The standard predefined shapes are box, cylinder, sphere, pipe, cone, beam, wirebox and gearwheel.

3.5.6 Adding Shapes for Double Pendulum

In order to use shapes for pendulum visualization we can utilize class `BodyV` which includes classes `Body` (Section 3.5.3) and `VisualMbsObject` (Section 3.5.5). The parts of the pendulum can be treated as a box of size $L_1 \times K_1 \times K_1$ and $L_2 \times K_2 \times K_2$. Such points have the default point of insertion in the center of the left face (Figure 18).

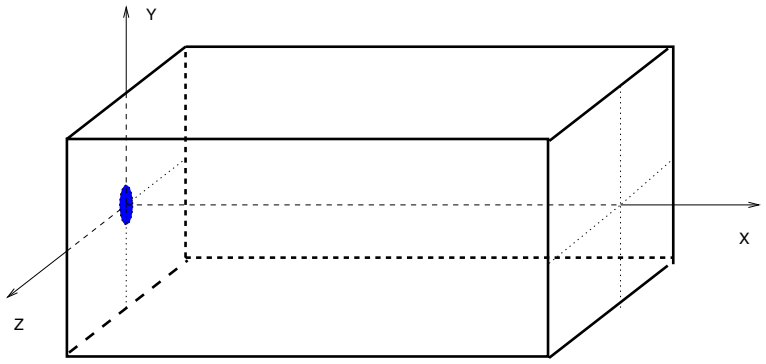


Figure 18: The default position and rotation of a "`box`" – a predefined shape of the MBS library.

The textual representation of Modelica model is following:

```
class Pendulum
 parameter Real L1;
 parameter Real L2;
 Inertial I;
 BodyV P1(mass=M1,r={L1/2,0,0}
   Shape="box", Size={L1,K1,K1}, r0={0,0,0},
    Material={1,0,0,0.5});
 BodyV P2(mass=M2,r={L2/2,0,0}
   Shape="box", Size={L2,K2,K2}, r0={0,0,0},
    Material={0,1,0,0.5});
 RevoluteS rev1(n={0, 0, 1});
 RevoluteS rev2(n={0, 0, 1});
 Bar arm(r={L1, 0, 0});
equation
 connect(I.b,    rev1.a);
 connect(rev1.b, P1.a);
 connect(rev1.b, arm.a);
 connect(arm.b,  rev2.a);
 connect(rev2.b, P2.a);
end Pendulum;
```

3.5.7 Interface to Non-Mechanical Parts of the Model

When a multidomain model in Modelica is designed, mechanical part of the model receives energy from some other parts of the model. The mechanical part should usually be connected with the part describing the drive train, which is in turn connected to the part describing the energy source, e.g. electrical circuit. There are standard Modelica libraries for all these components.

However, sometimes the attention is focused on an *isolated* mechanical part of the model. In this case the forces and torques acting on the model can be specified directly.

There exist three major ways to set up an interface between the mechanical part of Modelica model and other parts. These ways correspond to three connector types: `MbsCut`, `DriveCut` and `TransCut`.

Using `MbsCut`. Any object with connector of type `MbsCut` can be connected to model of external force or torque. This external force (or torque) will be applied to the corresponding point on the relevant body. This can be demonstrated as follows:

```
   ...
   ExtForce E;
   // or ExtTorque E;

   MbsCutB  MB;
   Bar      B;
equations
   connect(B.b,E.b);
   // or    connect(MB,E.b);

   E.fb={fx, fy, fz};
   // Force given  in the local coordinate frame

   // or

   E.fb=E.Sb'*{fx, fy, fz};
   // Force given  in the global coordinate frame
```

Using `DriveCut`. A torque can be applied to a revolute joint with a state variable. First, a motor (torque source) model is declared, and then it is used. Currently the MBS library uses old nonstandard syntax for specifying variables included in the connector. Function `foo` is often defined through the control equation `t = k*(qref-q) - d*qd`, where k and d are large constants obtained from experiments, and $qref$ is the reference angle which can vary. The motor steers the joint so that q tends to be very close to $qref$. The intermediate connector `D` is sometimes used when `R` is encapsulated in some other class (e.g. this occurs in the SolidWorks-to-Modelica translator output).

```
model Motor;
  Cut nDrive(accross={q;qd;qdd}, through=-t);
equations
  t=foo(q, qd, qdd);
   // for example    t = k*(qref-q) - d*qd;
end Motor;
...
  RevoluteS R;
  Motor M;
  DriveCutPos D;
equations
  conect(M.nDrive,D); connect (D,R.pDrive);
  // or connect(M.nDrive,R.pDrive);
```

Using `TransCut`. A force can be applied to a prismatic joint with a state variable. First, a motor model producing linear force is declared, and then it is used. Function `foo` is often defined through the control equation `f = k*(qref-q) - d*qd`, similar to the case above. Here $qref$ is the reference relative position.

```
model MotorP;
  Cut nTrans(accross={q;qd;qdd}, through=-t);
equations
  f=foo(q, qd, qdd)
end Motor;
...
  PrismaticS P;
  MotorP M;
  TransCutPos T;
equations
  conect(M.nTrans,T);connect (T,P.pTrans);
  // or connect(M.nTrans,P.pTrans);
```

3.6 Advantages of Using MBS for Dynamic Analysis

There are two major advantages in using the MBS and Modelica library for Dynamic Analysis.

- The simulation is not *interpreted*, but *compiled*.
- Multidomain modeling is made possible.

3.6.1 Interpretation and Compilation in Mechanical Simulation

There exist two ways to specify a model for mechanical simulation software. The software can *interpret* the model or *compile* it to an executable code.

In mechanical simulation libraries a particular mechanical model is usually described as a collection of program objects. These objects can be created by calling library functions, or they can be created by instantiating some classes from a class library. Alternatively the mechanical model is described in a proprietary format (or specified via graphical user interface), as this is done in Working Model

3D and ADAMS. In all these cases *interpretation* of the model information takes place.

In contrast, when using Modelica/MBS the mechanical model is first compiled to a set of equations, which are optimized. Then C code only including necessary arithmetic operations for the specific mechanical model is generated, optimized by C compiler, and finally compiled into a specific executable application. This results in much faster applications.

3.6.2 Multidomain Simulation

Modeling mechanical systems is one of the major Modelica applications. In Modelica this can be done by employing the Multi-Body System library (MBS). Classes from this library can be instantiated. A proper collection of instances as described in section 3.5 constitutes a system of differential and algebraic equations. The solution of this system yields the dynamic behavior of the corresponding mechanical system.

Another major Modelica application area is modeling of multi-domain systems. This means that for instance electric, hydraulic and control models can be incorporated into the same mathematical model as the model of a mechanical system. This is obtained by instantiation of classes from the corresponding libraries.

3.7 Difficulties of using the MBS library

There are two types of difficulties with the usability of the MBS library.

- There are connection rules for the `MbsCut` connectors, rules for using classes with or without state variables, and rules for constructs with kinematic loops. It is hard to see a mapping between these rules and physical objects. These rules should be used very carefully; if they are violated, there is a little chance that appropriate diagnostic messages are generated, since useful information is lost when class structures are converted into flat set of equations.

  However, currently, in 2000, a new MBS library for Modelica is being developed, which solve many technical problems[18]. In particular, the rules for kinematic loops are simplified via state deselection during simulation. New algorithms that perform state deselection are based on [34].

- Descriptions of mechanical models contain many numerical values. Typically for a rigid body dynamics model (when the surface geometry is ignored) at least 20 numerical parameters are necessary. These values cannot be derived by just looking at the bodies and reasoning about them. Instead, they should be obtained from some reliable source.

4 CAD Tools

In order to simplify the design of mechanical Modelica models, CAD tools can be utilized.

In this section we discuss the following questions:

- Which properties of CAD tools are important and necessary for dynamic analysis?
- Which properties are necessary for conversion of CAD models to Modelica code based on the MBS library?
- Comparison of several tools (SolidWorks, Pro/ENGINEER, Mechanical Desktop, WorkingModel 3D, 3D Studio Max) with respect to possibilities of conversion into Modelica/MBS.
- How constraints (mates in SolidWorks terminology and Joints in Modelica/MBS terminology) are specified in SolidWorks.

4.1 CAD Tools and Dynamic Analysis

There exist hundreds or even thousands of different software tools that assist in general design – so called computer-aided design tools. Only a small fraction of those can be used for dynamic analysis of mechanical systems. The minimal requirements for dynamic analysis are:

- The tool should be able to construct and extract information about 3D mechanical components (bodies) which can serve as point masses (abstraction of rigid bodies with all mass concentrated at their center of mass). There should be some way to specify *initial position and rotation* of three-dimensional shapes. The shapes can be either primitives (box, sphere, cylinder, cone), or complex (combination of primitive shapes), or feature-based (constructed by applying a sequence of operations in 3D), or arbitrary polytope (constructed by specifying a set of triangles in 3D).
- The tool must be able to describe mechanical *constraints* between the bodies, such as joints. It should be possible to specify reference points (alternatively axes, edges, faces, planes) on two bodies and choose which constraints applies to the bodies.
- The tool should be able to compute (or have all necessary information for computation of) the *mass*, the *center of mass*, and (preferably) the *matrix of inertia* for each component. In many cases CAD tools are able to automatically compute volumes of complex shapes and their centers of mass. The densities of bodies are considered as homogeneous. Therefore the mass can be found from the volume and density. In the general case the computation of

the volume and the center of mass can be a complicated and computationally intensive task. It should be noted that many CAD tools (e.g. VRML construction tools) operate with surface presentation only, and therefore volume and mass cannot be found.

These requirements are enough for the automatic construction of models which have no external interactions (forces, torques, collisions) applied. Typically some information should be specified in addition to the CAD model in order to simulate some mechanisms. For instance, definition of an external force should be attached to some point of certain body. This force can be either specified by the user within the CAD tool, or it can be described later in a simulation language, or by using a simulation environment interface. In the first case these specifications are stored as additional, user-defined attributes (user-defined properties) of mechanical parts. In the later cases all the bodies should have some distinctive names and the name of the body should be used as a reference.

4.2 Comparison of Various CAD tools

In this section we consider several CAD tools. We investigate whether and how it is possible to extract information necessary for dynamic simulation from these tools.

4.2.1 SolidWorks

The SolidWorks[52] environment supports two kinds of documents for mechanical design: the *part* model and the *assembly* model. An assembly might consist of parts and other assemblies. Each part corresponds to mechanical body. The parts are constructed by applying a sequence of CSG (constructive solid geometry) operations, called *features*. Each such operation converts a solid body model to another solid body model. The operations preserve or update some important features of solid bodies, for instance mass, position of the center of mass and value of the tensor of inertia.

SolidWorks does not support the description of movement constraints between the bodies in the form of joints. Instead a set of mates between points, axes, edges, faces or planes of two bodies is configured by the user. The translation between such representations is a difficult problem. Our variant of this translation is discussed in Section 5.4. SolidWorks automatically computes the volume of a part, the center of the volume and the tensor of inertia, assuming that the part has uniform density. The value of the density can be specified as a property of the part when it is created.

SolidWorks uses an application programming interface based on COM (Microsoft Component Object Model) to give access for browsing and modification of virtually all properties of parts and assemblies. This way all information about the kinematic features of the model can be extracted.

4.2.2 Working Model 3D

In the Working Model 3D ([29], see also Section 2.1.2) 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. Mechanical joints that can be specified in Working Model 3D directly correspond to joints in the Modelica MBS library. The mass, the center of mass and the inertia tensor are automatically computed from user-specified density. However, all these can be overridden by the user. The major drawback of Working Mode 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.

4.2.3 3D Studio Max

This tool ([5], see also 2.1.3) is able to specify object geometry in a very detailed way. However, there is a gap between geometry and physical properties. 3D Studio Max has several degrees of approximation between its objects and the corresponding physical objects. This quite useful approximation feature does not appear in other modeling tools.

The objects can initially be constructed as solid bodies or as surfaces. When such objects are used for dynamic simulation either a bounding volume (box or sphere), or a mesh subdivision of the object can be used as the geometry model for dynamic simulations. This choice is up to the user. Obviously, computations using bounding volumes are much faster and usually more inaccurate than computations using a mesh.

3D Studio Max is primarily oriented towards description of graphical and geometrical properties of surfaces. Therefore the computation of certain physical properties is not always adequate: objects might contain just a surface (e.g. a square) which has no solid body properties such as mass and volume. To solve this problem, the tool computes mass and volume using a certain degree of accuracy, in particular using bounding box around the surface, bounding sphere or (much more exact for complex shapes) using mesh subdivision.

Movement constraints in 3D Studio Max are specified as *locks*[6] (these forbid translation and rotation of child objects with respect to parent object in certain directions).

The mass and the position of the center of mass are either computed using the rules mentioned above, or can be overridden by the user. The inertia tensor is not computed nor used at all.

[6]This is 3D Studio Max terminology.

3D Studio Max uses MaxScript (a proprietary interpreted command and programming language) or a C++ API for manipulation and extraction of data from 3D scenes. Geometry can be extracted in VRML, STL, DXF and some other formats. Specification of locks and physical (dynamic properties) overridden by the user cannot be extracted by MaxScript, but apparently can be accessed via the C++ API. A tool for extraction of this information from 3D Studio Max to Modelica is being designed by Dynabits[13].

4.2.4 Mechanical Desktop

Mechanical Desktop (version 4)[6] is an environment which integrates the AutoCAD 2000 (a tool for modeling of parts and surfaces), ACIS 5.1 kernel for modeling complex geometry as well as features of Genius Desktop 3 and AutoCAD Mechanical 2000. Details about these products and various formats mentioned in this section can be found in [6].

It is one of the most popular tools for mechanical design. This tool is one of about 200 CAD tools related to AutoCad, which use the ACIS engine for internal representation and processing of CAD information.

All tools using the ACIS engine can exchange geometry data using the ACIS format (ASCII `.sat` files and binary `.sab` files). However applications use specialized formats for storage of all other information but geometry. In particular Mechanical Desktop uses binary DWG format.

Products of the AutoCad family have specialized support for many areas of computer-aided design. Special applications are targeted for building industry, geographical information systems and mechanical design. Also there is a whole series of applications for static and dynamic visualization of AutoCad models, including 3DStudio Viz. It uses Mechanical Desktop files for automatic assembly animation. Each modification of a model in 3DStudio Viz is reflected in Mechanical Desktop and vice-versa.

In order to construct an assembly, all the parts should be first referenced or localized into the assembly drawing. Mechanical models with motion constraints (assembly constraints in the AutoCad terminology) are constructed in Mechanical desktop by means of menu commands (`Mate`, `Insert`, `Flush`, `Angle`). These menu choices invoke command line interface commands with corresponding prompts:

- `AMFLUSH` specifies two parallel planes and a fixed distance between the planes (the distance is 0 by default).
- `AMANGLE` specifies an angle between two planes, two vectors (or axes) or between a plane and a vector (or an axis). The angle is 0 by default.
- `AMMATE` specifies that two planes, axes, points or non-planar faces (sphere, cone, cylinder, or torus) are coincident.

- `AMINSERT` specifies that two faces (e.g. cylindrical surfaces or edges) are coplanar and share common axis.

In order to extract information about relations between bodies it is possible to use ObjectARX (Object AutoCAD Runtime Extension) which provides a mechanism to manipulate the Mechanical Desktop programmatically from within or outside of the Mechanical Desktop. This Automation (Application Program Interface) gives C++ programs (clients) access to the internal representation of assembly information. Programs written using this interface are installed as plug-ins in the Mechanical Desktop environment.

In particular, C++ clients can access `McadConstraintDescriptor` object which contains information about two geometric objects operated on by this constraint, the type of constraint (one of `mcCompMate`, `mcCompFlush`, `mcCompInsert`, `mcCompAngle`) as well as a value which for example could be the offset distance or the angle.

The facilities for constraint definition in Mechanical Desktop are similar to those in SolidWorks (see Section 5)). As part of our future work we plan a converter from Mechanical Desktop assemblies to Modelica.

4.2.5 Pro/ENGINEER Tool Family

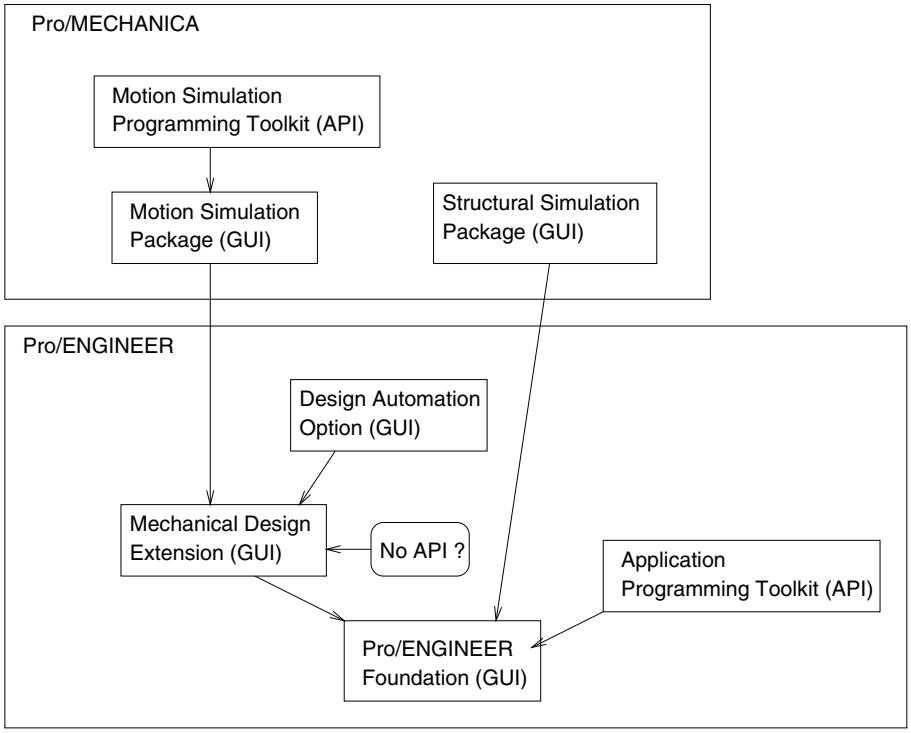


Figure 19: Structure of layers of Pro/ENGINEER and related tools.

Pro/ENGINEER[39] is one of the most widely spread tool families for mechanical design. It is targeted primarily for industrial applications and design optimization of products. Several different environments for different purposes are combined together, and a common CAD data representation model is used for information exchange. In order to create a mechanical model consisting of several bodies, a static model of collection of parts is created initially in the Pro/ENGINEER *Foundation Tool* (see Figure 19). When an assembly is created from parts, locations of future joints should be correctly aligned.

Pro/ENGINEER has several extension packages, in particular *Mechanical Design Extension*. This extension enables designers to specify motion of assembled Pro/ENGINEER parts. When the parts are assembled, joints of different types connect them. These joints are Pin, Slider, Cylinder, Planar and Ball. Every such joint define certain limitations of part movement relatively to each other. Rotational and translational degrees of freedom are reduced, and the motion is constrained.

Mechanical Design Extension has other important facilities for assemblies: every part of the assembly can be moved (dragged by mouse); predefined (prescribed) movements can drive the parts. Some performance features of the mechanisms can be assessed at this stage, e.g. clashes can be detected, and model extents in the coordinate space can be evaluated.

Pro/ENGINEER has a *Design Automation option*. This makes it possible to create animation sequences of moving mechanisms. The designer specifies positions of parts in assemblies for some key frames, and smooth movement of these parts is automatically generated.

Pro/MECHANICA is another product family which is built on top of core Pro/ENGINEER, and used for model simulation. This package contains several simulation tools, in particular the *Motion Simulation Package* for kinematic analysis (using equations of motion of rigid bodies) and *Structure Simulation* for structure deformation and stress studies (using finite element analysis). Kinematic analysis uses description of parts defined in Pro/ENGINEER Foundation Tool, and joint definitions defined in Mechanical Design Extension. Since Mechanical Design Extension is useful in the context of Pro/MECHANICA only it can be also considered as part of Pro/MECHANICA.

Kinematic simulation is used as a foundation for the more advanced features of Pro/MECHANICA: sensitivity analysis and automated design optimization. When multiple design parameters are specified and a design goal is expressed in measurable physical quantities, the tool can perform automatic design optimization. These simulation capabilities can be augmented even more by the *Motion Simulation Programming Toolkit*. This toolkit enables in particular extracting the equations of motion (in an encapsulated form) so that the simulation can be driven by an external program. This way components from other application domains (electric, control, hydraulic, etc.) can be attached to mechanical simulation in order to build a single application.

Access to information describing model geometry and all model features created by the Pro/ENGINEER Foundation Tool can be done by exporting data in the

MNF (Mechanica Neutral File) format or via the *Application Programming Toolkit*. The features specified by the *Mechanical Design Extension* (i.e. joints of different kinds) are stored in MNF and not accessible via the toolkit. This API offers the possibility of adding user-defined objects to the model. This way it would be possible to add special joints. These new joints would not be compatible with the joints defined in Mechanical Design Extension.

Regarding translation of mechanical models with joints from Pro/ENGINEER to Modelica we can conclude in its current version (Pro/ENGINEER 2000) neither *Application Programming Toolkit* nor MNF can be used for this purpose. New user-defined objects can be created, however. These objects can be defined in the same way as joints. The model together with such objects (Modelica's joints) can be translated to Modelica. The disadvantage of this approach is that these joints cannot be used in Pro/MECHANICA.

5 Mechanical Model Design in SolidWorks and Model Translation

5.1 Design of SolidWorks Parts and Assemblies

The system currently used in our project is SolidWorks[52].

SolidWorks uses the concepts of parts and assemblies. Each solid component (a rigid body) is modeled as a separate *part* document.

In the *assembly* document these parts are put together to form a complete model. Each part model can also occur more than once in the assembly.

The *assembly* document defines the mobility between the parts of an assembly. Between two parts, several so called *mates* are connected, each adding some constraint to the mobility between the parts.

A part consist of entities, such as planes, faces, edges, axes and points. A mate connects two entities from different parts. There exist several mate types. The most typical are *coincident* (all the points of one entity are inside another entity) or *parallel* (it keeps entities parallel to each other).

Two parts can be connected by one, two or more mates. Some combinations of mates are valid, some are not. Invalid combinations of mates (over-constrained systems) are automatically rejected by SolidWorks.

5.2 Mating Example

The example in Figure 20(a) describes a fragment of the pendulum model. The part `P1` has a front face (`f1`) and an upper edge (`e1`). The part `P2` has a front face (`f2`) and a bottom edge (`e2`). There is a mate that specifies that the planes of `f1` and `f2` are coincident. Another mate specifies that the edges `e1` and `e2` are coincident, i.e. they belong to the same line in the coordinate space. The SolidWorks system analyzes the mates and adjusts the positions of the parts (Figure 20(b)). The system automatically rejects invalid mate combinations. In this case our translator [30]

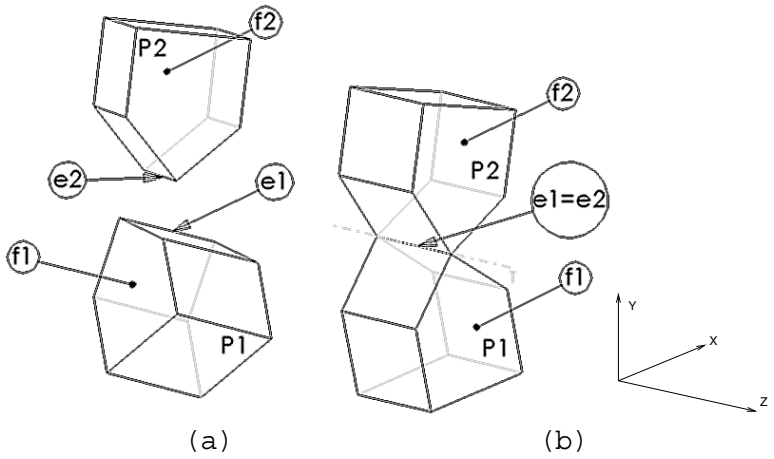


Figure 20: Parts and their mates specification before (a) and after (b) adjustment according to the mates.

finds that there is a joint with one rotational degree of freedom between the parts `P1` and `P2`, and calculates the position and orientation of the rotation axis. This pair of mates corresponds to an instance of class `RevoluteS` from the Modelica MBS library with an attached `Body` instance.

5.3 Classification of mates

There are seven different mate types:

Coincident: One entity belongs to another entity

Concentric: Cylinders and cones have a center line. This line can be shared between several round entities, or it can be coincident with some other line in the model.

Perpendicular: Planes or lines keep a 90° degree angle to each other.

Parallel: Entities are parallel to each other

Tangent: An entity touches a cylindrical entity

Distance: Two entities keep a fixed distance.

Angle: Two entities keep a fixed angle

When mates are applied to entities, the number of degrees of freedom (3 translational and 3 rotational degrees) are reduced. Table 2 shows the degrees of freedom between two parts with plane and line entities connected by one of the four most used mates. The complete table can be found in [30].

Entities	Mate type	Translational DOF	Rotational DOF
Plane/Plane	coincident	2	1
	distance	2	1
	parallel	3	1
	perpendicular	3	2
Line/Plane	coincident	2	2
	distance	2	2
	parallel	3	2
	perpendicular	3	1
Line/Line	coincident	1	1
	distance	–	–
	parallel	3	1
	perpendicular	3	2

Table 2: Degrees of freedom which are left between parts connected by mates in their entities.

5.4 Translation of mates into joints

In the example above the simplest case of translation of SolidWorks mates into MBS joint is demonstrated. Two lines (edges) are coincident and two planes (faces) are coincident. Parts free of constrains can rotate around and translate along all the axis. This can be denoted as movement freedom symbols (TX TY TZ RX RY RZ). The coincident lines (see Figure 20(b)) are directed along the Z axis. The coincident planes are located in a plane which is parallel to the XY plane.

The mates reduce degrees of freedom in the following way:

- Coincidence of two lines forbids (i.e. reduces freedom of) rotation around the X and Y axes and translation along X and Y axes. The symbols TX, TY, RX, RY are removed.
- Coincidence of two planes further forbids (i.e. reduces the freedom of) translation along the Z axis, and does not allow the parts to rotate around the X and Y axes.

Therefore the symbols TZ, RX, RY should be removed. The only symbol left is TZ. Therefore a revolute joint (one rotational freedom degree) should be inserted between the parts, and its axis is parallel to the axis Z. The point of the joint is chosen as the intersection point between the lines and the planes.

Each SolidWorks assembly consists of a set of parts, and stores a set of mates. All these are validated and translated to a set of Modelica MBS class instances and appropriate *connections* between them. The mass, position of center of mass and inertia tensor are extracted from the corresponding part documents by SolidWorks.

In the general case when mates are translated to joints a more complex method is used (see Figure 21). We consider two parts and all the mates between them

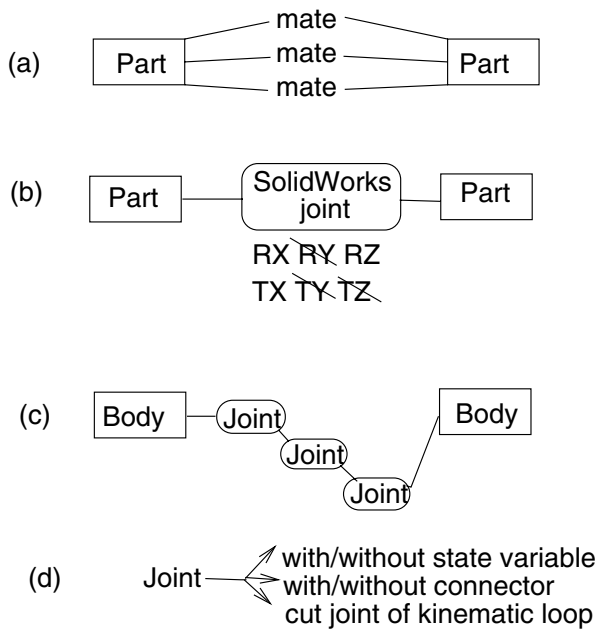


Figure 21: Translation of SolidWorks mates to Modelica joints.

(Figure 21(a)). Each mate is analyzed and translated into a `CDOF` (Class Degree of Freedom) object. This object stores the directions of the orthogonal vectors corresponding to TX, TY, TZ, RX, RY, and RZ. This object also contains a boolean array (with 6 elements) defining which of the vectors correspond to freedom of movement.

The resulting SolidWorks joint (Figure 21(b)) between two parts is derived from combination of all `CDOF` objects made for the corresponding mates. The case when two mates are applied at the same point of rotation is relatively simple. Axes of translation and rotation existing in both terms are left in the result. Otherwise the algorithm checks whether lines and planes considered are parallel or not, and in some cases a new rotation axis is created. The details of the algorithm are given in [30]. Each SolidWorks joint corresponds to (from one to five) Modelica joints (Figure 21(c)) directly attached to each other in a chain.

When the joints between all the parts are established, a multibody tree structure is built. The `SubInertial` object serves as the reference point and the root of the tree. The parts that have fixed position in the SolidWorks coordinate frame are directly attached to the `SubInertial` via `Bar` objects. For each pair of parts with joints a connection in the form of a `Bar` and one or several `Joint` instances is created. The `Bar` serves as a link from the frame of the `a` connector of a `Body` (its main reference point) to the point where the joint is attached to the body. Finally, a kinematic outline is obtained. This kinematic outline is a graph with vertices corresponding to parts (`Body` objects in MBS), `Bars` and joints. The edges of the graph correspond to the connectors.

5.4.1 Multibody Systems with a Kinematic Loop

A kinematic outline, in particular one created from SolidWorks assembly, may contain a cycle. Such cycles are called kinematic loops. In this case some special MBS library classes are used instead of usual `RevoluteS` and `PrismaticS` joints. These classes are called cut joints (`SphericalCut`, `RevoluteCut2D`, `RevoluteCut3D`). A cut joint replaces a usual joint in the outline so that the remaining joints build a tree. All other joints are divided to joints with state variables (`RevoluteS`,`PrismaticS`) and a group of joints without state variables (`Revolute`, `Translate`). The number of joints without state variables in the loop must be identical to the number of constraints in the cut joint. `SphericalCut` has 3 constraints, `RevoluteCut2D` has 2 constraints, `RevoluteCut3D` has 5 constraints.

For instance, the following way of programming a kinematic loop: (`SubInertial - RevoluteS - SphericalCut - Revolute - Revolute - Prismatic - SubInertial`) is legal. Another legal example is (`SubInertial - PrismaticS - PrismaticS - PrismaticS - RevoluteCut3D - Revolute - Revolute - Revolute - Revolute - Revolute - SubInertial`)

In the case when all the movements take place in the same plane, a planar 2D kinematic loop occurs, and `RevoluteCut3D` should be replaced by `RevoluteCut2D`. A legal example is (`SubInertial - PrismaticS - PrismaticS - RevoluteCut2D - Revolute - Revolute - SubInertial`). Another legal example is (`SubInertial - Revolute - RevoluteS - RevoluteCut2D - Revolute - SubInertial`).

The choice of type of joint between a joint with state variables, a joint without state variables, or a cut joint is performed by the user. The correctness of the choice is verified by the SolidWorks-to-Modelica translator.

In future an automatic algorithm to perform a choice of joint types can be implemented in the translator. Furthermore, a new MBS library is being developed, and the rules for kinematic loops are simplified since state deselection can be done during simulation[18]. New algorithms that perform state deselection are based on [34].

5.5 User Interface for Configuration of Joints

There are several ways how a mechanical joint found from a combination of SolidWorks mates can be translated into one or several instances of MBS joint classes. The attributes (Figure 21(d)) of the joint are available via a special menu item *Edit Joint Attributes* in the SolidWorks-Modelica plug-in. Every SolidWorks joint contains between zero and five degrees of freedom. A drive connector can be propagated for each separate degree of freedom. A `DriveConnectorPos` connector is available for rotating degrees and a source of external driving torque (e.g. a motor) can be attached to such a connector. A `TransConnectorPos` connec-

tor is available for translating degrees of freedom, and a linear actuator (external driving force) can be attached to it. The motors and actuators should currently be attached manually in the external code; in the future it will be possible to specify them directly via dialog windows of the plug-in.

The user can specify whether a Modelica joint has state variables or not. A joint without state variables should be used if it is included in a kinematic loop. The user can specify whether the joint is a cut joint or not.

5.6 Mechanism Example – Crank model

To illustrate the use of the SolidWorks–to–Modelica converter we consider an example of a crank mechanism (Figure 25). It includes several important features: joints of different kinds, kinematic loops, and external mechanical connectors.

A crank model has been designed in SolidWorks. It contains the fixed basement and two empty cylinders rigidly attached to the basement. A piston is moving in each cylinder. Each piston is connected to the crank by the rod.

When a force (e.g. an explosion) is applied to the pistons in the vertical direction, they move. This causes the crank to rotate. In order to apply a force to the pistons, mechanical connectors are defined in the dialog window of the translator. These connectors are attached to the pistons. They are called `pist_1_explo` and `pist_2_explo` in the generated Modelica code.

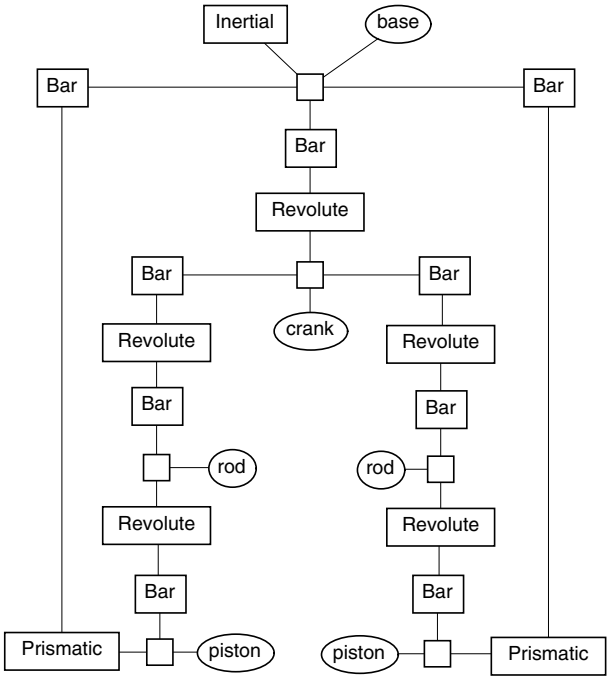


Figure 22: Kinematic outline of the crank mechanism. The type of the revolute joints is not defined yet.

When the model is translated from SolidWorks to Modelica (see Figure 22), all the revolute joints have state variables by default.

The model of the crank contains two kinematic loops. Therefore two cut joints should be inserted into the outline. The converter issues an error message that notifies that correct Modelica model cannot be generated since there are kinematic loops and no cut joints yet. The user should choose which joints should be replaced by cut joints. According to the rules of programming with the MBS library, the loops should be cut so that the kinematic outline is transformed into a tree. There are several ways to do that. The `Revolute` joint can be replaced by `RevoluteCut2D` joint in four different ways. In addition, one `Revolute` joint should have state variables, i.e. be an instance of `RevoluteS`. This can be done in three different ways. Since there is just one degree of freedom in this system, only one joint with state variables should be used. A non-convergence error during solution of the equations may occur if the value of a state variable (i.e. rotation angle for revolute joints and displacement value for prismatic joints) is close to the limit of possible values due to geometric constraints. Therefore the prismatic joints should not be used here as joints with state variables. When all these observation are taken into account, the joints between the crank and the rods (`J22` and `J23`, see Figure 23) are replaced by the `RevoluteCut2D` joints. The joint between the base and the crank (`J21`) becomes a revolute joint with state variables `RevoluteS`[7]. As a result, in each kinematic loop there are two joints without state variables: `J26` and `J27` in one loop, `J24` and `J25` in another loop.

The model `r1` generated from SolidWorks should be used together with a wrapper model which defines external forces acting on the pistons. This wrapper model is defined as follows:

```
model world
    Inertial I;
    ExtForce EF1;
    ExtForce EF2;
    Real  F1   "force applied to piston 1";
    Real  F2   "force applied to piston 2";

    parameter Real F1per = 1;
    parameter Real F2per = 1;
    parameter Real F1amp = 1;
    parameter Real F2amp = 1;
    parameter Real F1offs = 0;
    parameter Real F2offs = 3.14;
    r1 r;
 equation
    connect(I.b, r.a);
    F1= F1amp*sin( 2*3.14* Time/ F1per + F1offs );
    F2= F2amp*sin( 2*3.14* Time/ F2per + F2offs );
    connect(r.pist_1_explo, EF1.b);
```

[7]From mechanical engineering point of view the axes of the joints `J21`, `J22` and `J23` should not belong to the same plane, as it is in our model, since a ”dead” position may occur when no forces from the pistons can shift the crank anymore.

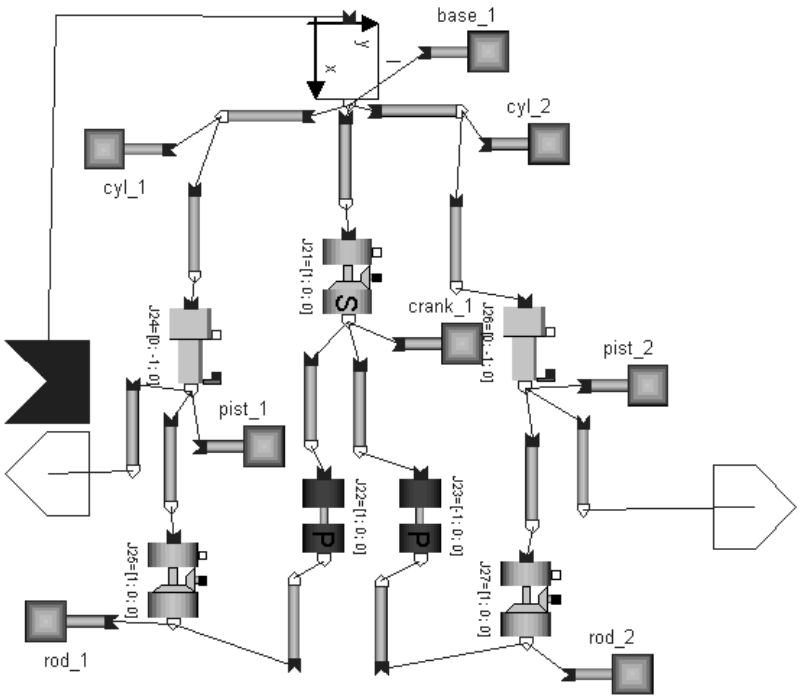


Figure 23: Modelica diagram for the crank mechanism (model `r1`).

```
    EF1.fb=0,F1,0;
    connect(r.pist_2_explo, EF2.b);
    EF2.fb=0,F2,0;
end world;
```

In this case the force is defined as a simple periodic function. In the engineering application any other function depending on any parameters (e.g. position and velocity of the piston) can be used here.

Engineers can use a 2D plot describing the angle of rotation `r.J21.q` in order to draw conclusions about the result of applied force function. In this case the crank rotates in the opposite direction for some time ($3 < time < 5$), but rest of the time it rotates correctly. The 3D visualization is automatically created for this model by the MVIS tool. The result can be seen on Figure 25.

5.7 Mechanism Example – a Swing Model

A model of a man on a swing (Figure 29) illustrates a mechanism with kinematic loop and a motor force applied to a revolute joint. The swing contains a fixed basement with two rotating rods that are attached to a platform (a ”horse”). A ”man” is sitting on the platform. It is attached by a revolute joint to the platform; this joint performs oscillating movements. The model illustrates a well known effect: when the force applied to the swing is oscillating with the same frequency

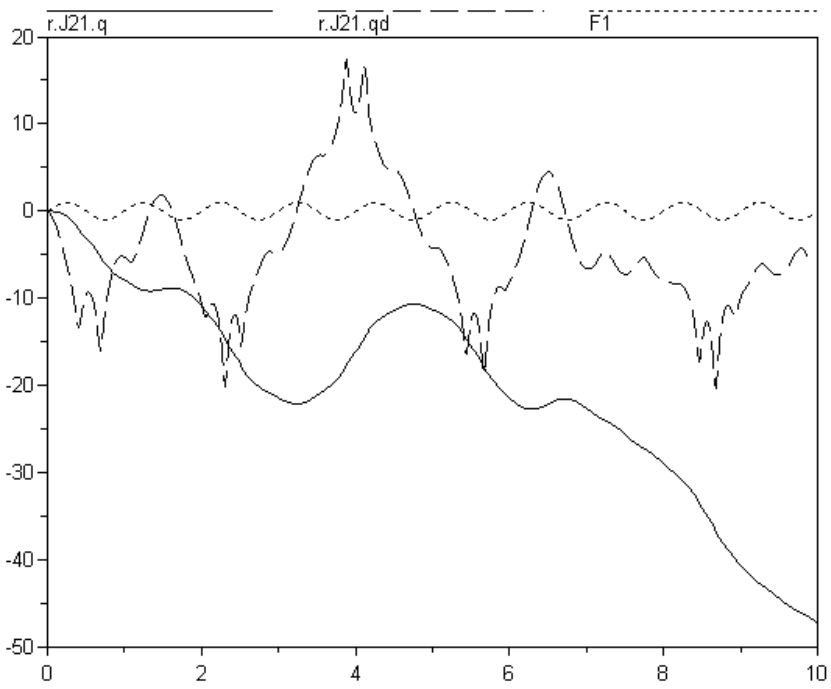


Figure 24: Angle of rotation (`r.J21.q`, solid line) and angular speed (`r.J21.qd`, long dashes) of the crank caused by the applied periodic force (`F1`, short dashes).

as the swing's own, a resonance occurs.

This mechanism (see Figure 26) contains four revolute joints in the loop and one outside the loop. Of the four joints one should be with state variables (because of one degree of freedom), two without state variables (because there are two constraints in the cut joints `RevoluteCut2D`), and one joint is replaced by `RevoluteCut2D`. In this mechanism these joint types can be chosen arbitrarily between the four joints in the loop. The resulting Modelica diagram is shown in Figure 27.

The wrapping model should define the torque applied to the connector "roll" (a black square in Figure 27) so that the joint between the "man" and the "horse" oscillates.

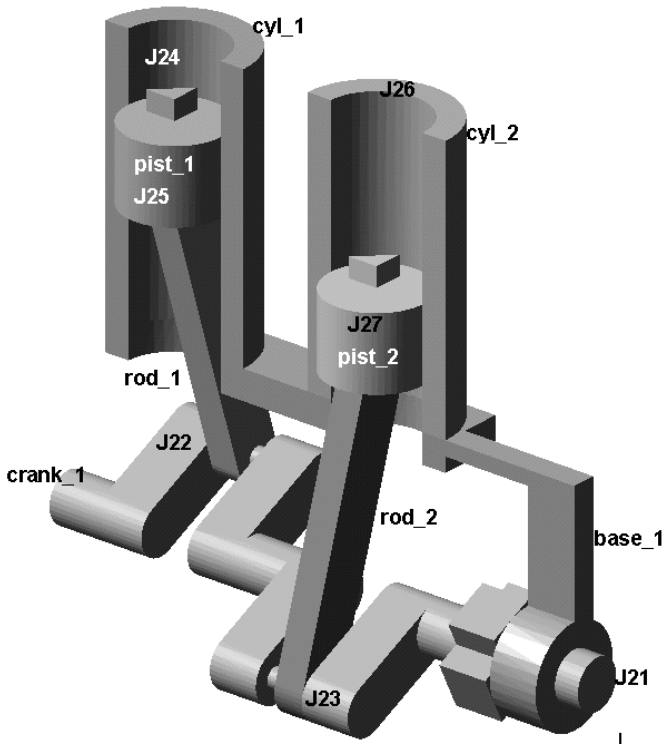


Figure 25: Dynamic 3D visualization of the crank model. Annotations were added to the image later for clarity.

```
model Motor
 "produces a torque such that the angle of the joint q is close to qref"
  parameter Real  k = 0;
  parameter Real  d = 0;
  Real qref, q, qd, qdd, torque;
  LineCut nDrive(across={q; qd; qdd}, through={-torque});
equation
  torque = k*(qref-q) - d*qd;
end Motor;

model world
    Inertial I;
    Motor D1 (k=10000,d=1000);
    Swing1 swing "generated from SolidWorks";
    parameter Real  period_start=0.252;
    parameter Real period_speedup=0.015;
    Real  period "changes gradually";
    parameter Real  amplitude = 1.5;
 equation
    period=period_start+time*period_speedup;
    connect(I.b, swing.a);
    connect(D1.nDrive, swing.roll);
    D1.qref=amplitude * sin( time* 3.1415*2 / period );
end world;
```

The wrapper model in this case describes an experimental setup. The period of

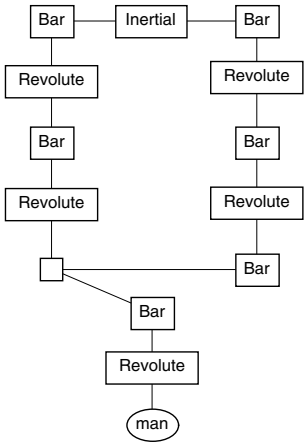


Figure 26: Kinematic outline of the swing mechanism. The types of the revolute joints are not defined yet.

oscillation changes from 0.252 to 0.327 during 5 seconds. Starting from a certain instant (see Figure 28) the resonance effect can be observed.

The simulation can be dynamically visualized by the MVIS tool (see Figure 29).

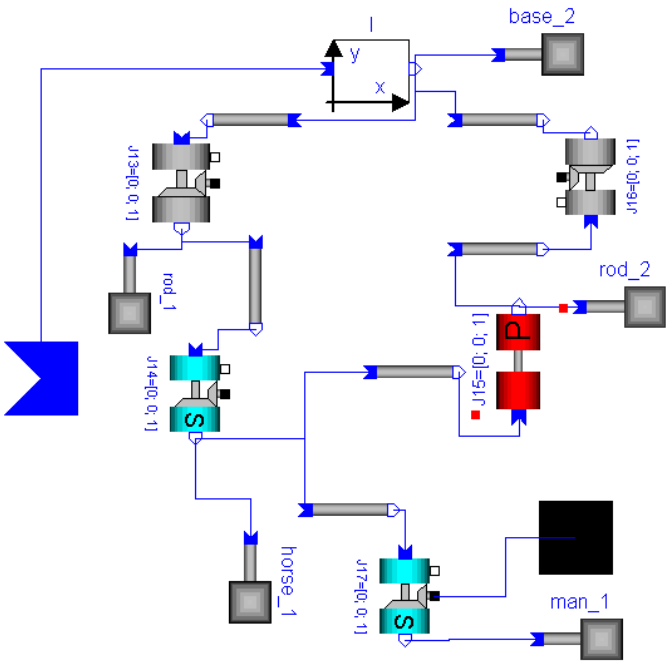


Figure 27: Modelica diagram of the swing mechanism.

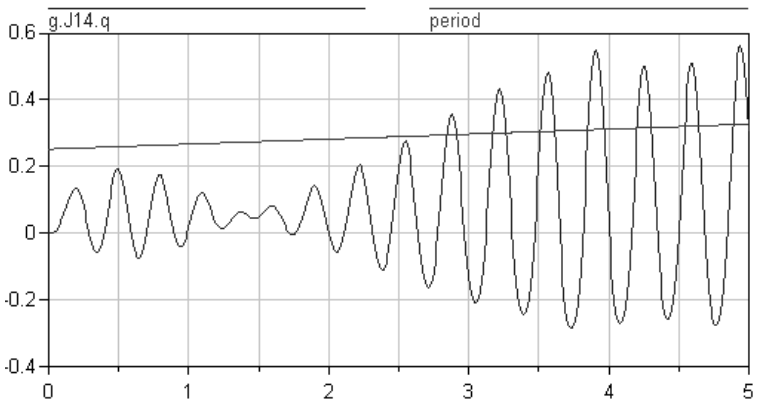


Figure 28: Oscillation period (changes from 0.252 to 0.327) and the angle of rotation of joint `J14`.

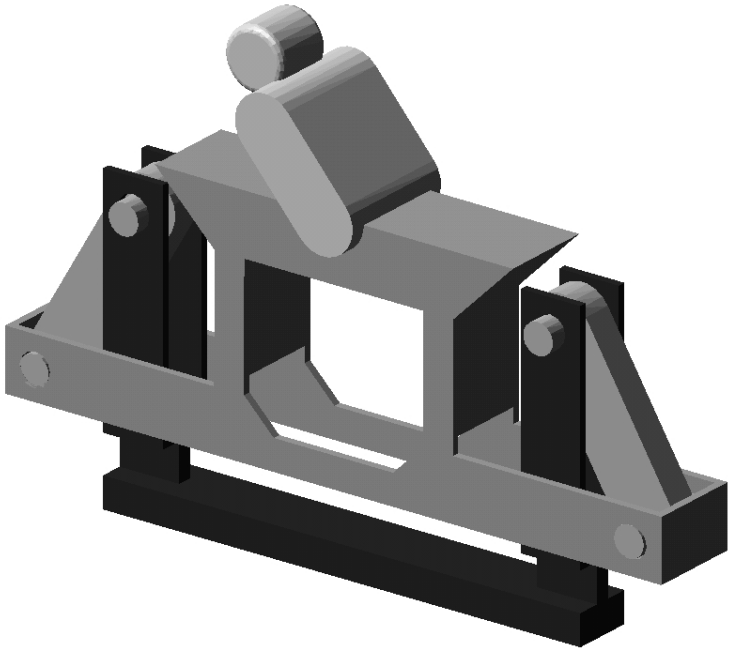


Figure 29: Dynamic visualization of the swing model: a stylized ”man” swings on a swinging ”horse”-like platform. This dynamic visualization not only shows the model in motion, but also helps to validate the design, for instance, detect clashes (i.e. undesirable collisions).

6 Structure of the Integrated Environment

Figure 30 represents components of our integrated environment for design, simulation and visualization of mechanical models. A model designed in SolidWorks serves as the starting point in the system design. The model consists of an assembly document and several part documents. The parts are used in the assembly.

The assembly contains information about position, orientation and categories of the mates. From each part information about its geometry as well as mass and inertia can be extracted. Our translator from SolidWorks to Modelica takes information about the mates and produces a corresponding set of Modelica class instances with connections between them. The mass and inertia tensors for each part are computed by SolidWorks. These are extracted and used in the Modelica model. Geometry information is saved in a separate STL [1] or VRML [54] file for each part.

By default the gravity force is applied to the mechanical model. Usually this is not enough for simulation. All external forces that are applied to the bodies, as well as motor forces that are applied to revolute and prismatic joints, should be specified. This is done outside the SolidWorks model by adding code for new class instances to the Modelica model. For instance external forces can be specified by adding an instance of the `ExtForce` class from the MBS library. Forces can be passed to mechanical model objects from drive train objects or from electrical objects through corresponding transformations. Such objects can be instantiated from classes described in the Drive Train library and Electrical component library. A control subsystem that controls the forces according to a certain plan (mission) can be written in Modelica. If necessary, arbitrary external functions can be called from the model.

All instances of classes in the system constitute a collection of equations binding a set of variables. This is a system of differential and algebraic equations. If initial conditions are given, the solver is normally able to find a solution, i.e. values of all variables for each time step. When a mechanical model is simulated, the position and the orientation for each time step for each part (`Body` instance) is computed. Other useful information, like forces, torques, linear and angular velocity, linear and angular acceleration for each `Body` instance is also available and can be used for analysis of the dynamic system.

For Modelica simulations we have used the Dymola tool with Modelica support [15].

The results of a simulation can be presented to the user in many different ways. Two main types of presentation are 2D and 3D visualizations.

The 2D visualizations are graphs that represent values of some variables (e.g., v) of the model relative to the modeled time (t,$v(t)$). It is also possible to generate parametric plots (e.g. w with respect to v). Such plots are built by the concatenation of lines between consecutive points ($v(t)$,$w(t)$). The 2D visualization is available from the `Plot` window of the Dymola environment.

The 3D visualizations are scenes that display the geometry of the parts in mo-

tion prescribed by simulation results. There are several variants of 3D visualizations available. The choice of variant is determined by the environment preferred by the user. We make a difference between *offline* and *online* visualization. Offline visualization starts when the simulation is finished and visualizes a recorded trace of the simulation. Online visualization takes place during simulation. The user can steer simulation via a graphical user interface and obtain feedback from the simulation via 3D visualization. Additionally, an online visualization tool can be used in order to record and replay a certain fragment or a whole simulation.

The following visualization possibilities have been investigated in the context of integrated Modelica environments:

- The offline visualization tool DymoView, as a component of the Dymola environment. This tool is already integrated into that environment. It can display certain predefined shapes in the MBS library (box, cone, cylinder etc.). It has also facilities for visualization of external files in the DXF[7] format.

- Our offline visualization tool MVIS (Modelica VISualization tool). This tool reads the trace of information on position and rotation of bodies and is able to display complex geometric 3D shapes of arbitrary complexity, stored externally in STL[1] or VRML[54] format. This tool can display the predefined shapes as well.

- The online visualization toolkit MVIS-server. This tool is linked together with the simulation application and obtains the position and rotation of each body via the Modelica external function interface. The tool has the same display capabilities as the MVIS tool. Besides that, MVIS-server is integrated with our MODIC (MODelica Interactive Control) graphical user interface in order to steer a running simulation by control signals from the user.

- Our online visualization toolkit Modelica-VRML. This tool consists of both client and server applications. The server is linked together with the simulation application and sends position and rotation of every simulated object at each time step. The client is a combination of compiled Java code and a VRML[54] browser, running as an Internet browser plug-in (e.g. to Internet Explorer). The tool can display arbitrary shapes, which should be given as VRML files. The performance of the application is limited by the capabilities of VRML browsers. There is a configurable dialog window that can be used for steering simulation via this visualization environment. Users can specify values for signals which are sent to the simulation.

- The online visualization toolkit Modelica-Cult3D. This tool also consists of a client and a server application. The server is also linked together with the simulation application and sends positions and rotations of all simulated objects at each time step. The client is a Cult3D[11] object with compiled Java

code embedded in it. This object is displayed in the usual Internet browser window. The Java routines communicate with the server via sockets. There is a configurable dialog window that can be used for communicating steering signals to the simulated model. The tool can display arbitrary shapes, which should initially be given as VRML or STL files. These files are translated afterwards to 3DStudioMax objects, which in turn are converted to Cult3D objects.

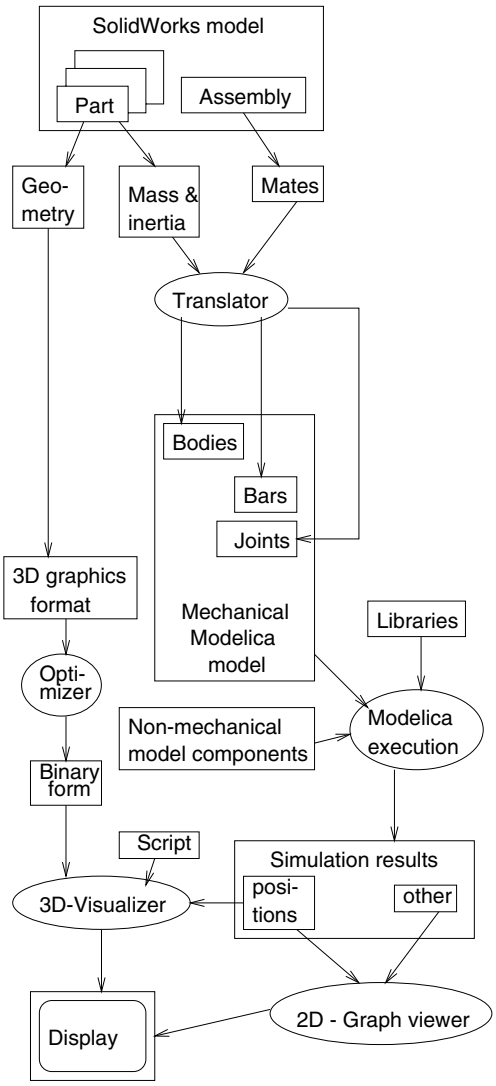


Figure 30: The path from a SolidWorks model to dynamic system visualization.

7 Requirements for Visualization of Mechanical Models

Simulations of mechanical and multidomain models can be interactively visualized in many different ways. It is a difficult task to chhose the right functionality and to guarantee satisfactory interactive performance. To acieve these goals, creation of visualization software is usually a combined effort of its designer and its potential user. In many cases, this is the same person. Since visualization is often used for revealing new phenomena that are not known in advance, it is hard to predict in detail all the visualization requirements needed by potential users. Still, experience has lead to a rather comprehensive set of general requirements, both design requirements (how this software should be designed) and usage requirements (visualization needs by the end user).

7.1 Design Requirements

Since substantial design, coding and debugging efforts are usually spent on visualization of computational results, it is important that such software adheres to some general design requirements.

- **Component-oriented design** Visualization software should be created in a component-oriented way, i.e. using, re-using and combining tools, toolkits, libraries and other software components. In particular, the efforts spent for writing specific program code for each visualization application should be minimized. An application should ideally just incorporate the necessary reusable components which have well-defined functional interfaces. The AVS/Express[3] tool often used in scientific visualization (especially for volume visualization) and FEM visualization, is a typical example of component-based design. This tool has an extensible library of fine-grain visual programming objects that provide a complete design and visualization environment.

- **Program design automation** Visualization software should ideally be created automatically. This would eliminate substantial programming efforts. For each specific application and specific engineering goal appropriate configuration settings can be chosen. Of course, software is never created completely automatically. By automatic programming we mean machine translation of concise and user-oriented high-level formal specifications into nontrivial amounts of procedural program code in some programming language. This can be achieved in two ways: through *compilation* from one computer language (or specification format) to another, or as *interpretation* of specifications using an interpreter.

- **Open design** Software should have open interfaces in order to attach new program components when necessary. This can be achieved by using libraries of classes and by object-oriented design. Availability of source code makes debugging, performance tuning and porting much easier.

- **Lightweight components and core** Since rendering speed is very important in visualization applications, the design should avoid using computationally "heavy" components that can affect visualization performance. Such components can be interpreters of high-level specification languages (that can be rather slow), data preprocessing during visualization, using program memory heap extensively, network communication and file input/output during dynamic visualization.

7.2 Usage Requirements

The usage requirements of visualization software depend on the goals of visualization. Several typical goals are enumerated below:

- **Visual testing and debugging** Visualization is usually necessary in order to check the correctness of a model. The simulation engineer may have some expectations about the model behavior. At this stage he or she can possibly observe that the movements of the model do not match the expectations. Strictly speaking, the only formal way to validate model correctness is to check that the mathematical model matches the physical laws and that the simulation result matches the mathematical model. In practice engineers build different mathematical models for the same phenomena, perform simulation by different simulation tools, and if the results are identical, this can be used as a strong argument for the correctness of the mathematical model.

  Visual testing and debugging normally requires schematic representation of the 3D world (including object geometry and coordinate systems), navigation facilities, and display of object trajectories. If the 3D objects move from one position of the 3D world to another, the virtual camera should be moved too. Therefore it is often necessary to attach the camera target point to a point on the moving object. Sometimes it is necessary to visualize objects that do not exist in the physical model, but they are used in computations, e.g. abstractions of obstacles should be displayed when obstacle avoidance algorithms are used.

- **Comparison of simulations** Often simulation is performed for the purpose of parameter optimization. It can be difficult to formulate formal criteria for the choice of parameters. Instead, simulations with different parameter sets are compared, and the visualization of the results is used for this comparison. When two 2D plots are compared, it is necessary to place them in the same coordinate system. Observing the difference between these plots can be very useful for comparison. When two 3D visualizations of mechanical models are compared, these should be usually placed in different windows. As an alternative, two models can be rendered in the same window, but using different rendering styles (e.g. wire-frame and filled color).

- **Search and presentation of phenomena** Visualization can be performed to facilitate searching for certain phenomena, such as oscillation, unexpected motion trajectory, unusual velocities and forces. These can be first observed in 2D plots. After that, a more detailed study of 3D motion is often necessary. It might happen that the relative motion of 3D objects is much smaller than their size. If the motion is so small that it maps to a distance less than one pixel, the phenomena cannot be observed at all. In this case zooming is necessary in order to enlarge the visualization scale of some region.

  Alternatively, *motion magnification* is used. All displacements of objects from their home position are multiplied by some coefficient (magnification coefficient) and are then displayed. Such visualizations have been successfully used in a roller bearing visualization tool[23] where the motion corresponding to studied phenomena is between 100 and 10000 times smaller than size of the whole model.

- **Model presentation** Often visualization is used just for presentation of the models. In this case, the computations can be approximte, but the visual realism of the visualization is the primary concern for the users. Such visualizations require more information than normally used for engineering applications: colors, textures, lighting models, background color, smooth shading and realistic shadows. A visualization can potentially become a piece of art which requires substantial creative work. It is important to ensure that default settings for all these parameters give a reasonably good visualization. However, it should be possible to tune these attributes further in order to achieve even better image quality.

- **Performance** The number of frames per second, or number of triangles per second, is normally used to measure the performance of graphical applications. For engineering visualizations, the bottleneck is usually the number of triangles of complex mechanical graphic models. When CAD models are designed, many round features are created. Each such feature corresponds to one or several spline surfaces (also called NURBS [58]). Certain graphics acceleration hardware may have support for such surfaces in future. Most graphics hardware today has no such support. Therefore hundreds or thousands of small triangles need to be rendered when such surfaces are visualized. It is important to reduce the number of the triangles in this case. This also can be done automatically by increasing the granularity of subdivisions. A performance between 5 and 10 frames per second is sufficient for typical engineering goals.

  For non-engineering applications visualization realism becomes more important. Appropriate materials, textures, and lighting models should be used. These features, however, decrease application performance. A performance of between 15 to 25 frames per second is necessary.

In this section we have only discussed certain generic requirements for visualization. Each application domain and each application usually have some specific requirements, which have not been covered here. In Section 8 below we discuss how the requirements apply for visualization of Modelica simulations.

8 MVIS - Modelica Interactive Visualization Tool

Currently available tools for visualization of mechanical models simulated in Modelica (DymoView) do not match the requirements formulated in Section 7. The reason is, in particular, that DymoView was not designed for technical visualization of such models, but primarily for rough estimation and validation of the 3D layout of models and their movements.

In particular, DymoView (a component of the Dymola tool[15] used for 3D visualization) currently lacks support for visualization of lines, online visualization (i.e. visualization simultaneously with running simulation), using graphic hardware acceleration on UNIX machines, etc.

In addition to the requirements formulated in Section 7 there are the following needs:

- Portability.
- Possibility to use hardware graphics acceleration.
- Possibility to extend the visualization with new graphical objects (such as lines and coordinate axes),
- Visualization of arbitrary shapes with specific color, material, or texture on each face.

We have made experiments with a number of graphical tools, such as AVS[3], 3DStudioMax[5], MultiGen[38] and graphics libraries such as Maverik[9]. Some of our requirements cannot be satisfactory resolved by these tools. Therefore a new tool has been constructed.

Our contribution to this area is the development of a new visualizing tool, MVIS (Modelica VISualizer) for dynamic visualization of Modelica simulations. This visualizer has a dual purpose: it is designed both for technical visualization and for high quality model presentation.

Depending on the particular application and the simulation goals either online or offline visualization should be created for Modelica models.

The visualization architecture and tool names are shown in Figure 31.

Online visualization occurs during simulation execution. In this case the simulation routines are linked together with our visualization routine library (MVISLIB), or they communicate via TCP/IP sockets. In the latter case, the simulation tool works as a server, and the visualization tool is a client.

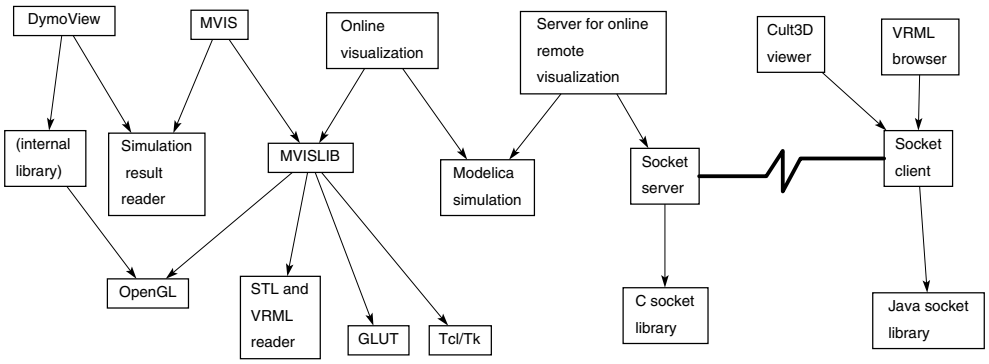


Figure 31: Visualization tool architecture. The Internet-based visualization is discussed in Section 9.

Offline Modelica visualization tools (MVIS and DymoView) read the simulation results from a file which is initially created and saved by the simulation tool. Depending on file format and file writing routines, the offline tool can start reading *before* the simulation has finished writing. This, however, is currently not possible to do with the Modelica implementation in the Dymola tool because it uses a special binary format to store simulation results which cannot be read until it is completely written.

All graphical facilities of online visualization and the MVIS tool are based on the MVISLIB library, which works with OpenGL, processes various geometry formats, creates windows and menus using GLUT and Tcl/Tk.

8.1 Offline and online Visualization Interfaces from Modelica

This section discusses the way in which Modelica simulations currently create data structures used in visualizations (using the tools MVIS and DymoView). The functionality of MVIS and MVISLIB is discussed in Section 8.2.

8.1.1 Data structures used in visualization

Data values for online and offline visualization are the same, but their use is different. In the offline variant the data is stored in Modelica **`output`** variables , which automatically appear in the simulation result file (see Figure 32). In the online variant these data are sent over to a special external function (see Figure 33)

Modelica simulations in Dymola tool save static and dynamics data used for visualization together with other simulation results in a MATLAB-compatible format. This format and corresponding Dymola and Modelica classes were designed by Hilding Elmqvist[15]. Files written in this format contain several MATLAB array variables, in particular a list containing Modelica variable names with references to the corresponding column in the array of values. For each visualized

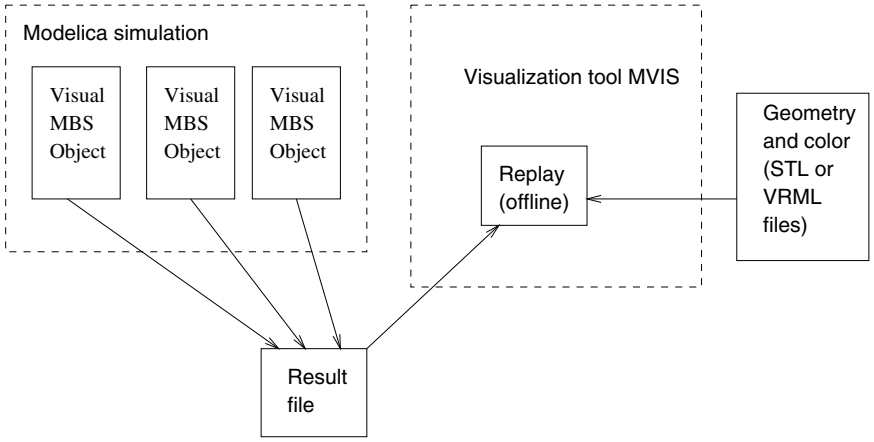


Figure 32: Offline visualization of Modelica simulations.

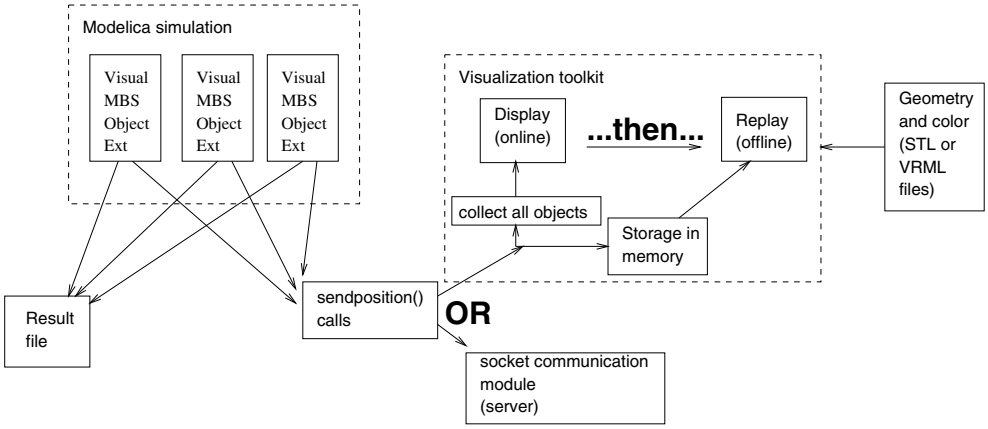


Figure 33: Online visualization of Modelica simulations.

object several variables are needed. They are represented in Table 3. Some of the variables change during simulation.

The format described in Table 3 is an extension of a format used in the Dymola environment. Predefined shapes can be displayed in DymoView (a 3D visualization tool used in Dymola). Custom shapes are displayed there as bounding boxes only. The MVISLIB library can visualize all three kinds of objects: predefined shapes, custom shapes and lines.

The MVIS tool is a stand-alone application that reads visualization data from a file with simulation results. MVISLIB has facilities to read and parse necessary STL or VRML files (mi`.stl` or mi`.wrl`, where i =`stlIndex`) with geometry and color definitions, and display the corresponding graphical objects.

A Modelica model should use certain library classes in order to have visualization data saved in the described format. These classes differ by the method they use to obtain position and rotation of the shape from the simulation. This position

Field name	Type	Meaning
`Shape`	`Integer` (`101...108`) encoded into `Real`	One of the eight predefined shapes (box, sphere, cylinder, cone, pipe, beam, wirebox, gearwheel). The value `101` is used for custom shapes and lines.
`r0`	`Real[3]`	Vector from the origin of the world coordinate system to the current origin of the shape (or line)
`rx`	`Real[3]`	Current direction of the x axis of the shape
`ry`	`Real[3]`	Current direction of the y axis of the shape
`size`	`Real[3]`	These values are specified in the local translated and rotated frame. Size of the predefined shape. Size of the bounding box for custom shapes. Current vector for a line.
`mat`	`Real[4]`	Color (red, green, blue and alpha components) for predefined shapes and lines. Default color for custom shapes (can be overridden by colors in VRML).
`extra`	`Real`	Extra size used for predefined objects cone and pipe.
`stlIndex`	`Integer`	0 for predefined shapes; 11–999 : index of a file in STL or VRML format; 2001–2099 : line of specific thickness

Table 3: Visualization data format saved by Modelica applications

and rotation can be obtained via a mechanical connector `MbsCut`. This connector is used in mechanical models and contains position, rotation, force and some other variables (see Section 3.5.1). In mechanical systems bodies and other components of multibody systems connect to each other using this connector. This connector is also used for objects describing visualization.

8.1.2 Force and Torque Equations for Visualization Classes

This section explains a subtle difference between two groups of classes in the existing MBS library, defining visualization data in Modelica models. This difference does not affect visualization, but should be taken into account when connections between class instances are set up, and when new classes are created. This sections explains the meaning of the column 3 in Table 4.

A difficulty arises with the *force* component[8] of the `MbsCut` connector. Modelica models contain objects describing physical bodies (with mass and inertia) and objects describing visualization data (with colors, shapes etc.). These objects should be attached (connected) to the rest of Modelica model using `MbsCut` connectors. When two or more `MbsCut` connectors are connected, equality of position and rotation between corresponding frames is established. Simultaneously, a balance of forces is established, since the variable f (force) is included in the connectors.

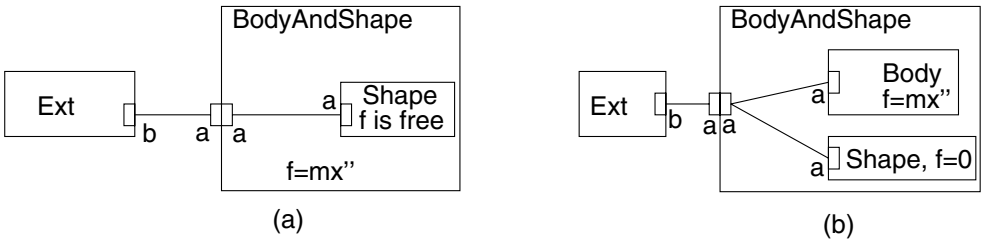


Figure 34: Relations between mechanical cuts of bodies and shapes. Two variants are considered: (a) shape within the body and (b) shape is connected to the body.

There are two variants of connections between body classes and visualization classes:

- The visualization object can be a subcomponent of an object describing a mechanical body, e.g. the `BodyAndShape` object in Figure 34(a). In this case the forces (represented by the **`flow`** variable f) in the connectors are constrained by equation $BodyAndShape.a.f + Shape.a.f = 0$. Since the visualization should not affect the forces, $Shape.a.f$ is a free variable in the `Shape` object.
- The visualization object can be attached as an independent component to the connector used for the body (Figure 34(b)). In this case three `MbsCut`

[8]The same discussion and classification applies to the torque component.

connectors are connected together. The forces (f) in the connectors are constrained by the equation $BodyAndShape.a.f + Body.a.f + Shape.a.f = 0$. Since the visualization should not affect the forces, the equation $f = 0$ should be specified in the `Shape` object.

Because of these two different situations, two different classes are needed for shapes classes with the equation $f = 0$ and without it.

8.1.3 Standard and New Classes for Visualization

A summary of standard classes and additional classes is given in Table 4. The column *Body* specifies whether a model of a physical body (with mass and inertia) is included in the model. The column *Number of MbsCuts* specifies how many connectors of type `MbsCut` belong to the model.

The class `VisualMbsObject` is typically used in applications. In particular, the SolidWorks-to-Modelica translator produces models which directly or indirectly include `VisualMbsObject`. This class computes output variables described in Table 3. The simulation environment provides that these variables are written to the file with simulation results.

The class `VisualMbsObjectExt` is an alternative to `VisualMbsObject`. It contains a call to the external function `sendposition`. This function has the following signature:

```
function sendposition
   input Real t;
   input Real id;
   input Real shape;
   input Real Form;
   input Real rxvisobj[3];
   input Real ryvisobj[3];
   input Real rvisobj[3];
   input Real size[3];
   input Real material;
   input Real extra;
   input Real stlIndex;
   output Real dummy;
equation
  external
end myfun;
```

This function has the same formal parameters as the output variables of `VisualMbsObject`. When the model is used for online visualization all these parameters are sent to the visualizing part of the program (see Figure 33). As soon as the data with positions and rotations of all the objects has arrived, the program draws all the objects in their current position. The function returns a value to a dummy **`output`** variable. Therefore the simulation environment calls this function at each communication step[9].

[9]The size of the communication step defines a fixed simulation time interval between two consecutive records in the file with simulation results. This is not the solver iteration step.

Model name	**Body**	**Number of `MbsCuts`**	**Shapes**	**Remarks**
Standard classes				
`BodyV`	yes	One, f is free	predefined	Uses `VisualMbsObject`
`Visual-Shape`	no	no	predefined	Requires input variables for position and rotation
`Visual-Object`	no	no	predefined	Requires input variables for position and rotation
`BodyShape`	yes	Two	predefined	Uses `MbsShape`
`Bar` combined with `VisualMbs-Object`	no	Two	predefined	Can be constructed if needed
`VisualMbs-Object`	no	One, f is free		
`MbsShape`	no	One, $f = 0$		
`Cylinder-Body`	yes	Two	pipe	Uses `VisualMbsObject`
`BoxBody`	yes	Two	box	Uses `VisualMbsObject`
-	yes	no	-	Mass cannot be used of the object is not connected via `MbsCuts`
Extension for offline visualization				
`BodyME`	yes	One, f free	custom	Uses `VisualMbsObject`
`ShapeME`	no	One, f free	custom	Uses `VisualMbsObject`
`BodyME0`	yes	One, $f = 0$	custom	Inherits `BodyME`
`ShapeME0`	no	One, $f = 0$	custom	Inherits `ShapeME`
Extension for online visualization				
`BodyMEF`	yes	One, f free	custom	Uses `VisualMbsObjectExt`
`ShapeMEF`	no	One, f free	custom	Uses `VisualMbsObjectExt`
`BodyMEF0`	yes	One, $f = 0$	custom	Inherits `BodyME`
`ShapeMEF0`	no	One, $f = 0$	custom	Inherits `ShapeME`

Table 4: Summary of Modelica classes used for visualization.

All information regarding the positions and rotations of the bodies is saved in memory and can be replayed again.

The Table 4 contains the entries for the eight classes we designed for online and offline visualization.

8.2 Rendering Properties and Design Aspects

The integrated environment includes a visualizer that provides online dynamic display of the assembly (during simulation) or offline (based on saved state information for each time step).

The STL (STereo Lithography) format [1] is a very simple format suitable for visualization. All surfaces are divided into triangles. The coordinates of the triangle vertices, as well as the normal vectors of the triangles, are listed in the STL-file.

The VRML (Virtual Reality Modeling Language) format [54] is a standard language which is primarily used for the definition of complex scenes consisting of virtual objects in three-dimensional space. The objects defined in this language contain a geometry (defined via primitives or sets of polygons), materials (color, texture mapping, lighting properties), light sources and cameras.

The visualizer loads the corresponding STL or VRML file for each part and optimizes it for rendering. After that, rendering is performed by OpenGL [50] library functions. During the optimization all the vertices positioned very close to each other are merged together. Arrays with point coordinates are stored in a binary file (cache) for future use. On the average, these binary cache files are four times smaller than the original STL files and can be read much faster.

The user of the visualizer can alternatively utilize the pop-up menu system, keyboard shortcuts, or a command string in order to control various options. We found that the following facilities (that can be turned on and off) should be available. All these facilities have been implemented and evaluated in our MVISLIB library and are available for online and offline visualization.

Moving the camera The major advantage of 3D visualization is the possibility to observe objects from different viewpoints. For this purpose rotating, moving and zooming facilities have been implemented. The most often used facility is rotating the model around the target point of the camera. Initially the target point coincides with the origin of the coordinate system. The rotation of the model can be performed in two directions (left-and-right and up-and-down). In some cases it is necessary to displace the camera, together with the target point. This can be done in three orthogonal directions (X, Y and Z). After the camera is moved, rotation is performed around the new target point.

Zooming in and out Zooming in and out changes the scale of the image. The camera is not moved during zooming in and out. An automatic zooming facility chooses such a scale that all of the objects fit into the window[10].

[10]This facility is not implemented at the moment of writing this report.

Attaching a camera to a body For visualization of a small object moving a relatively large distances it is desirable to direct the camera so that the moving object is seen all time at the center of the view. For this purpose the camera can be targeted to the reference point of a particular part. When the part moves, the camera will move together with the object. An extension of this mode is a facility to rotate the camera together with the target part.

Projections A perspective and orthographic projection can be used for visualization. For engineering purposes an orthographic visualization should be used. It preserves the sizes and the angles of the model independent of the distance between the camera and the model. The perspective projection is used when more realism is necessary.

Lighting 3D visualization depends on the positions of light sources, their properties, positions of surfaces of the visualized objects relative to the light sources and their material properties. We found that in order to obtain high quality technical visualization lighting should often be adapted for the model. The visualization tool has facilities to change light source positions, and to change the ambient and diffuse component of each light source. In this way any number of light sources can be manipulated. However, for technical visualization it is enough to operate with just two light sources. In some cases only outline (wire-frame) of the model or rendering in flat shading is used. The user can also combine the wire-frame and light shading together in order to observe the exact borders of objects in the scene.

Hiding Certain parts in 3D environments occlude others. To solve the occlusion problem and observe occluded fragments of the model it is often necessary to hide some parts.

Environment Display of an application-specific landscape, for instance, a road for car simulation, or a runway surrounded by a hilly landscape for flight simulation. Such a landscape can be created as a large mechanical part that does not move, or directly in C using OpenGL. The rendering of the environment is very important for achieving realism, for presentation purposes. It can also be used for engineering visualization, in order to put mechanisms into an environment familiar to the engineer.

Trajectory One of the most important purposes of visualization is the comparison between the expected behavior of a simulated system (mission) and the actual behavior of this system (motion trajectory). For this purpose a special file with mission description can be given to the visualizer. The mission is visualized as a line in 3D coordinate space connecting several control points displayed as small boxes. The motion trajectory of a moving object in the simulated mechanism can be displayed and compared to the mission.

Grid For engineering visualization the origin of the coordinate space and the origin of each body are shown with three coordinate axes (red, green and blue, – corresponding to the X, Y, and Z axes). In order to estimate the distances between bodies in the visualization environment, a grid is used. The grid can be placed in the XY, YZ or XZ plane.

Pseudo-shadow The pseudo-shadow is not dependent on light at all. We found that for flight simulation it is convenient to display a projection of a vehicle and its trajectory on the ground plane.

Animation Animation is very important for engineering visualization. 3D animation helps the user to observe complex motion patterns, relative rotation of some parts, relation between motion of different mechanism components. There are standard controls for animation: starting, stopping, continuing animation, stepping forward and backward. There are controls specific for simulation applications: changing the stride, i.e. the number of simulation output steps which are omitted between two consequent displayed frames. For instance, animation can be configured so that it displays only every second or every third simulation step. If the animation is displayed too fast, or time steps are not equidistant, it is useful to synchronize the animation with the machine clock. In this case, a special delay is inserted between animation steps.

8.3 MODIC, Modelica Interactive Control Interface

Modelica simulations can input and output values via a graphical user interface during simulation. From the Modelica side this is done using external function calls. Then these external functions create or modify graphical windows, output values to these windows, or read the signal value which is currently set by the user.

8.3.1 Interface for Output Values

In the simplest form the external function signature for output values is defined as follows:

```
function outfun "sends data to GUI"
  input String   label "label for the widget";
  input Real      val   "current value";
  output Real funres "result is ignored";
equation
  external
end outfun;
```

This function is called the following way:

```
  output Real dummy "help variable";
  Real val "a value to be output";
equation
  dummy=outfun("Label",val);
```

The Modelica language specification does not guarantee that the function `outfun` is called with a certain frequency. However, the Modelica simulation environments ([15] and [31]) have a facility to specify the *communication step* i.e. how often the `output` variables (e.g. `dummy`) are written to the simulation result file. Normally the function `outfun` is called with the same frequency as that specified by the communication step size.

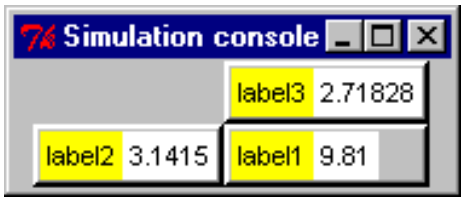


Figure 35: Presentation of output data in the dialog window of interactive simulation.

When the function is called for the first time, a label with the specified text appears in the interactive dialog window (Figure 35). Each time the function is called a numerical value is printed in the window near the label.

The position of the label in the dialog window can be specified by `row` and `column` parameters:

```
function outfungrid
  input String   label;
  input Real      val;
  input Integer  row;
  input Integer  column;
  output Real funres;
equation
  external
end outfun;
```

This function is called the following way:

```
  ...
  output Real dummy;
  Real val;
equation
  dummy=outfungrid("label1",val,1,1)+
        outfungrid("label2",val,1,0)+
        outfungrid("label3",val,0,1);
```

The + sign is used just in order to reduce the number of necessary dummy variables. The columns in the window are numbered from zero, starting from left to right; the rows are numbered from zero, starting from the top and downwards.

If necessary the output can be encapsulated into an instance of a library class:

```
model  OutputWindow " Library class for output window"
  Real val;
  parameter String   label="Undefined";
  parameter Integer  row=-1;
  parameter Integer  column=-1;
  output Real dummy;
equation
  dummy=outfungrid(label,val,row,column);
end OutputWindow;
```

```
model World "example of use"
   OutputWindow w1(label="Speed",row=3,column=4);
   ...
equation
  w1.val=robot.speed;
end World;
```

8.3.2 Interface for Input Values

In the simplest form the function call for input values is defined as

```
function infun "a signature for input function"
  input String  label    "a text label to place in GUI";
  input Real    timeval  "current simulation time";
  input Real    startval "a value to begin with" ;
  input Real    minval   "expected smallest value";
  input Real    maxval   "expected largest value";
  output Real funres;
equation
  external
end infun;
```

```
  ...
  Real force;
equation
  force=infun("Label",time,0.0,-20.0, 20.0);
```

When this function is called for the very first time, a label, a window for editing the value, and a scale bar is created. The scale bar has the a lower and an upper limit (`minval` and `maxval`). The row number (`input Integer row`) can be specified when the function `infungrid` is called. The code fragment

```
   force1=infungrid("label4",time, 10, 1.0, 20.0, 2)
   force2=infungrid("label5",time, 0.6, 0.1, 0.7, 4);
```

creates a fragment of input data window displayed in Figure 36.

8.4 Synchronization Problem in the Interface for Input Values

This section explains the problems caused by simultaneous input and output of time- dependent data in interactive visualizations.

In interactive simulation environments human users interact with simulations. Users get some visual information for each simulation step and are able to instantly

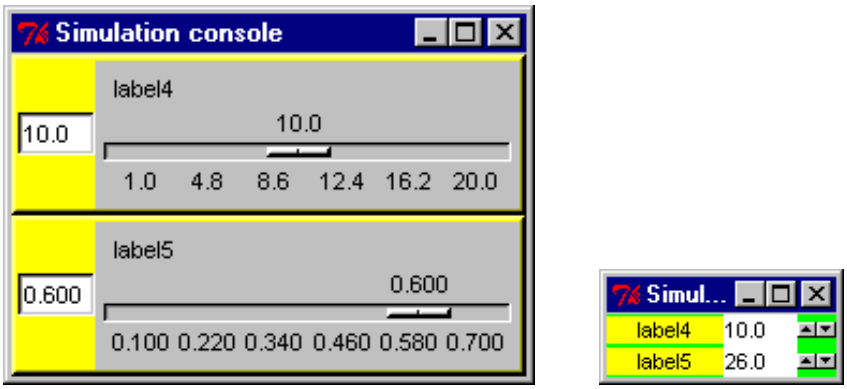


Figure 36: Presentation of input data in the dialog window of an interactive simulation, (a) – with scale bar; (b) – with buttons for incrementing and decrementing values.

respond with signals which are based on this visual information and immediately affect the simulation.

For a human, the interaction with an ideal simulation environment should not differ from interaction with a physical system in the real world. Just as a steering wheel of a car changes driving direction, moving the scale bar control should affect the solver and new motion should be reflected in the interactive online visualization. It can be noted that interactive simulation is similar to simulation where an external control system is involved.

The visualization of simulation results is a function of the growing simulation time value t. For *interactive* simulation it is necessary, that visualization goes forward all the time, and it is synchronized with the human perception of time. The user receives some visual information, and almost immediately responds with a signal.

The input signal (i.e. input for Modelica) at an instant t is the user response, based on visualization of the output signal for some time interval up to the instant $t - \Delta t$, where $\Delta t > 0$ is a very small time interval. An ideal value Δt would be simulation time interval between two subsequent frames of visualization.

Visualization and interaction is more realistic if the simulation time matches the time of the machine clock. However for some simulation models this is very hard to guarantee, since solution methods with adaptive step are often used. One second of simulation time may correspond to a CPU time interval of a length between 0.1 and 100 seconds.

The major problem of using interaction during computations is that interactive simulation output cannot always be synchronized with interactive simulation input[11].

[11] Another major problem is discontinuity of the input signal, discussed later.

A numerical ODE solver is used in order to obtain the simulation results. The solver work consists of a number of solver iterations. At each iteration certain simulation time instant is used for approximate computation of all state variables. At each iteration external functions needed for computation of state variables are called. These external functions, in particular, should perform input of signals from the user interface.

There are two major groups of numerical ODE solvers – fixed step solvers and adaptive solvers. They have different stability properties, in particular, fixed step solvers cannot handle non-trivial mechanical simulations in realistic time. Adaptive solvers attempt to predict an appropriate step size each time. At each iteration computation error is calculated, and if it is too large then a smaller step size is used. As result, it often happens that a larger time value is used in a solver iteration *before* a smaller time value. Therefore, time sometimes goes "backwards".

The simulation environments have facility to specify k, a *communication step size*[12]. At solver iteration steps with simulation time $t = 0, k, 2k, 3k$, etc., the state variable values are found (with specified accuracy), and all external functions called, if they are needed for computation of output variables. In particular, a function producing online visualization is called (since its result is a dummy **`output`** variable).

In order to solve the ODE system the adaptive solver should know the input for a simulated time t in order to produce the output for time t. Since time can go "backwards" such solver cannot directly be used for interactive visualization.

Let us consider the situation closer in order to highlight the problem.

During Modelica model simulation a system of differential and algebraic equations is solved. Assume that a differential equation $x'(t) = F(t, u(t))$ is being solved during the simulation. We assume that $x(t)$ is an output value, and that $u(t)$ is an input value. In order to visualize the output value $x(t_1)$ at simulation time t_1 the equation should be solved at the time value t_1.

For some differential equation solvers (e.g. based on the Euler method) it is enough to know $F(t, u(t))$, where $t \leq t_1$. However, many other solvers, specially those handling stiff problems (solvers with adaptive time steps, using an implicit method of solution), utilize so called trial steps and time step prediction methods (see Figure 37). Therefore $F(t_1 + h, u(t_1 + h))$ is needed for the computation of $x(t_1)$. Here $h > 0$ is the predicted time step. Usually several prediction steps are tried in a sequence, with smaller and larger values of h.

The value of $u(t_1 + h)$ is requested from the graphical user interface, i.e. from the user. However the user cannot specify this value because he or she have not seen the development of the visualization between the time step t_1 and $t_1 + h$. The only thing the user can do is to specify the value (we denote it u_1) based on observation at the previous visualized step of the simulation, e.g. the very last step observed at time $t_1 - \Delta t$.

[12]The size of the communication step defines a fixed simulation time interval between two consecutive records in the file with simulation results. This is not the same as solver step.

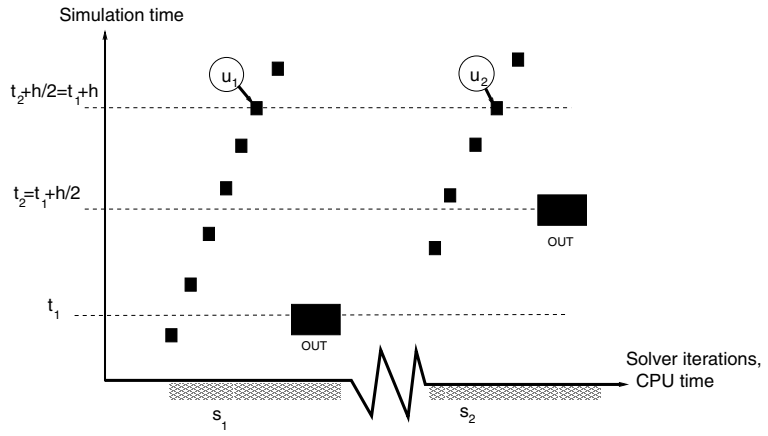


Figure 37: History of solver iterations and values obtained via the input from the user interface during these interactions. This diagram assumes that a method with prediction steps is used and some new input value is obtained at each solver iteration. The boxes with out mark places when communication step occurs and therefore the visualization function is called. Both u_1 and u_2 are input signals coming from the user interface and specify input value for the simulation time instant $t_1 + h$.

The solver continues the solution process assuming that $u(t_1 + h) = u_1$.

Assume, that after some simulation steps the solution for time $t_2 = t_1 + h/2$ is searched. The prediction step may occur to be any small positive number, for instance $h/2$. I order to find the solution $F(t_2 + h/2, u(t_2 + h/2))$ is needed for computations. This expression is equal $F(t_1 + h, u(t_1 + h))$. This time the *same* value $u(t_1 + h)$ is requested from the user, who already observed the behavior of the visualization until t_2. Based on these observations, the user specifies a new value u_2. This value may differ from u_1. The solver continues the solution process assuming that $u(t_1 + h) = u_2$.

Normally ODE solvers assume that the function $F(t, u(t))$ is continuous and differentiable up to a certain order and that it can have only one value at $t = t_1 + h$. In case of interactive input it might occur that this function gets *two* different values at $t = t_1 + h$.

This leads to a contradiction, and the solver behavior is undefined in this case. The solver might produce wrong results, or it does not converge at all, causing fatal simulation error. The larger is difference between u_1 and u_2, the less stable is the solver. Obviously, this is inadmissible in case of end-user interaction with the system.

There are three solutions to this problem. All of them attempt to avoid simulation errors, causing, however some other disadvantages.

Using sampling for interactive data input The statement

```
when sample (offset,period) then ... end when;
```

stops integration each `period` simulation seconds, activates equations within the statement, calls the function, and restarts the integration with new values of the variables. An example of using this statement is given below.

```
when sample(0,0.1)  then
  grab := infun("Grab", time, 0.0, 0.0, 1.0);
end when;
```

The function `infun` is called only once for each sampled instant, and it happens every 0.1 seconds. No contradiction between values may occur. A disadvantage, however is that due to restart of the numerical integration, the simulation works slower than without interactive input. Another disadvantage is that the input data from the user is not propagated to the application immediately: it takes up to 0.1 seconds of simulation time to deliver the new value to the equation solver.

It is not clear how to stop integration just when user changes some value and do not stop it in other cases, like below:

```
when user_changed_input_value()  then
  grab := infun("Grab", time, 0.0, 0.0, 1.0);
end when;
```

The problem is that the value of condition in `when` statement is computed "literally". The fact that user will change the value at simulation time t should be known to the solver in advance, i.e. some time before the model is processed at event instant t.

Using solvers without prediction A Modelica simulation has a choice of using different types of solvers. In some cases solvers that do not use prediction work well enough. In this case no contradiction between values may occur. The input signal from the user is directly delivered to the solver. It is not continuous and therefore must be smoothed some way.

The disadvantage is that solvers without prediction are less stable than adaptive solvers with prediction. It is hard to guess whether a given equation system can be solved by a solver without prediction. This is even harder to do if parameters and some input variables can unexpectedly change.

Indirect change of values In some cases input values can be changed indirectly. The user can, for instance, manipulate the average speed of changes of a variable. Assuming that t_0 and u_0 are fixed, and $v(t)$ is an input signal from the user, we define $u(t) = u_0 + (t - t_0)v(t)$. The values t_0 and u_0 can also be specified at run time by the user, but they should not change as often as $v(t)$. The difference between u_1 and u_2 will be small, and $u_1 = u_2$ if the input signal v does not change. The disadvantage is that this way of

manipulation is rather unnatural for the user. This approach can be used if the user "behavior" is known in advance and only some parameters should be tuned slightly. This approach generates a function that is continuous and has the second derivative 0 in most intervals.

Smoothing of input signals The Modelica solvers require that all external functions during continuous integration are differentiable up to n-th order (where $n \geq 1$ is the order of numerical integration method used in the solver).

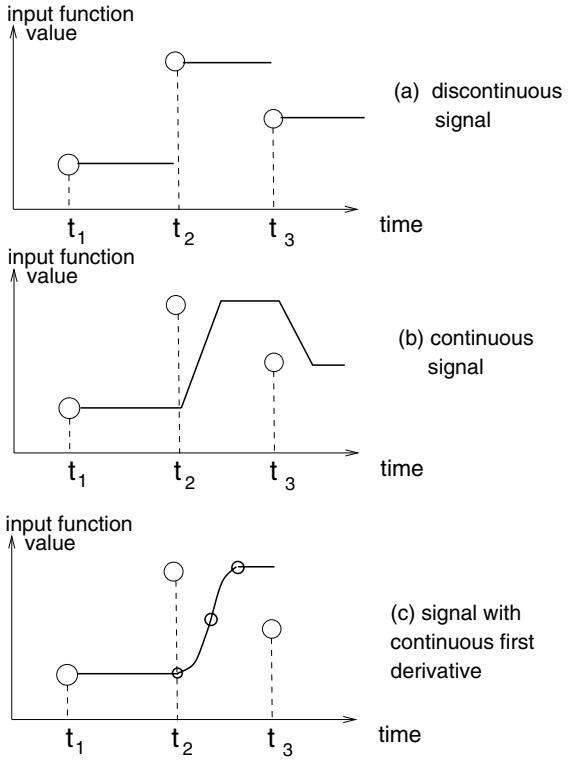


Figure 38: Smoothing data before it is returned from external function. The circles represent the values obtained at sampling time t_1, t_2, t_3. Since an external function cannot be discontinuous, one of smoothing algorithms should be performed.

Therefore all input data should be smoothed using one or several splines connected together. Two variants of smoothing of the initial discontinuous signal (Figure 38(a)) the are represented : continuous (Figure 38(b)), and signal with continuous first derivative (Figure 38(c)). The last one is constructed by gluing together two second order polynomials and one constant interval. Higher order splines can potentially be used too. This approach requires some more experimentation with different solvers and applications which will be performed in the future. Similar problem with discontinuous input signal which is, however, rather smooth, is discussed in [19].

As we see from this discussion there is no ultimate methods to deal with synchronization between input and output yet. Several approaches with advantages and disadvantages are considered. Our future work will be directed to finding an appropriate combination of solvers, their parameters and input signal smoothing techniques.

9 Modelica Visualization on the Internet

There are several reasons why it is highly relevant to visualize Modelica simulations using the Internet:

- Modelica simulations can be executed on a powerful workstation. An end-user or model designer can continuously access the result of certain simulations on his/her local personal computer with limited computational power. High speed networks or modem communication can be used for interaction between these computers.

- If the designers working on a certain Modelica model are geographically distributed it is necessary to communicate simulation results in visual form. Collaborative environments, such as virtual offices and meeting rooms for design of mechanical models can be created using virtual reality technologies. Currently, commercial solutions exist only for static CAD models. For instance, CoCreate Software Inc. developed the OneSpace tool[8] based on their special CAD tool SolidDesigner. This tool can be used for collaboration between developers who can observe and point on model components simultaneously. At each time interval the model is "owned" by just one of the designers.

  Another virtual collaborative environment, Dive[12] supports cooperative work with virtual objects. These objects or their parts can move, and these movements are either responses to events triggered by the user, or some predefined movements which can be programmed in Tcl/Tk. There is a project on using broadband communication with Dive and other systems for distributed collaborative CAD engineering[53].

  However there are no mature tools yet that support cooperative work with interactive, physically based simulations of mechanical models.

  There exist general purpose collaborative work tools, such as Microsoft Netmeeting. However, they use bitmap images of the screen for sending information over the network which greatly reduces the communication speed. The tools for cooperative work should primarily send over 3D geometry and motion data[13], like what is done in the Dive environment and our Internet-based Modelica visualization environments with VRML and Cult3D discussed in more detail in the next two sections.

[13] Of course, video and bitmap data can be useful as well.

- Modelica simulation results should be made available to a wide range of users working on different platforms. These results should be placed in interactive environments. For the end user the Modelica technology behind the simulation should be ”transparent”, i.e. the user gives input signals and inspects the output, without noticing whether Modelica or some other simulation environment is used for simulation.

Two approaches to visualization of results of mechanical modeling on the Internet have been developed in our work: using VRML and Cult3D. In both cases two software components, a client and a server, are used. Socket communication is established between these components. The components can run on the same computer or via the Internet:

Modelica server A Modelica simulation server computes the positions and rotations of bodies in 3D and sends this information at each simulation step to the client. The server can also accept user input signals and place the signal values into some Modelica variables (see Sections 8.3.2 and 8.4 for details regarding the problems of interactive input).

Java-based client The client reads the correct positions and rotations of the bodies from the socket, and displays the objects. If the user changes any signal values, the client sends user input signals to the server.

9.1 VRML-Based Simulation Visualization

VRML (Virtual Reality Modeling Language) is a standard format for static and dynamic scenes used on the Internet and on Intranets. The format contains nodes placed in a tree hierarchy. The nodes define shapes (spheres, triangular meshes etc.), colors, lighting and behavior (reaction to events). During dynamic visualization events are triggered by the user or by behavior nodes.

VRML files can be edited by conventional text editors. However, usually VRML data is exported from modeling tools (CAD tools, computer animation tools or specialized VRML tools).

In the VRML-based simulation visualization, a Java program (client) is connected together with a scene description in VRML so that coordinates of objects shown in a VRML browser are affected by the current values in the client (Figure 39). In order to connect Java and VRML the External Authoring Interface (EAI) is used. The Java applet is running on the client computer in an Internet browser. This applet communicates with the server through an Internet TCP/IP socket connection.

There exist two design alternatives for connecting Java with VRML:

- The VRML97 standard uses a mechanism of VRML events and requires definition of Java script nodes. The script nodes receive information about user actions and events that happen in other nodes. After that the nodes

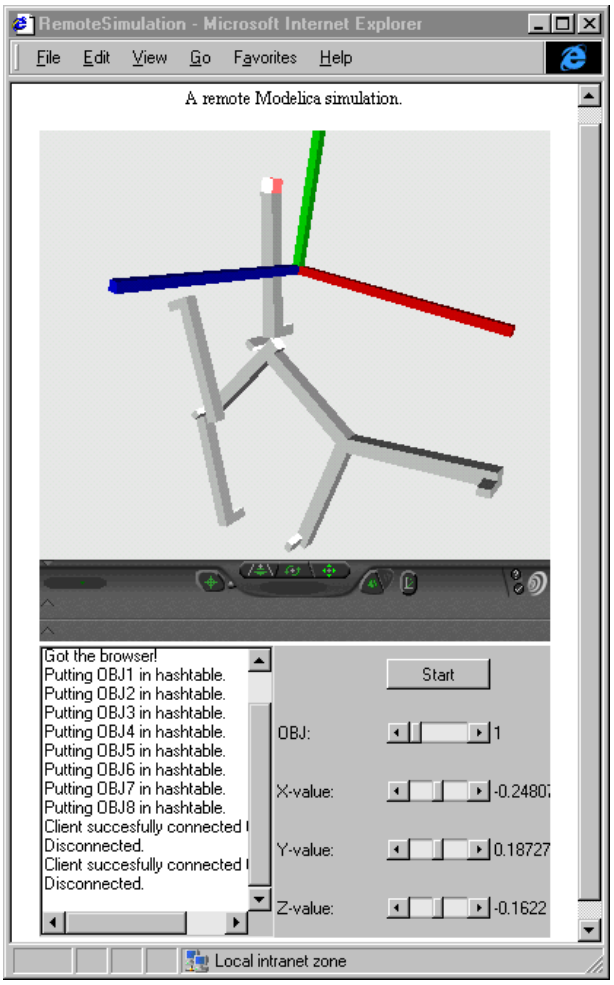


Figure 39: Interactive visualization of pendulum simulation in Modelica using a VRML browser.

execute a program module and send events to other nodes. In particular, such a node can refer to a `.class` file. During browsing the node can activate a Java class, and a link between the VRML scene and the Java applet or application can be established.

Normally this method should be used when movements are initiated and controlled primarily by the VRML scene and VRML browser.

- VRML2.0 which adopted the name VRML97 has another method called External Authoring Interface. This method is not standardized yet, but is implemented by several major VRML browsers. This method has better control over the VRML scene. In particular, it has access to functionality of the Browser Script Interface. Also, it can send events to nodes inside the scene and get events which are sent from nodes of the scene.

This method is preferable when the movements are initiated and controlled by an external program written in Java.

In an External Authoring Interface-based applet, a handle of an instance of the browser is obtained in the applet `start()` method. After that the Java program using External Authoring Interface classes can navigate in the tree of the VRML scene, fetch nodes and manipulate with events and nodes responsible for translation and rotation.

As a VRML browser we use the most popular one, CosmoPlayer[10]. It is implemented as a plug-in for Internet Explorer and Netscape.

The major problem of this approach is performance limitations of the link between VRML and Java. The actual 3D rendering is fast, but Java libraries and the link between these libraries and the browser have quite low performance. Since the External Authoring Interface facility in an internal part of the browsers we could not easily overcome this problem.

For a scene containing 8 bodies (there were 18 triangles in each body) the animation performance is just 2 frames per second. This speed is not affected by the socket communication. However the same scene in the same browser without the Java interface is rendered with approximately 30 frames per second[14].

Our conclusion about this approach is:

- VRML-Modelica communication is robust, it is easy to control and maintain.
- The currently available VRML browser do not reach the performance necessary for engineering applications.

9.2 Cult3D Approach

Cult3D[11] is a recently developed format and plug-in rendering engine for static and dynamic 3D graphics on the Internet. Scenes described in this format contain the following components:

- A static tree of nodes is the skeleton of a scene. The nodes represent 3D objects, light sources, cameras, etc.
- Surface geometry, position, rotation, color, and lighting parameters are attributes of the nodes. These can change dynamically.
- An event map is stored in the Cult3D format. When Cult3D objects are interactively visualized, events are triggered by user actions. There are also internal events, e.g. ”World start” and ”World step” which are triggered by the browser. Calls to Java methods or modification of node attributes can be responses to these events.

[14] Here and in the further experiments we were using a 266 MHz PentiumII-based computer without graphic card and with 128 MB RAM.

- Compiled Java classes. Java programs can manipulate dynamic attributes of the nodes, in particular, position and rotation.
- Textures are attached to 3D object geometry.
- Sounds are additional resources and can be activated by events.

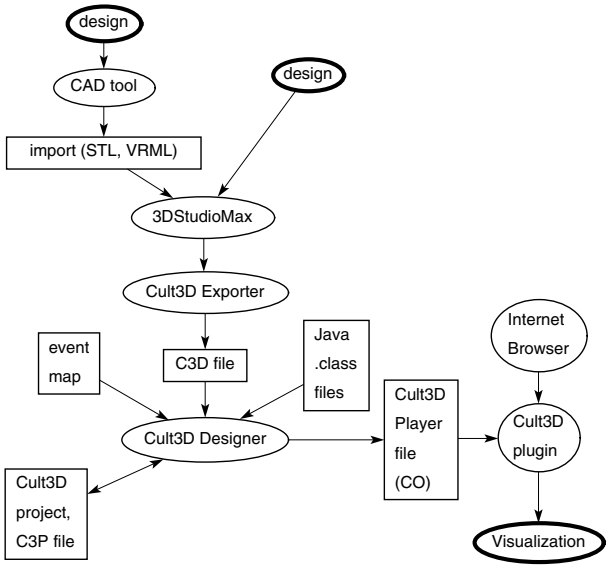


Figure 40: Steps of Cult3D design.

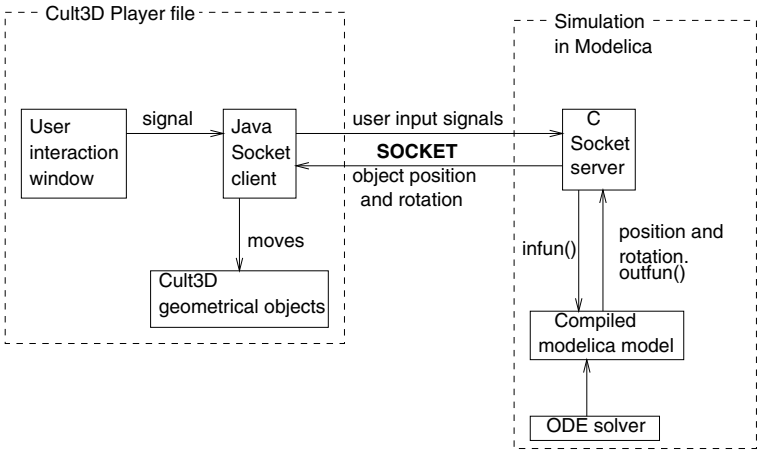


Figure 41: The client-server model of Cult3D-based visualization.

Files in the Cult3D format can only be produced by a tool called Cult3D designer. These files are stored in an internal, very compact, binary format and therefore cannot be modified. The design of Cult3D objects goes through several steps

(see Figure 40). Initially a model is designed in some CAD tool and imported into 3DStudioMax[5]. Alternatively, the model is designed in 3DStudioMax directly. A Cult3D exporter plug-in for 3DStudioMax extracts the information about 3D objects, light sources, and cameras, and stores it in a binary Cult3D design file (a C3D file). The Cult3D Designer is an interactive tool that reads the C3D file and has facilities for the specification of an event map and Java functions calls. The resulting Cult3D player file contains all necessary components and can be displayed by the Cult3D plug-in in an Internet browser window.

When Cult3D is used for visualization of Modelica simulations, the Java classes contain the following components (see Figure 41).

- The Java client obtains the current position and rotation of each body via socket communication.
- A pop-up Java AWT window is created for interactive control of simulations. The user can specify input signal values during simulation. These values are then delivered to the server and affect the simulation.
- Cult3D API methods are called in order to specify the position and the rotation of each 3D object for the current time frame.
- The method connected to the "World Step" event is activated each time Cult3D is about to redraw its objects. This method calls routines that obtain the new positions and rotations of all the objects from the server. After that these positions and rotations are sent to the corresponding Cult3D object.

Our experiments show that this application has fairly good performance. A mechanism consisting of 6 bodies (totally 900 triangles) is visualized with the speed of 10 frames per second. The major difficulty with Cult3D visualization construction is that a number of design tools have no command line interface. The availability of command line interfaces for these tools would allow automatic construction of Cult3D environments directly from CAD models.

9.3 Using 3DStudioMax

3DStudioMax[5] has a facility for automatic creation of non-interactive frame sequences (animations). This is done by importing positions and rotations of 3D objects at certain time moments. The geometry of the 3D objects used in simulation is imported into 3DStudioMax. STL or VRML format are used for this purpose.

Frame sequences in 3DStudioMax are defined via keyframes. A keyframe is associated with a certain frame index (moment of time) and a key. A position and an orientation of objects are examples of the keys. In order to define an animation, it is enough to specify the position and orientation of the objects at some keyframes, and 3DStudioMax automatically interpolates these keys to obtain smooth motion of objects between these frames.

The moment of time, rotation angles and object position can be specified in the script language MaxScript which consists of interpreted commands used in 3DStudioMax. A sample script is shown below:

```
at time 0.51 (
  r=[89.92,0.01,-88.73]
  p=[160.6,146.2,823.6]
  f.rotation=(eulerangles r) as quat
  f.position=p)
at time 0.57 (
  r=[89.92,0.01,-88.73]
  p=[185.7,175.4,823.6]
  f.rotation=(eulerangles r) as quat
  f.position=p)
```

This script specifies different object positions at the time moments 0.51 and 0.57 produced as a result of the simulation. 3DStudioMax interpolates the positions at the time moments between these steps and produces a smooth animation sequence. The script is automatically generated from simulation results of a multibody system simulation. 3DStudioMax has facilities to specify camera and light source positions and attributes (they can change during animation). In the final result a high quality animation can be obtained and saved in the AVI or MOV format.

This visualization tool has an advantage of high flexibility of rendering. Light conditions, shadows and many advanced rendering and animation features of 3DStudioMax can be used. The resulting animation in AVI format can run with virtually any speed. A disadvantage is that when the animation is ready, the user cannot affect the visualization, and even cannot take different view angle. In addition, AVI files are rather large. Rendering with quality suitable for engineering application takes 5 seconds per frame, and each frame occupies between 30 and 50 KBytes (depending on desired color quality). A one minute animation (with frequency 12 frames per second) is rendered in one hour and it occupies at least 20 MBytes. If textures are added for higher scene realism, rendering takes 15-30 seconds per frame.

10 Conclusions

Several design and visualization components for integrated environment for simulation of mechanical and multi-domain models has been implemented using Modelica as a standard model representation. Many mechanical and general-purpose simulation tools exists, but only Modelica is able to integrate CAD-based design, extendibility of equation collections, interactivity, reasonable performance and robustness, and high performance interactive visualization environment. Complex models can be rapidly created, simulated, and visualized locally or via the Internet.

In addition to crank and swing models described in this paper, helicopter and industrial robot model [56, 47, 19] were successfully designed and simulated.

Connection between CAD and Modelica made possible also to integrate collision detection and response into mechanical simulations in Modelica[20].

The tools described in this report were implemented as working prototypes which should be further developed and extended, in particular as part of the MathModelica environment[31] and as contribution into the RealSim project [48] of the European Commission.

11 Acknowledgments

The Modelica definition has been developed by the Eurosim Technical Committee 1 (Modelica Design Group)[35] under the leadership of Hilding Elmqvist (Dynasim AB, Lund, Sweden) and Martin Otter (DLR, Germany). The Dymola tool has been designed by Dynasim AB. The Multibody Simulation Library has been developed by Martin Otter. The work has been supported by the Wallenberg foundation as part of the WITAS project [56] and the European Commission as part of the RealSim project [48]. The master students Håkan Larsson [30], Daniel Larsson[40] and Andreas Gustafsson [27] participated in tool development.

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] ADAMS and Mechanical Dynamics *Adams*, ADAMS and Mechanical Dynamics, Inc., http://www.adams.com

[3] Advanced Visual Systems Inc., AVS/Express. http://www.avs.com

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

[5] Autodesk Inc., *3DStudioMax*, http://www.ktx.com

[6] Autodesk Inc., *Mechanical Desktop*, http://www.autodesk.com

[7] Autodesk, Inc., *Autocad 2000 documentation. Drawing Interchange File Format.*, http://www.autodesk.com

[8] CoCreate Software Inc., OneSpace, http://www.cocreate.com/

[9] Jon Cook, Maverik, a system for managing display and interaction in virtual reality applications. http://aig.cs.man.ac.uk/systems/Maverik/

[10] Cosmo Software, *CosmoPlayer (VRML browser)*, http://www.cai.com/cosmo/

[11] Cycore AB, *Cult3D Home Page*, http://www.cult3d.com

[12] Dive research group, Interactive Collaborative Environments Laboratory, Swedish Institute of Computer Science, *The DIVE tool*, http://www.sics.se/dive/

[13] Dynabits, *WWW page*, http://www.dynabits.com

[14] Hilding Elmqvist, *A Structured Model Language for Large Continuous Systems,* PhD thesis TFRT-1015, Department of Automatic Control, Lund University of Technology, Lund, Sweden.

[15] Hilding Elmqvist, Dag Brück, Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual, Version 4.0*, from Dynasim AB, Research Park Ideon, Lund, Sweden, http://www.dynasim.se

[16] Hilding Elmqvist, Sven-Erik Mattsson. Modelica – The Next Generation Modeling Language – An International Design Effort. In *Proceedings of First World Congress of System Simulation*, Singapore, September 1–3 1997.

[17] Hilding Elmqvist, Sven Erik Mattsson and Martin Otter *Modelica - A Language for Physical System Modeling, Visualization and Interaction*. Plenary paper. 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Hawaii, August 22-27, 1999

[18] Hilding Elmqvist, *Personal communication*, April 2000.

[19] Vadim Engelson, *Simulation and Visualization of Autonomous Helicopter and Service Robots*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 013. Available at: http://www.ep.liu.se/ea/cis/2000/008/

[20] Vadim Engelson, *Integration of Collision Detection with Multibody System Library in Modelica.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 010. Available at: http://www.ep.liu.se/ea/cis/2000/010/

[21] Vadim Engelson, *Integration of Modelica and 3D Geometry.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 009. Available at: http://www.ep.liu.se/ea/cis/2000/009/

[22] Thilo Ernst, Stefan Jähnichen, and Mattias Klose, The Architecture of the Smile/M Simulation Environment, in *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modeling and Applied Mathematics*, Vol. 6, Berlin, Germany, pp. 653-658, 1997

[23] Dag Fritzson, Peter Fritzson, Patrik Nordling, Tommy Persson, Rolling Bearing Simulation on MIMD Computers, *International Journal of Supercomputer Applications and High Performance Computing*, 11(4), 1997.

[24] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High-Level Mathematical Modelling and Programming, *IEEE Software*, 12(4):77-87, July 1995

[25] Peter Fritzson, Vadim Engelson, Modelica – A Unified Object-Oriented Language for System Modeling and Simulation, in *Proceedings of European Conference on Object-Oriented Programming (ECOOP98)*, Brussels, July 20–24, 1998.

[26] Peter Fritzson, personal communication.

[27] Andreas Gustavsson, *Integration of Cult3D and Modelica Simulations*, Master Thesis, IDA, Linköping University, Sweden, to be published in May 2000.

[28] ISO, *ISO 10303, Industrial Automation Systems and Integration - Product Data Representation and Exchange*, ISO TC 184/SC4, 1992.

[29] Knowledge Revolution Inc., *Working Model 3D*, MSC Working Knowledge / Knowledge Revolution Inc., http://www.krev.com

[30] Håkan Larsson, *Translation of 3D CAD Models to Modelica*, Master Thesis, LiTH-IDA-Ex-99/30, IDA, Linköping Univ., Sweden, March 1999.

[31] MathCore AB, *MathModelica* , available from MathCore AB, http://www.mathcore.com

[32] MathWorks Inc., *MatLab*, http://www.mathworks.com/products/matlab

[33] Sven Erik Mattsson, Hilding Elmqvist, Martin Otter, Physical system modeling with Modelica, *Control Engineering Practice*, 1998, vol. 6, pp. 501–510.

[34] Sven Erik Mattsson, G. Söderlind, Index reduction in differential-algebraic equations using dummy derivatives, *SIAM Journal of Scientific and Statistical Computing*, 1993, 14:3, pp. 677-692.

[35] Modelica Design Group, *Modelica WWW site*, http://www.modelica.org

[36] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[37] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial and Rationale. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[38] MultiGen-Paradigm, Inc., *MultiGen*, http://www.multigen-paradigm.com

[39] Parametric Technologies Inc., *Pro/ENGINEER*, http://www.ptc.som

[40] PELAB *Modelica activities in PELAB*, Programming Environments Laboratory, Department of Computer and Information Science, Linköping University, http://www.ida.liu.se/~pelab/modelica

[41] PELAB, *ObjectMath Home Page*, http://www.ida.liu.se/labs/pelab/omath

[42] Martin Otter, *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. Dissertation, Fortschrittberichte VDI, Reihe 20, Nr. 147, 1995.

[43] Martin Otter, Hilding Elmqvist and Sven Erik Mattsson *Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle*, 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Hawaii, August 22-27, 1999

[44] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, in *Proceedings of the NATO-Advanced Study Institute on Computer Aided Analysis of Rigid and Flexible Mechanical Systems* , Volume II, pp. 91-110, Troia, Portugal, 27 June - 9 July, 1993.

[45] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[46] Jon Owen, *STEP – An Introduction*, Information Geometers Ltd., 1993, ISBN 1-874728-04-6.

[47] Johan Parmar, *Modeling an Autonomous Helicopter and its Maintenance using Modelica*, Master Thesis, LITH-IDA-Ex-99/63, IDA, Linköping Univ., Sweden.

[48] RealSim, *European Commission Research and Development – Project RealSim Consortium Page*, http://www.ida.liu.se/~pelab/realsim

[49] Per Sahlin, Alex Bring, and Ed F. Sowell, *The Neutral Model Format for building simulation*, Version 3.02. Technical Report, Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.

[50] SGI, *OpenGL Web Site*, http://reality.sgi.com/opengl

[51] Sheshardi, K., Peter Fritzson, A Mathematica-based PDE-Solver Generator, in *Proceedings of 1999 Conference of the Scandinavian Simulation Society*, Linköping, Sweden, October 18-19, 1999.

[52] *SolidWorks*, SolidWorks Corporation, http://www.solidworks.com

[53] Peter Törlind, Mathias Johansson, Mårten Stenius, Distributed Engineering, Project report, Luleå University, 1999, http://www.mt.luth.se/division/dmk/research/integration/ding.html

[54] VRML Consortium, *Virtual Reality Modeling Language – Repository*, http://www.web3d.org/vrml/vrml.htm

[55] Waterloo Maple Inc.,*Maple*, http://www.maplesoft.com

[56] WITAS group, *The Wallenberg Laboratory for Research on Information Technology and Autonomous Systems*, Linköping University, http://www.ida.liu.se/ext/witas

[57] Wolfram Research *Mathematica*, Wolfram Research Inc., http://www.wolfram.com

[58] Mason Woo, Jackie Neider, Tom Davis, *OpenGL Programming Guide*, Second Edition, Addison-Wesley Developers Press, 1996.

Paper 6

Simulation and Visualization of Autonomous Helicopter and Service Robots

Vadim Engelson, PELAB, IDA, Linköping University

Abstract

To decrease the costs and the time it takes to develop and test new products, computer simulations are very helpful. Models can be simulated, and their behavior can be examined. This applies not only to hardware, but even to software products that can be divided to several components, so that their cooperative work is simulated in a virtual environment. Some components of this environment can later be replaced by physical, real world devices. Some other components can be just prototypes, and they are replaced later with more complex and realistic software components. In any case the idea is to construct a model and simulate both software and hardware before the actual production starts. In the WITAS project there is a need to develop a system which contains helicopters, robots and various control software and hardware. In particular there is a need to simulate the dynamic behavior of an autonomous aircraft within a virtual environment. There is a need to simulate a service environment, where robots can interact with the landed helicopter.

In this report a study of object-oriented modeling of mechanical systems using Modelica is presented. Mechanical features of an autonomous helicopter have been modeled in order to verify the control system. A robot which is able to grab, move and release objects using automatic or manual control has been modeled. The geometry and dynamic structure of these systems has been designed in CAD tools and later integrated with control systems for steering these devices. The simulation has been performed in Modelica.

1 Introduction

The paper contains a study of simulation of autonomous helicopters and service robots that has been performed in the framework of the WITAS [14] project. That project concerns research in the area of autonomous aircraft and other autonomous systems. One of major objectives of the project is to investigate tasks that unmanned autonomous aircraft will be able to solve in future. For this purpose a virtual environment is needed to simulate these tasks and solutions. The environment includes a command and control architecture with a system of active vision.

One of the tasks for an unmanned autonomous aircraft, i.e. a helicopter could be to monitor traffic situations as well as providing emergency help. The tasks of the robots include servicing the helicopters, as well as loading and unloading various objects to and from the helicopter.

The most complex part of the project includes design of control systems working in several layers. These control systems are organized as software tools in three layers (see Figure 1):

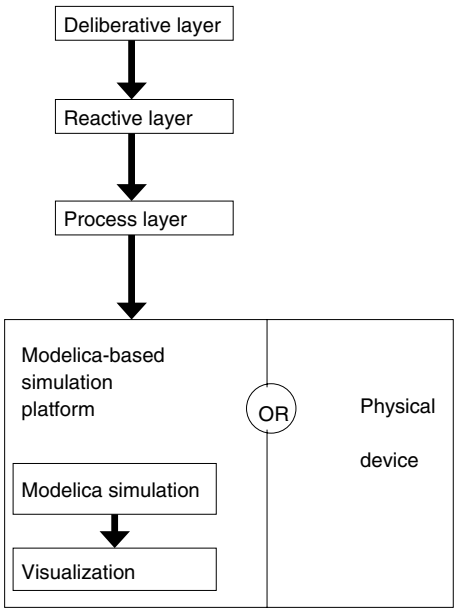


Figure 1: Layers of simulation architecture for an autonomous vehicle.

- A deliberative layer produces plans, e.g. a plan of movement.
- A reactive layer produces responses when certain events happen.
- A process layer gets information from sensors and produces signals for mechanism actuators (e.g. motors).

The mechanical model is placed ”under” these layers. Depending on the environment either a mathematical model of the mechanism is linked to the control system software, or a real physical device in the laboratory or on the field is used as a mechanical model. In order to test and verify the control system the mathematical model and the real device should respond to actuators in the same way and give the sensors the same answer.

Therefore, it is necessary to enhance the realism of simulations in order to verify the correctness of the control systems.

This can be done in different ways; in the case of flying and moving mechanisms it is quite important to take the dynamics of the mechanisms into account.

Initially the models were represented as point masses. Since the dynamics can be simulated in Modelica[7] we designed the corresponding models of the helicopter and the robot in this language.

The assemblies contain many components that move and rotate in certain ways. This information has been extracted [5, 4] from CAD models designed in the SolidWorks tool [13] and glued with other Modelica code. Simulation has been performed using the Dymola tool [2], multibody system library[6], together with online and offline visualization and interactive control[4].

The report contains a description of the helicopter model (Section 2) and the robot model (Section 3). Conclusions are given at the end of these sections.

2 Helicopter modeling

The helicopter model consists of two major parts:

- A control system [12] written in Ada (later translated to C). This control system has been initially tuned with a simplified helicopter model [10, 11] described using expilicit motion equations.
- A mechanical dynamic system modeled in Modelica where the major part of the code is generated automatically by the SolidWorks–to–Modelica translator.

These parts communicate with each other via function calls.

2.1 The control system

The part of the control system which communicates with the Modelica model is the process layer. This system takes the plan (helicopter mission) as commands expressed in a language FCL (Flight Command Language). Typical commands in this language look like `FLY-TO- POINT(x,y,z,v,p)` or `LAND()` where `(x,y,z)` are space coordinates of a mission point, `v` is the required velocity, `p` is the required precision. The command `FLY-TO-POINT` means that the control system will try to steer the vehicle to a specified point so that the magnitude of velocity is `v`. Each command is considered as executed as soon as the vehicle occurs at a distance less than `p` from the target point. Then the next command is requested from the upper layer of the control architecture. For testing purposes the test sequence of FCL commands is just read from an input file.

In order to describe the architecture of the control system, the coordinate frames (Figure 2) should be considered. The origin coordinate frame O corresponds to the ground, inertial system. The body-fixed coordinate system, F is attached to the main part of the helicopter. The relation between these systems is determined by the position of the helicopter and its rotation around each axis. The rotation angles Ψ, Φ and Θ are yaw, roll and pitch rotations around the Z, X and Y axes correspondingly.

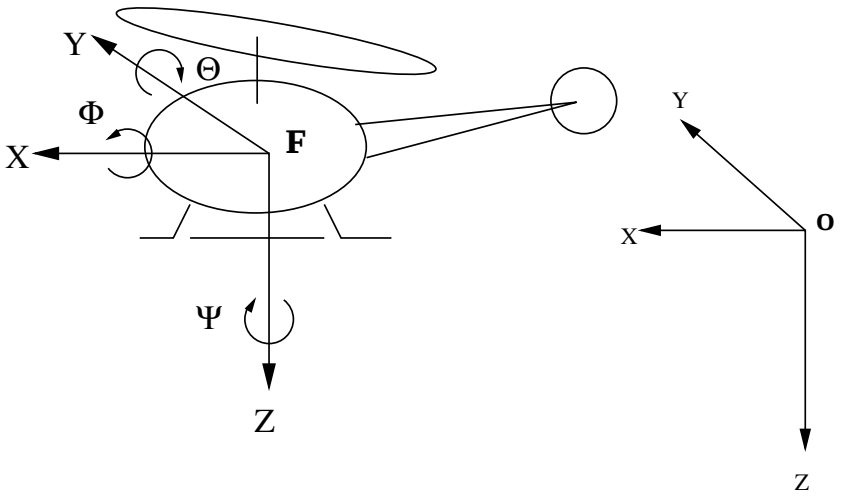


Figure 2: The origin (O) frame and the body-fixed (F) frame.

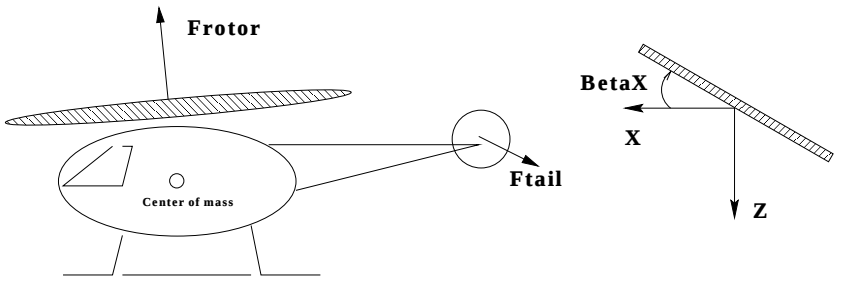


Figure 3: Positions of the forces F_{rotor} and F_{tail} on the helicopter. The angle β_x is the deviation of the rotor plane from axis X.

The magnitude of two forces (F_{rotor} and F_{tail}) and values of two angles (β_x and β_y) are delivered to the helicopter model by the control system (see Figure 3). The force F_{rotor} is a lifting force applied to the main rotor of the helicopter. The rotor is located above the center of mass of the helicopter. The rotor is tilted to the angle β_x around the local Y axis and angle β_y around the local X axis. Finally, F_{tail} is the force created by the tail rotor.

The control system switches between different $modes$. A switch happens when the execution of a new command starts, or the helicopter position or rotation reaches a certain value (i.e. reaches a threshold value and cause an internal event). Each mode has a specific feedback loop which maps current position, velocity and rotation to new force magnitudes and angle values. Rules defining these modes are described in [12, 8].

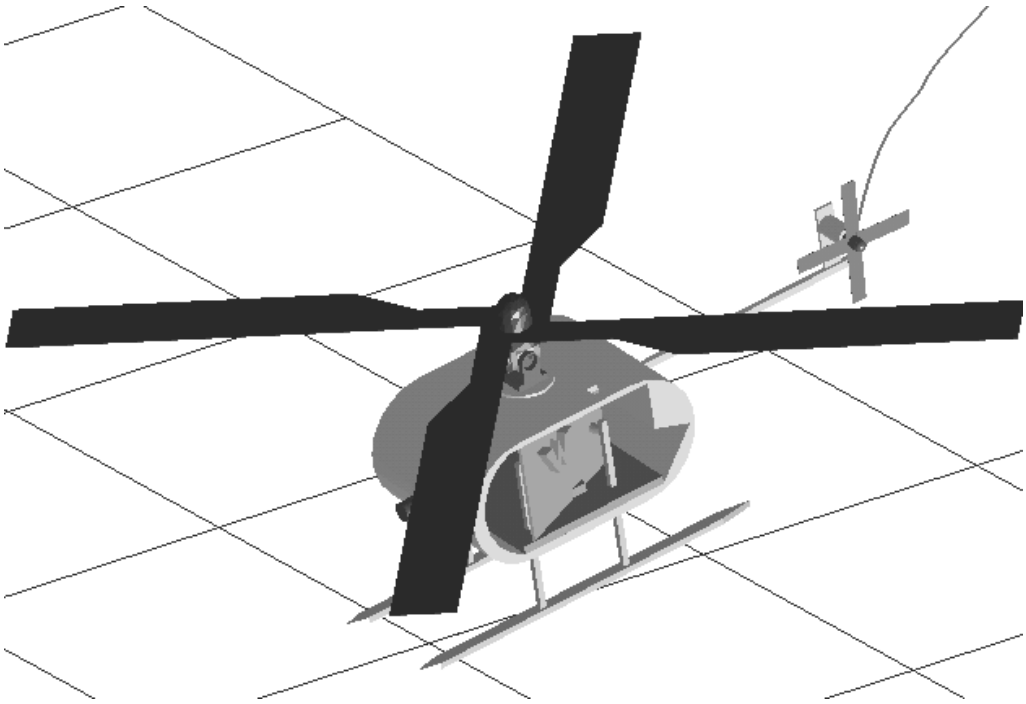


Figure 4: Helicopter model designed in SolidWorks.

2.2 Mechanical model of the helicopter

The mechanical parts of the helicopter model (see Figure 4) has been designed in SolidWorks [13]. There are eleven parts in the model. The kinematic skeleton for the moving parts of the model is shown in Figure 5. The Modelica connection diagram of the helicopter is shown in Figure 6.

The major parts are the main body of the helicopter, the main rotor, the tail rotor, left and right landing gears, and the door. There exist minor parts, which were convenient to define for debugging of the model and integrating with the robot model. The ”reference cylinder” and load platform are attached to the main body. Three intermediate parts used for rotation are inserted between the main body and the rotor. All the parts are considered as rigid bodies. They are connected by rigid, revolute (rotational) and prismatic (translational) joints.

The size of the helicopter used in the project is $1.5 \times 0.5 \times 0.5$ m, the weight is about 50 kg.

The mechanical part of the helicopter is just one of components in the helicopter model. There exist a set of components which cannot be defined just by drawing in the CAD tool. In Modelica these components are connected to the mechanical part by ten `MbsCutB` connectors.

This connector is a data structure which contains position and rotation of a coordinate frame attached to some mechanical part at some point. In addition, this

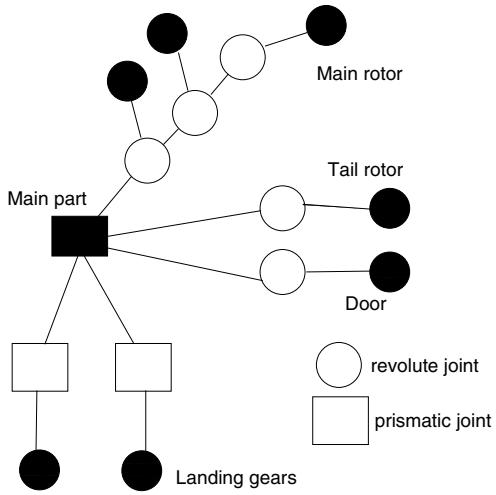


Figure 5: Kinematic skeleton of the helicopter. Five revolute joints, two prismatic joints and one rigid joint are used.

connector contains variables defining the force and the torque that can be applied to this point. They are denoted as pentagons on the leaves of the Modelica connection diagram (Figure 6)

The components used in the model are the following:

Springs in the landing gears. A spring and a damper is inserted between the landing gear and the helicopter. This models the flexibility of the landing gear necessary for soft landing.

Motors for rotating the rotors and the door. These motors have a reference angle which changes with constant or varying velocity. A feedback torque is applied to the main and tail rotors according to the current and desired angular position and velocity:

$$t = k(q_{ref} - q) - d\dot{q};$$

where q is the current angle, q_{ref} is the reference angle, k and d are feedback coefficients, t is the resulting torque.

Cabin floor contacts. Four `MbsCutB` connectors are placed on the floor of the helicopter. These can be used in order to attach a load to the helicopter. If a load has the corresponding four `MbsCutB` connectors, some attraction force is set up between the points at the helicopter floor and the points at the load.

Ground contacts. There are four ground contact point specified for landing gears. There exist two points (front and rear) for each gear. A force is created as soon as a contact point is close enough to the ground (otherwise it is

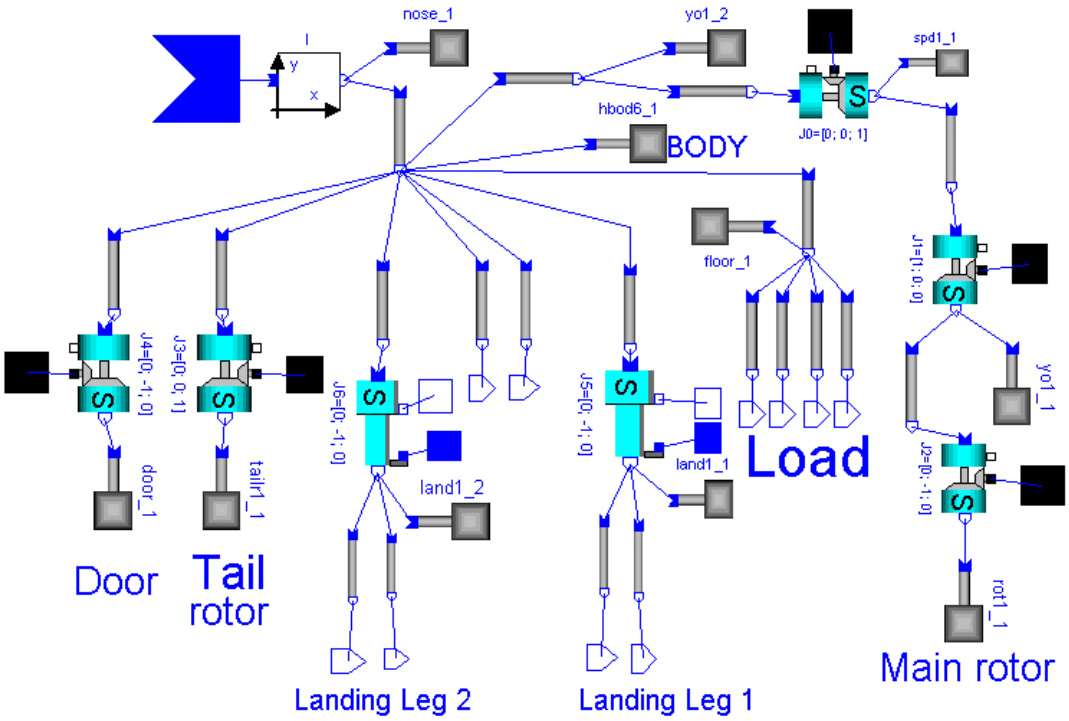


Figure 6: Modelica diagram of the mechanical part of the helicopter.

zero). The force is defined as a sum of horizontal friction force and vertical collision force.

$$
\begin{aligned}
F &= -(f_1 S_b' n_y) + (-f_2 S_b' n_x) + (-f_3 S_b' n_z) \\
f_1 &= c(s - s_0) + d v_y \\
f_2 &= c_{fric} v_x \\
f_3 &= c_{fric} v_z
\end{aligned}
$$

where

- S_b is the current rotation matrix of the contact connector on the landing gear.
- n_x, n_y, n_z are unit vectors.
- (v_x, v_y, v_z) is the current velocity of the contact connector relative to the ground.
- s is the current height of the landing gear contact point relative to the origin.
- s_0 is the height of the ground level relative to the origin. It is a constant

if a flat ground is modeled. Otherwise it is a value defined externally as function $s_0(x, z)$.

- c and d are spring and damping coefficients for collision with the ground. We use here a simplified force-based collision model. More advanced models have been designed later, see [3].
- c_{fric} is the coefficient of the friction constant preventing the helicopter from sliding after landing.

Just after a landing, if four such forces are applied to the landing gears of the helicopter, these forces compensate the gravity force and the helicopter stays on the ground. Guessing the right values of the coefficient is a difficult tasks. This has been done by trial and error. In order to achieve naturally looking movements during the landing: $c = 10^5, d = 10^3, c_{fric} = 15$.

There are some other components that can be easily attached to the model

Load. A load of appropriate size can be attached to the floor of the helicopter. The force between the load and the floor is modeled by stiff springs.

Ground level height. The height of the ground level can be defined by a table or by an external function which contains the geometry of the ground. This gives the possibility to model a landing failure if the landing point is on a slope. However if a "mountain" appears between two mission points, the control mechanism cannot prevent collision, since there is no built-in collision avoidance model.

Wind. Additional constant, random or customized forces and torques in arbitrary directions can be applied in order to destabilize the helicopter. This way wind and turbulent air flows can be modeled.

2.3 The Integration of the Helicopter Control System and the Helicopter Mechanical System

The mechanical system (the Modelica model) and the control system (the C or Ada model) have their own loops for finding solutions of arising mechanical and control equations. The control system uses an Euler method and fixed time steps (0.01 seconds).

The mathematically correct way of integration of Modelica model with sampled, discontinuous, external signal is using a `when sample(...)` statement. This statement triggers an event, updates the variables, and restarts integration each time an event occurs. Unfortunately, integration restart takes too much time, and computation time will be non-realistically large in this case. However, the control system is designed so that its output values change smoothly. Therefore we use external functions directly.

In order to synchronize the solvers special interface routines have been developed. One iteration of the control system loop might correspond to many iterations

of the mechanical system loop. The time in the control system increments by a fixed value at every solver step. The time values used in the solver steps in the mechanical system are dependent on the used solver. We found that an adaptive solver should be used, since other solvers lead to simulation failure (see below). Therefore, the time steps might vary. Simulation code generated by Dymola tool for the Modelica model of helicopter contains a number of solver iterations. During each iteration the external functions are called as follows:

- A solver step starts.
- Modelica simulation calls an external function that obtains the most recently saved values for forces and angles. These values were saved in the memory buffer during one of the previous steps.
- According to the equations of mechanics, the current helicopter acceleration, velocity and position is determined.
- The Modelica simulation calls an external function with the current helicopter position, the current time and other variables as parameters.
- If the modeled time in Modelica simulation code is larger than or equal the modeled time in the control system, the control system takes the parameters of the helicopter and evaluates new forces and angles that should be applied to the helicopter. New forces and angles are stored in the memory buffer. Also, the control system advances its time.

It is also possible to move the first call of the external function to the end. Since Modelica language specification does not explicitly define the order of evaluation of external functions, this order can be controlled indirectly by adding artificial dependencies between function calls. In this case the compiler attempts to reorder the sequence of the calls. For instance if two equations $a = f_1(b)$ and $c = f_2(a)$ are specified, and b is a known variable, f_1 will be called first. This is not a safe way, however, and should be used with care.

Additional uncertainty arises since functions in Modelica can be called many times at different solver iteration steps with "trial" time which can go forward and backward. In our simulations we made a simplification and used the largest reached time value and ignored all new simulated time values which were less than the largest one.

In the solver the external function for retrieving the data from the controller is called at every iteration of the solver, and different values of $time$ are given to this function. As soon as time becomes larger than the communication instant, e.g. t_1 at iteration s_1, the controller is called and returns values (forces and angles). These values either remain constant until the iteration s_2, or smoothly change (using one of smoothing approaches) as described below.

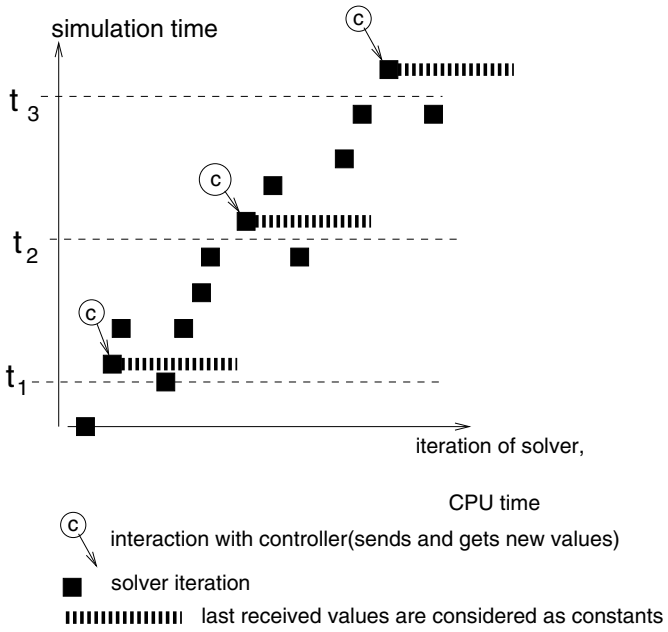


Figure 7: An approach to sampling values at regular simulated time interval from external source. The solver iterations are represented as small black boxes. As soon as time corresponding to the iteration is larger than t_1 (the same applies to t_2, t_3, etc.), a value is read from the external source.

The solvers of ordinary differential equations, in particular those used with Modelica simulation code, require that all external functions during continuous integration are differentiable up to n-th order (where $n \geq 1$ is the order of numerical integration method used in the solver).

Therefore all input data should be smoothed using one or several splines connected together. Three variants of smoothing were tried: initial discontinuous signal (Figure 8(a)), continuous (Figure 8(b)), and signal with continuous first derivative (Figure 8(c)). The last one is constructed by gluing together two second order polynomials and one constant interval. Despite of these variations the numerical results and simulation performance were roughly the same in all three cases. Therefore we made a conclusion that discontinuity of the input data does not affect the solver substantially in our model.

We have a hypothesis regarding an accurate approach to the co-simulation of the control system and the mechanical system. If the control system uses a fixed-step explicit equation solver, the mechanical system should use the same, fixed-step explicit equation solver, with probably smaller step size. In practice, however, the mechanical system is quite complicated, therefore the time step for it should be extremely small in order to use an explicit method and obtain a solution that converges. Computation time will be non-realistically large in this case.

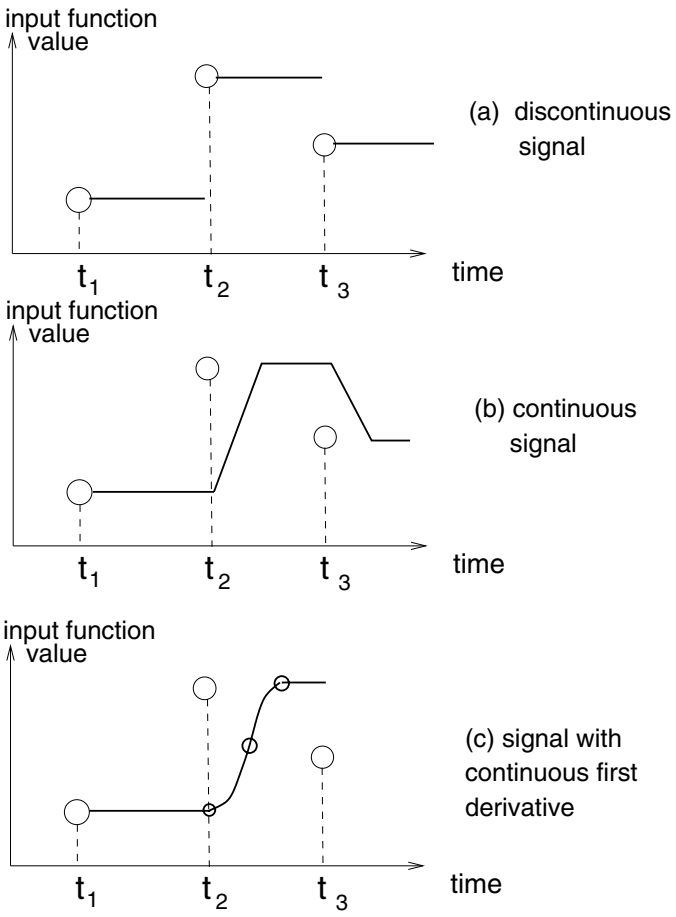


Figure 8: Three variants of smoothing data in external function. The circles represent the values obtained at sampling time t_1, t_2, t_3. Since an external function cannot be discontinuous, one of three smoothing algorithms is performed.

2.4 Helicopter Visualization

The helicopter visualization is obtained by the MVIS tool. The following properties were particularly useful for visualization:

The mission lines. The points used in the flight specifications in the Flight Command Language (mission points) are shown as boxes in space, connected by straight lines. Helicopter normally should fly along the lines and turn near the boxes. It is easy to see cases when the flight deviates from the mission lines.

Helicopter trajectory. The trajectory of some part of helicopter is shown on the screen and compared with the mission line. If the main part of the helicopter is observed, the line describes the general path of the flight (see Figure 10).

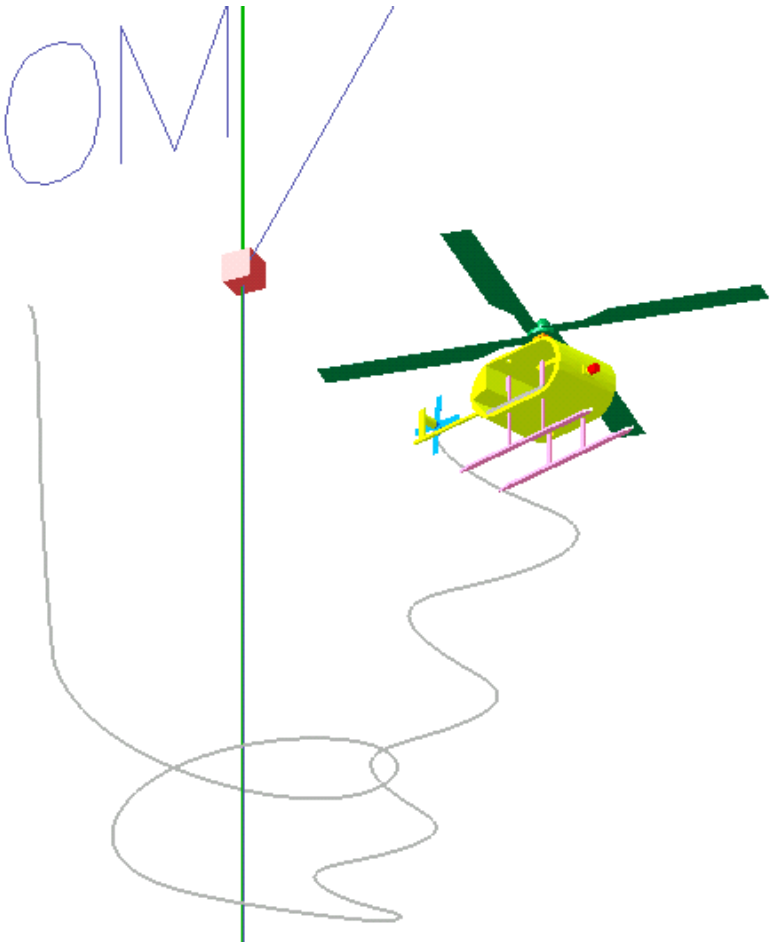


Figure 9: Trajectory of the tail rotor of the helicopter.

If the tail rotor position is observed, stability of helicopter rotation can be visually assessed (see Figure 9).

Shadow. The shadow from the helicopter is a projection of its parts on the ground plane. It helps to determine exact position of the helicopter in the horizontal direction.

2.5 Conclusions on Helicopter Simulation and Visualization

The helicopter modeling, simulation and visualization project was the first large model obtained by the SolidWorks–to–Modelica compiler. It turned out to be possible to change the design of the helicopter in the CAD tool and rapidly produce a corresponding Modelica model.

The generated model can be conveniently connected to other, non-generated components, such as landing, ground model and load model. This added some

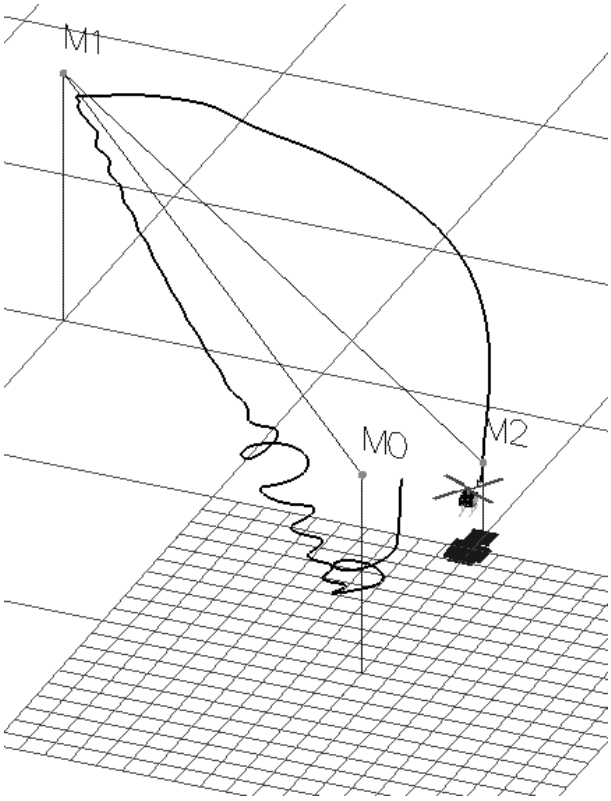


Figure 10: The whole trajectory compared to the mission path.

more complexity to the set of equations to be solved.

The mechanical model of the helicopter was initially considered separately from the control system. It is necessary in order to validate position of the center of mass with respect to the rotors, as well as to test functionality independent of the controller (e.g. landing). The model behaves as expected, and the solver computes the movements quite fast (10 seconds of flight are modeled using 5 seconds of the CPU time[1])

When the model is connected to the external control system, various problems arise, both instability and performance problems. The trajectory of the flight, the success of mission and time it takes to perform the simulation depends on many aspects of the model and its simulation.

In many cases the mission was performed successfully, i.e. all the points were visited, flight was stable, and the helicopter landed in normal vertical position at the landing point. Simulation time have been close to the real time (30-50 seconds of CPU time are necessary for simulation of 30 seconds of flight).

The simulation currently fails in the following three cases:

[1]Here and in the further experiments we were using a 266 MHz Pentium II-based computer without graphic card, with 128 MB RAM.

- The control system loses control over the horizontal position of the helicopter and cannot steer it to the target position. The helicopter flies away from the target.
- The control system loses control over the vertical or/and rotational position of the helicopter. It chaotically applies various forces and angles, but the helicopter falls down anyway.
- The adaptive solver takes too small integration steps so that simulation takes 30 times more than the real time, i.e. 1 minute of flight is modeled using 30 minutes of CPU time.

The following factors affect the failures:

Control system simulation step size. The step size in the original model was 0.01 sec. Variations of this value (0.02, 0.03) sometimes caused failures, sometimes removed the failures.

Mechanical system solver step size. Larger steps typically causes less stability and faster simulation.

Mechanical system solver precision. Larger precision threshold caused less stability and faster simulation.

Mechanical coefficients. A number of unknown coefficients were guessed, such as density of material, speed of rotation of the rotors, weight of the rotors, etc. Small changes of these coefficients affect the failures.

Mission properties. Sharp corners of the mission line and short distances between the points caused failures. Actually all the failures occurred in the case of very short flights (less than 30 seconds), and typically due to small distance between mission points (less than 20 m). All experiments with larger missions were much more successful, however they were no so spectacular for visualization and not so convenient for validation.

As a result of the simulation some errors in the control system were identified. Some coefficients used in this system were adjusted. Therefore the goal of the experiments was partially achieved. However the mechanical model is certainly too simplified. Currently the interaction between the helicopter and the air is modeled by a single lifting force. More accurate models should be created including aerodynamic properties of rotor blades.

3 Robot Modeling

A study of service robots based on modeling and simulation was initiated in the framework of the WITAS [14] project. The major objective of the project was

to predict tasks that a service robot-manipulator cooperating with unmanned autonomous aircraft will be able to solve in the future. The service tasks that the robot will perform in the future can, for instance be the following:

- Refueling the helicopter.
- Loading and unloading of cargo.
- Exchange of components of the helicopter.
- Minor repairs.

The initial plans contained a study of cooperation and interaction between a service robot and a helicopter. However in a typical situation the task of the robot can be described as "to take some load from one container and to place it to another container". A helicopter can serve as one of these containers. The position and rotation of the containers are known to the control system of the robot. Finally the robot modeling problem was formulated the following way:

- Input:
  - Given positions of containers (later also slots) and their identifiers, such as `c1` or `c2`.
  - Given a sequence Robot Command Language (RCL) commands (missions), such as `MOVE(c1,c2)`.
  - Given a virtual robot constructed in SolidWorks[13].
- Output:
  - Torques applied by the motors in order to perform the mission, for instance, to grab a load from container `c1` and move it to container `c2`.
  - The movement trajectory of the robot and all its joints when the mission is performed.

3.1 Mechanical Part of the Robot and the Load Model.

The mechanical part of the robot model was initially designed in SolidWorks and then automatically translated to Modelica. The model contains seven rigid bodies with six rotational joints between them. There is a motor controlled by the control system at each joint. In addition, the robot is placed on a mobile platform. The mobile platform is located on the ground (floor) and has two translational joints. All the joints are steered by motors, which are controlled by the controller. The platform can reach any point on the plane. From the platform the robot can reach any point in the volume $2 \times 2 \times 1\ m$ above the center of the platform.

The robot can interact with the load. Loads represent spare parts or cargo. The load is a cube of size $0.1\ m$ in each dimension. There are three connectors on

the grip of the robot, which correspond to three connectors on the load. When the grabbing mode is turned on, three stiff springs-and-dampers are activated between the grip and the load. After some interaction the robot keeps the load firmly in its manipulator.

The load has a simplified model of collision detection with the ground. When the load reaches the ground, and some of the vertices of the load occur under the ground a stiff spring-and-damper is activated and pushes the vertex away from the ground. After some interaction the load rests still on the ground.

3.2 Environment Model

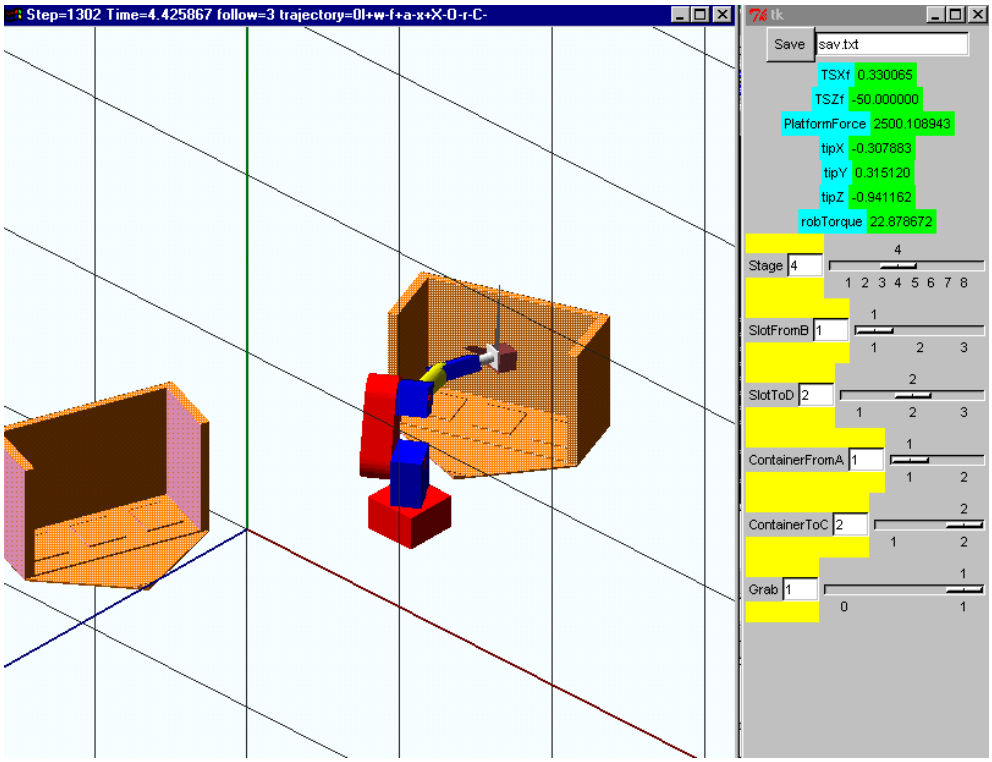


Figure 11: A virtual environment includes a robot, load (a small cube) and two containers.

The operating environment consists of two or more containers (see Figure 11). Containers represent helicopters, or storehouses with spare parts or cargo. Each container has an identifier and it contains three or more slots which have their identifiers. Containers are constructed as open boxes (approximately $0.5 \times 0.5 \times 0.5$ m) with the top and front wall missing. The slots have fixed positions within the containers. The load can be placed at the slots or removed from them. We assume that all the containers in the simulated world are standardized, i.e. the slots

have fixed position. The position and rotation of container is known to the control system.

3.3 Scenario for Load Movement

Step	Platform	Manipulator	Grabbing	Terminate if
0	origin	home	off	immediately
1	source	home	off	$D_{platf} < \varepsilon$
2	source	source	off	$D_{manip} < \varepsilon$
3	source	source	on	$D_{grab} < \varepsilon$
4	target	home	on	$D_{platf} < \varepsilon$
5	target	target	on	$D_{manip} < \varepsilon$
6	target	target	off	immediately
7	origin	home	off	$D_{platf} < \varepsilon$

Table 1: Steps of scenario for moving a load from source to target.

The scenario for load movement is shown in the Table 1. The controller switches from one step to another when certain conditions become true. At each step specific target values for the actuators are specified. The platform motors can move on the plane between the origin position, source and target container. Every container has a reference point where the platform should be located. The actuators for the platform compute force necessary to move the platform in needed direction.

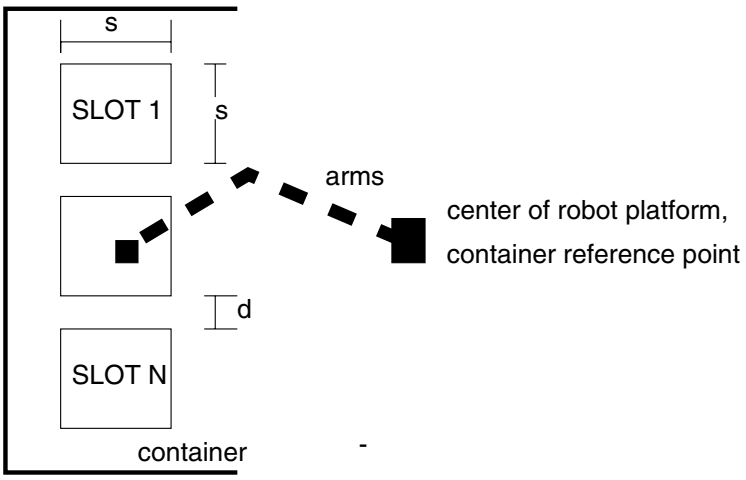


Figure 12: Container with three slots. View from above.

The manipulator can be in three positions: home, source slot and target slot position. In the home position the grip is at a position about 1 m above the ground. The slot positions are shown in Figure 12. The slot position for each slot (relative

to the container reference point) is known from the container geometry. However since the robot has six revolute joints, the inverse geometry problem should be solved in order to move the grip to the necessary position and rotation. We discuss our solution to inverse geometry problem in Section 3.4.

The grab flag turns on and the stiff spring between the manipulator grip and the load, modeling a magnet-based manipulator.

The termination conditions are defined by functions D_{platf}, D_{manip} and D_{grab}. If the difference between required value and actual state (angles of the revolute joints, position of the prismatic joint, length of the spring) is close to zero, and its derivative is close to zero, we consider that the required value is reached. It appeared to be difficult to derive appropriate values for ε for each case. Therefore the termination condition is computed "visually" by the user, and he or she can interactively control the simulation by giving an order to shift to the next step of the plan.

The plan is implemented as an array in Modelica, and the value of variable for indexing the array is obtained via a graphical user interface (GUI) input function.

In order to get the new value from GUI, a sampling method is used:

```
when sample (0, 0.1) then
 step=infun(...)
end when;
```

Each 0.1 seconds the integration stops, new value from GUI is obtained, and integration restarts with possibly different target value for the new step, according to the Table 1. If the value from the GUI is still old, no variable change occur, and therefore the integration restart does not take much time from the solver.

3.4 The Inverse Geometry Problem

Objects in the simulated world can be easily located using three Cartesian coordinates, and their rotation is defined by additional three angles. When robot links are connected by revolute joints, the position and rotation of the grip of the robot is defined by six angles. The inverse geometry problem in our case is finding such angles of the robot joints, so that the grip is located at the required position (in Cartesian coordinates) and the required rotation.

3.4.1 The Inverse Robot in 2D

The 2D case is illustrated in Figures 13 and 14. In Figure 13 a manipulator (direct robot) is shown. It has two motors, two bars (links) and the tip. The position of the tip (X, Y) is defined by the angles A and B. Therefore we call such model a *direct* robot.

The inverse geometry problem is to determine values for A and B from the given X and Y. For this purpose the model depicted in the Figure 14 has been constructed. The angles (joints) A and B are free now. In addition a revolute joint

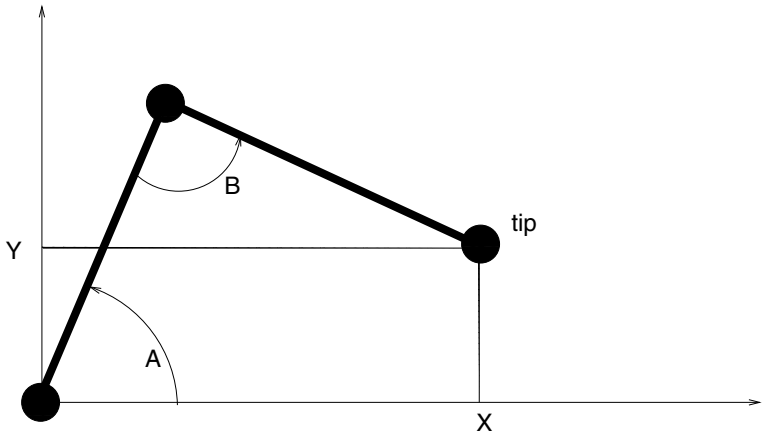


Figure 13: Direct kinematics model for two links with two revolute joints.

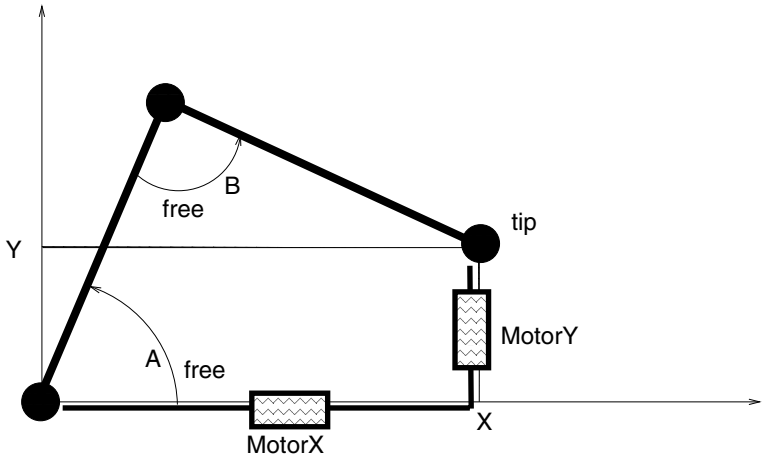


Figure 14: Inverse kinematics model for two links with two revolute joints.

appears at the tip. Two prismatic bars with actuators are added to the model. One bar moves along the X axis. Another bar is attached to the first one and moves along the Y axis. The positions of these bars define (X, Y)–position of the robot tip. The angles at the revolute joints A and B can be obtained. This model is called an *inverse* robot.

3.4.2 Alternative Solutions

Generally, the inverse geometry problem can be solved by trigonometric computations in 3D, and requires complete information about the kinematics of all the links (i.e. the positions of the joints and their types). The inverse geometry problem is a special case of the inverse kinematics problem where movement of a point in Cartesian coordinates should be mapped to the change of angles in revolute joints.

The inverse geometry problem is solved by finding a Jakobian, which describes the relations between the changes. It might appear that the solution is missing or that the process of solution reaches a singularity point.

3.4.3 The Inverse Robot in 3D.

In our approach we use a kinematic loop in order to obtain necessary values for the inverse geometry. For this purpose from original (direct) robot model a corresponding model of the inverse robot is created. This model has the same geometry as the original robot, but is controlled in a different way.

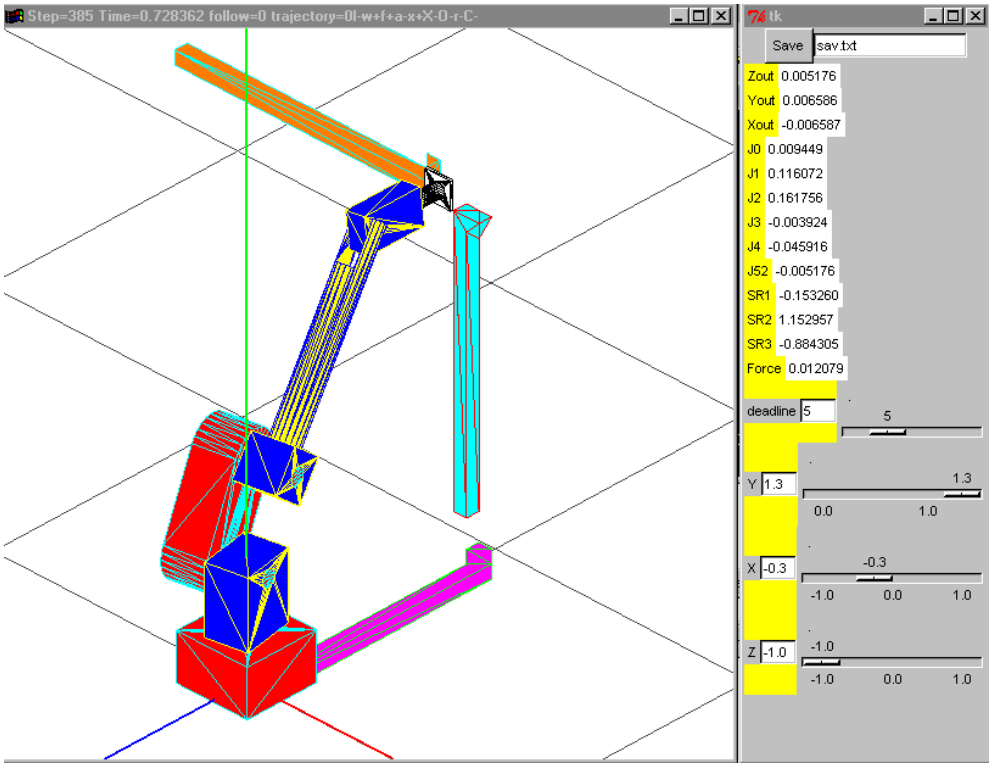


Figure 15: The inverse robot with additional bars has a kinematic loop. Required position of the tip is set up by the three scale bars (for X, Y, and Z).

The direct robot has six revolute joints with state variables. The inverse robot (Figure 15) has five revolute joints without state variables and one revolute joint `RevoluteCut3D`[2]. instead. These joints connect the links from the base to the grip. In addition three prismatic joints with state variables are added, and their movement is visualized by three bars. The reference position of the prismatic joints

[2]The model `RevoluteCut3D` is a model of revolute joint with one rotational degree of freedom. In the MBS library[2, 6] when kinematic loops occur such joint model should be used instead of usual revolute joint model `RevoluteS`

(scale bars for X, Y, and Z position) can be controlled interactively, just like the motors *MotorX* and *MotorY* in Figure 14.

The inverse geometry problem is solved in the following way. Initially all the joints stay in the home position. The user interactively specifies reference positions for the three orthogonal, consequently connected bars. Actuators apply forces to the bars and these move to the specified position. Since the inverse robot grip is attached to the last bar, all the joints of the robot move correspondingly. Finally the grip reaches the target and the angles ($\alpha_1, ..., \alpha_6$) at all the revolute joints of the robot are stored for future use.

Now we don't need the inverse robot anymore. We start simulation of the direct robot, which has motors attached to the revolute joints

When the angles $\alpha_1, ..., \alpha_6$ are used as reference angles for the joints of the direct robot, the grip of this robot will be gradually moved to the required position. The same angles $\alpha_1, ..., \alpha_6$ can potentially be applied to the joints of a physical robot.

The inverse kinematics problem is solved just as a slight extension of the method described above. If the required position of the three prismatic joints is given as a function of time, the corresponding values of the angles for certain time span can be saved, and used later in order to directly steer the direct robot.

3.5 Conclusions on Robot Simulation

A framework for mechanical simulation of manipulators has been developed. We considered a specific manipulator (with six rotational degrees of freedom, on a mobile platform), but the method can also be applied to other virtual devices. We show that a simplified deliberative and reactive layer of control for a robot can be designed using Modelica. Alternatively to table-driven plan, the robot operation plan can be composed also as a set connected primitive plan steps.

Currently the plan is hard-coded in Modelica model. As alternative, it can be obtained from the interactive environment during simulation, or read from an input file. This way programmable virtual robots can be modeled.

If inverse and direct robot are integrated in the same simulation it is possible to give robot commands which include Cartesian coordinates. It greatly simplifies construction of the movement plan.

The disadvantages with our approach are that it can potentially be rather slow, and can, just as other methods, reach a singularity point. Also it is difficult to handle with cases when the required position is out of reachable space for the robot.

The advantage is that the same model (or model with small and well-defined modifications) can be used both for direct and inverse kinematics.

4 Acknowledgments

The Modelica definition has been developed by the Eurosim Technical Committee 1 (Modelica Design Group)[7] under the leadership of Hilding Elmqvist (Dynasim AB, Lund, Sweden) and Martin Otter (DLR, Germany). The Dymola tool has been designed by Dynasim AB. The Multibody Simulation Library has been developed by Martin Otter.

This work has been supported by the Wallenberg foundation as part of the WITAS project [14].

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] Hilding Elmqvist, Dag Brück, Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual, Version 4.0*, from Dynasim AB, Research Park Ideon, Lund, Sweden, http://www.dynasim.se

[3] Vadim Engelson, *Integration of Collision Detection with Multibody System Library in Modelica.* Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 010. Available at: http://www.ep.liu.se/ea/cis/2000/010/

[4] Vadim Engelson, *Tools for Design, Interactive Simulation, and Visualization for Dynamic Analysis of Mechanical Models*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 007. Available at: http://www.ep.liu.se/ea/cis/2000/007/

[5] Håkan Larsson, *Translation of 3D CAD Models to Modelica*, Master Thesis, LiTH-IDA-Ex-99/30, IDA, Linköping Univ., Sweden, March 1999.

[6] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[7] Modelica Design Group, *Modelica WWW site*, http://www.modelica.org

[8] Johan Parmar, *Modeling an Autonomous Helicopters and its Maintenance using Modelica*, Master Thesis, LITH-IDA-Ex-99/63, IDA, Linköping Univ., Sweden.

[9] PELAB Group, *Modelica activities in PELAB*, Linköping University, http://www.ida.liu.se/~pelab/modelica

[10] Erik Skarman. *An aircraft model*, Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 4 (1999): nr 013. Available at: http://www.ep.liu.se/ea/cis/1999/013/

[11] Erik Skarman. *An helicopter model*, Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 4 (1999): nr 014. Available at: http://www.ep.liu.se/ea/cis/1999/014/

[12] Erik Skarman. *A helicopter control system*, Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 4 (1999): nr 015. Available at: http://www.ep.liu.se/ea/cis/1999/015/

[13] SolidWorks Corporation, *SolidWorks*, http://www.solidworks.com

[14] WITAS group, *The Wallenberg Laboratory for Research on Information Technology and Autonomous Systems*, Linköping University, http://www.ida.liu.se/ext/witas

Paper 7

An Environment for Design, Simulation and Interactive Visualization for CAD Models in Modelica*

Vadim Engelson, Håkan Larsson, Peter Fritzson
Linköping University, Sweden †

Abstract

The complexity of mechanical and multi-domain simulation models is rapidly increasing. Therefore new methods and standards are needed for model design. A new language, Modelica, has been proposed by an international design committee as a standard, object-oriented, equation-based language suitable for description of the dynamics of systems containing mechanical, electrical, chemical and other types of components. However, it is complicated to describe the system models in textual form whereas CAD systems are convenient tools for this purpose. We have designed an environment that supports the translation from CAD models to standard Modelica notation. This notation is then used for simulation and visualization. Assembly information is extracted from the CAD models, from which a Modelica model is generated. By solving equations expressed in Modelica, the system is simulated. A 3D visualization tool based on OpenGL visualizes expected and actual model behavior, as well as additional parameters. The environment has been applied for robot and flight simulation.

Keywords: CAD, Mechanical modeling, Simulation, Animation, Interactive Visualization, Modeling languages, Modelica, Bridge construction, Indoor climate. .

1 Background

The use of computer simulation in industry and construction is rapidly increasing. Simulation is typically used to optimize product properties and to reduce product development cost and time to market. 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 mechanical, electric, hydraulic, thermodynamic, and control system components.

A new language called Modelica [10, 4, 11, 5] for hierarchical physical modeling is being developed through an international effort. Modelica 1.1 was announced in December 1998. 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 of-

*Published in *Proceedings of 1999 IEEE International Conference on Information Visualization*, IEEE Computer Society, 14-16 July 1999, London, pp. 188-193, ISBN 0-7695-0210-5. Submitted for publication in the International Journal of Computer Integrated Construction, special issue on Visualization in Architecture, Engineering and Construction.

†The contact address: Vadim Engelson, IDA, Linköping University, S-58183, Sweden; telephone: +46 13281979, fax: +46 13284499, e-mail: vaden@ida.liu.se

fers three important advances: 1) non-causal modeling based on differential and algebraic equations; 2) multidomain 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 class construct.

Modelica is a language for *dynamic* simulation. In particular, mechanisms (such as construction tools, robots, vehicles) and constructs under dynamic load (such as hanging bridges [20]) have been modeled, and interactively simulated. Thermodynamics applications are modeled too; in particular, models for indoor climate and energy simulations were developed in NMF which is similar to Modelica ([21]). This application inputs a building map designed in a CAD tool and generates equations for indoor climate simulations.

Future application areas can be e.g. modeling of the dynamics of buildings during earthquakes.

Modelica is a standard notation which is used for standard domain libraries and for applications that use these libraries. Tools and environments are built to comply with this standard.

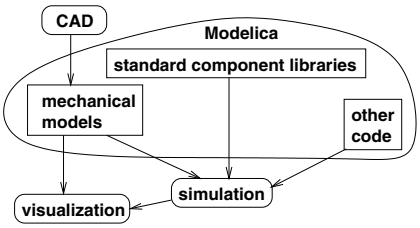


Figure 1: **Structure of the integrated environment.**

The structure of the environment that leads the user from interactive design to interactive visualization is shown in Figure 1. Section 2 gives an introduction to the Modelica language and its standard libraries for electrical and mechanical modeling. Section 3 describes the CAD tool we use and translation from CAD models to the standard Modelica notation. Sections 4 and 5 describe simulation and visualization issues.

2 The Modelica Language

2.1 Simple Electric Circuit

As an introduction to Modelica we will present a model of a simple electrical circuit. Our goal is to describe features of universal Modelica standard notation, which can be used in applications in various domains (such as electrical, mechanical or chemical). A detailed description of this example can be found in [5, 10]. The system can be broken into a set of connected electrical standard components.

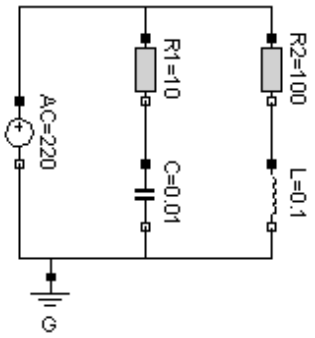


Figure 2: **Sample circuit structure in Modelica graphical notation.**

Assume that the sample model (Figure 2) consists of a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of such components are available in Modelica standard class libraries for electrical components.

A declaration like the one below specifies `R1` to be an instance (i.e. an object) of standard library class

Resistor and sets the default value of the resistance, R, to 10 (i.e. R1.R is 10).

```
Resistor R1(R=10);
```

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);
connect(R1.n,C.p);
connect(C.n,AC.n);
connect(R1.p,R2.p);
connect(R2.n,L.p);
connect(L.n,C.n);
connect(AC.n,G.p);
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.

The components (Resistor, Capacitor, etc.) are subclasses derived from the class TwoPin which in turn contains two Pin objects:

```
class Voltage = Real;
class Current = Real;

connector Pin
 Voltage  v;
 flow Current i;
end Pin;

class TwoPin
 Pin p, n;//positive and negative pin
 Voltage v;
 Current i;
equation
 v = p.v - n.v;//voltage difference
 p.i = - n.i;//current going inside via two pins
 i = p.i;
end TwoPin;
```

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 an equality for v and flow balance for i, namely: Pin1.v = Pin2.v and Pin1.i + Pin2.i = 0 .

Modelica and its standard libraries for electrical models provide short, clear, extensible and concise notation for such models.

During system simulation the variables i and v evolve as functions of time. The solver of algebraic and differential equations computes the values of all variables in the model for all simulation time steps.

2.2 Implementation of Model Simulation

Instances of classes in a model, including equations, are translated into a flat set of equations, constants and variables. After flattening, all the equations are sorted in order of data dependence.

The symbolic solver/simplifier performs a number of algebraic transformations to simplify the dependencies between the variables. It can also solve a system of differential equations if it has a symbolic solution. Finally, C code is generated which is linked with a numeric solver. As the result a function of time (t), e.g. R2.v(t) can be computed for a time interval [t_0,t_1] and displayed as a graph or saved in a

file. This data presentation is the final result of system simulation.

2.3 Mechanical System Modeling in Modelica

To facilitate mechanical system modeling there exists a standard Modelica class library for modeling multi body mechanical systems (MBS) [9, 13], i.e., systems of rigid bodies connected to each other with certain degrees of freedom.

A model that uses MBS consists of an inertial system (instance of class `Inertial`), joints (instances of classes `RevoluteS` or `PrismaticS`), massless bars (class `Bar`) and bodies (class `Body`) with mass. The objects are connected together with the Modelica `connect` statement.

The `Inertial` object defines the global coordinate system and the gravitational force. All other objects are in some way connected to this object, either directly or through other objects.

The use of the MBS library can be represented by a double pendulum example (see Figure 3):

```
class Pendulum
 Real L = 0.5;
 Inertial I;
 Body P1(rCM={L/2,0,0});
 Body P2(rCM={L/2,0,0});
 RevoluteS rev1(n={0, 0, 1});
 RevoluteS rev2(n={0, 0, 1});
 Bar arm(r={L, 0, 0});
equation
 connect(I.b,    rev1.a);
 connect(rev1.b, P2.a);
 connect(rev1.b, arm.a);
 connect(arm.b,  rev2.a);
 connect(rev2.b, P1.a);
end Pendulum;
```

The instances of Modelica classes (such as `P1`, `P2`, `rev1` etc.) have attributes that can be modified. For instance, `Body` has attributes that define mass, inertia tensor and location of center of mass relative to the local coordinate system (`rCM`). Instances of `RevoluteS` (revolute joint) have an attribute `n` that defines direction of rotation axis. Instances of `PrismaticS` (prismatic joint) have an attribute that specifies direction of allowed translation. For a `Bar` the coordinates of its end are specified as `r`.

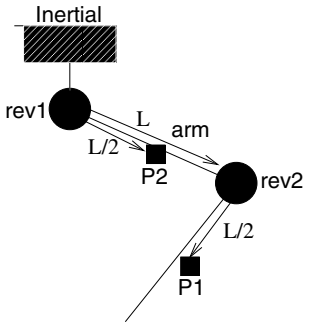


Figure 3: Logical presentation of the pendulum.

The classes of the MBS library have connectors called `a` and `b`, of type `MBSCut`. A **`connect`** statement usually connects two `MBSCuts` attached to two different instances. This specifies equality of rotation, position, velocity and acceleration. It also specifies that there is a balance for force and torque between the connectors.

Every MBS class contains differential equations that specify relations between rotation, positions, and forces at its connectors `a` and `b`.

During system simulation all equalities, balances and differential equations are solved and the values of all the numeric items are computed for each time step.

There are 70 other classes in the MBS library (for bodies and joints) and 30 classes in drive train library (for motors and other mechanical elements).

3 CAD Tools

In order to simplify design of mechanical Modelica models, CAD tools can be utilized. The system used in our project is SolidWorks[16].

SolidWorks uses the concept of parts and assemblies. Each solid component (a rigid body) is modeled as a separate *part* document.

In the *assembly* document these parts are put together to form a complete model. Each part model can also occur more than once in the assembly.

The *assembly* document defines the mobility between the parts of an assembly. Between two parts, several so called *mates* are connected, each adding some constraint to the mobility between the parts.

A part consist of entities, such as planes, faces, edges, axes and points. A mate connects two entities from different parts. There exist several mate types. The most typical are *coincident* (all the points of one entity are inside another entity) or *parallel* (it keeps entities parallel to each other).

Two parts can be connected by one, two or more mates. Some combinations of mates are valid, some are not. Invalid combinations of mates are rejected by SolidWorks automatically. In [7] we analyze valid sets of mates between pairs of parts and translate them to corresponding sets of Modelica joints. There are similar concepts for parts, assemblies and mates in other 3D CAD tools, e.g. [19].

3.1 Example

The example (Figure 4(a)) describes a fragment of the pendulum model. The part `P1` has front face (`f1`) and upper edge (`e1`). The part `P2` has the front face (`f2`) and bottom edge (`e2`). There is a mate M_1 that specifies that planes of `f1` and `f2` are coincident. The mate M_2 specifies that the edges `e1` and `e2` are coincident. The SolidWorks system analyzes the mates and adjusts positions of the parts (Figure 4(b)).

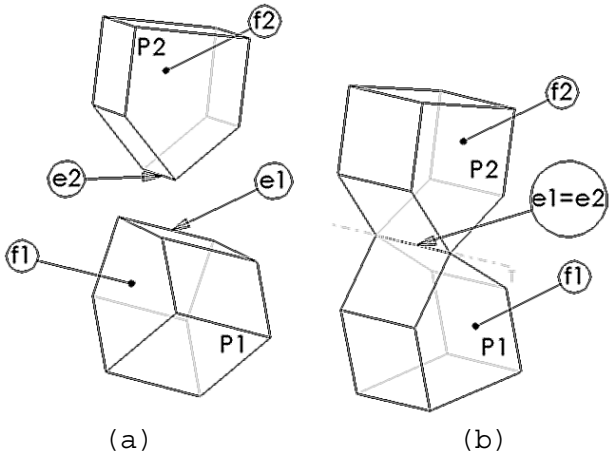


Figure 4: **Parts and their mates specification before (a) and after (b) adjustment according to the mates.**

The system automatically rejects invalid mate combinations. Our translator [7] finds that there is a joint with one rotational degree of freedom between the parts `P1` and `P2`, and calculates the position and orientation of the rotation axis. This pair of mates corresponds to an instance of class `RevoluteS` from Modelica MBS library with attached `Body` instance.

3.2 Modelica Model

Each SolidWorks assembly consists of a set of parts, and it stores a set of mates. All these are validated and translated to a set of Modelica MBS class instances and appropriate *connections* between them. Mass, position of center of mass and inertia tensor are extracted from the corresponding part documents by SolidWorks. The result of a Modelica model simulation is position, rotation, velocity, acceleration and other physical properties of each `Body` as functions of time during the simulated time period.

4 Translation and simulation

Figure 5 represents components of the environment needed for visualization. Our translator from SolidWorks to Modelica takes information about the mates and produces a corresponding set of Modelica class instances with connections between them. The mass and inertia tensors for each part are computed by SolidWorks. These are extracted and used in a Modelica model. Geometry information is saved in a separate STL [17] file for each part.

By default the gravity force is applied to the mechanical model. Usually this is not enough for simulation. All external forces that are applied to the bodies, as well as motor forces that are applied to revolute and prismatic joints should be specified. This is done outside the SolidWorks model by adding code for new class instances to the Modelica model.

A control subsystem that controls the forces according to a certain plan (mission) can be written in Modelica. If necessary, external code in C can be added to the model.

When a Modelica model is simulated, the position, orientation, velocity and acceleration for each part (`Body` instance) is computed. For Modelica simulation we use the Dymola tool with Modelica support[3].

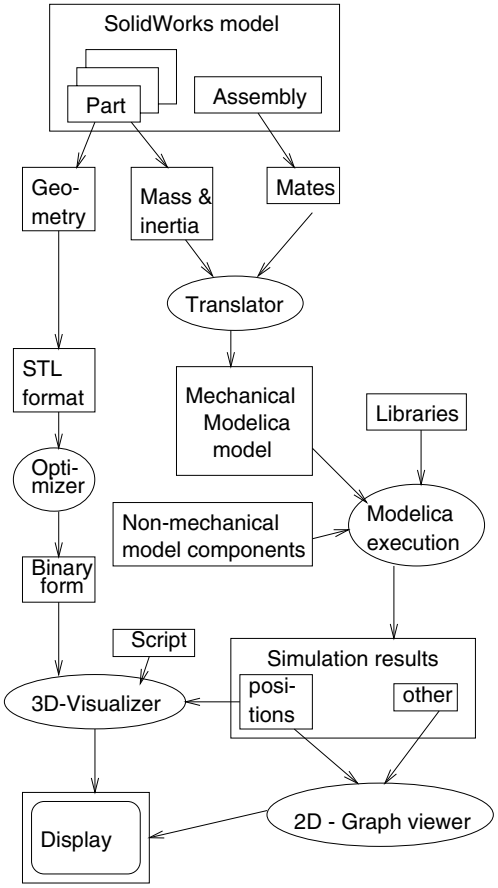


Figure 5: **The path from SolidWorks model to dynamic system visualization.**

5 Visualization

The integrated environment includes a visualizer that provides online dynamic display of the assembly (during simulation) or offline (based on saved state information for each time step).

The STL format[17] is a very simple format suitable for visualization. All surfaces are divided into triangles and the the coordinates of the triangle vertices, as well as normal vectors of the triangles, are listed in the STL-file.

The visualizer loads the corresponding STL file for each part and optimizes it for rendering. After that, rendering is performed by OpenGL [8] library functions. During optimization the vertices positioned very close are merged together. Optimized STL code is stored in a binary file for future use.

The user of the visualizer can alternatively utilize the pop-up menu system, keyboard shortcuts, or a command string in order to control various options. We found that the following options (that can be turned on and off) should be available, and we have

implemented them.

— Rotating the camera in 2 degrees of freedom (DOF), moving the camera in 3 DOF, zooming in and out.

— Using perspective and orthographic projection.

— Display of a part as a wire-frame, filled with certain color, using a lighting model with certain light sources, or hiding a certain part.

— Display of an application-specific landscape, for instance, road for car simulation, or a runway surrounded by hilly landscape for flight simulation. Such a landscape can be created as a SolidWorks assembly that does not move, or directly in C using OpenGL.

— Display of planned trajectory (mission) of some parts.

— Display of actual trajectory of some parts.

— Display of origin and coordinate axes (local for bodies and global one) as well as grid lines.

— Display pseudo-shadow. The pseudo-shadow is not dependent on light at all. We found that for flight simulation it is convenient to display a projection of vehicle and trajectory on the ground plane.

— Synchronization of animation with machine clock.

— Starting, stopping, continuing animation, stepping forward and backward.

— Targeting camera center of view on a particular part, so that camera follows the part all the time.

— Rotating camera together with the target part.

The integrated system has been used for modeling industry robot behavior as well as helicopter flight simulation (Figure 6). We have designed a robot and helicopter control models and performed design optimization for the control system [18, 15]. The helicopter is designed in SolidWorks and consists of 10 parts, 4 revolute and 2 prismatic joints.

It is possible to export the visualization to 3D Studio MAX [1] (to create and save movies) and MultiGen[12] on SGI (to design Virtual Reality applications).

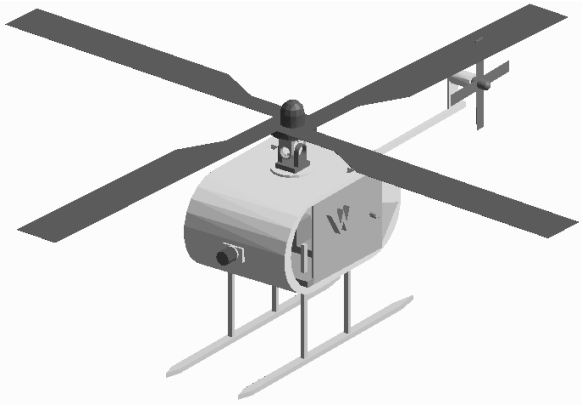


Figure 6: **Helicopter dynamic visualization.**

6 Related Work

Our approach has similarities to the Working Model 3D tool [19]. This tool permits construction of a mechanical model with joints (free and motor-controlled), springs, dampers and ropes. It also has built-in collision detection options. Working Model 3D can import assemblies from SolidWorks.

However, Working Model 3D is a *closed* system and user-defined control code can be used there in a very limited way. On the contrary in Modelica we specify arbitrary control algorithms for mechanical and mechatronic models. Working Model 3D is limited to certain types of mechanical systems, whereas Modelica supports general multi-domain modeling.

Similar mechanical simulation features are available in some other advanced CAD tools.

7 Future work

7.1 Using STEP/EXPRESS for Contact Computation

STEP (Standard for the Exchange of Product data)[6, 14] is an international standard for data exchange. It includes a language called *EXPRESS*, that can be used for exchanging advanced geometric information between CAD/CAM systems. This is an advanced file format where the geometry of the solids is represented in a more mathematical way than in STL. This format could be useful when calculating points of contact between parts or if a representation of the part geometry on a closed form was to be included in the Modelica model.

7.2 True Multidomain Applications

Modelica is suitable for multiple application domains. Components from several domains (mechanical, electrical, hydraulic simulation) can be used within the same Modelica model. For instance, electrical components can be combined with mechanical components in the model. The electrical parts can be written by hand, or designed using a block-oriented editor, or extracted from an electric CAD system. The generated Modelica code for electrical components in combination with a mechanical design tool produces a multidomain Modelica model.

8 Conclusions

An integrated environment for simulation of multidomain models has been implemented using Modelica as a standard model representation. The user can work with arbitrary SolidWorks models, extend the corresponding Modelica model in various ways, and analyze the simulation results in high performance visualization environment. A complex model such as a pendulum consisting of seven bars can be created in ten minutes. It is simulated 5-7 times faster than corresponding model in Working Model 3D [19]. The helicopter and industrial robot models [18, 15] were successfully designed and simulated.

9 Acknowledgments

The Modelica definition has been developed by the Eurosim Modelica technical committee [10] under the leadership of Hilding Elmqvist (Dynasim AB, Lund, Sweden). The work has been supported by the Wallenberg foundation as part of the WITAS project [18].

References

[1] *3D Studio Max*, Autodesk Inc., http://www.ktx.com

[2] *Cult 3D Home Page*, Cycore AB, http://www.cykar.se

[3] *Dymola Home Page*, Dynasim AB, http://www.dynasim.se

[4] H. Elmqvist, S. E. Mattsson. Modelica – The Next Generation Modeling Language – An International Design Effort. In *Proceedings of First World Congress of System Simulation*, Singapore, September 1–3 1997.

[5] P. Fritzson, V. Engelson, Modelica – A Unified Object-Oriented Language for System Modeling and Simulation, In *Proc. of Eur. Conf. on Object-Oriented Progr. (ECOOP98)*, Brussels, July 20–24, 1998.

[6] *ISO 10303, Industrial Automation Systems and Integration - Product Data Representation and Exchange*, ISO TC 184/SC4, 1992.

[7] H. Larson, *Translation of 3D CAD models to Modelica*, Master Thesis, IDA, Linköping Univ., Sweden, April 1999.

[8] J. Leech, *OpenGL Web Site*, http://reality.sgi.com/opengl

[9] M. Otter, H. Elmqvist and F. E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[10] *Modelica WWW site*, Modelica Group, http://www.modelica.org

[11] *Modelica activities in PELAB*, PELAB Group, Linköping University, http://www.ida.liu.se/~pelab/modelica

[12] *Multigen*, OpenFlight section, MultiGen-Paradigm, Inc., http://www.multigen.com

[13] M. Otter, *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. Dissertation, Fortschrittberichte VDI, Reihe 20, Nr. 147, 1995.

[14] J. Owen, *STEP – An Introduction*, Information Geometers Ltd., 1993, ISBN 1-874728-04-6.

[15] J. Parmar, *Modeling Autonomous Helicopters using Modelica*, Master Thesis, IDA, Linköping Univ., Sweden, To appear in June 1999.

[16] *SolidWorks*, SolidWorks Corporation, http://www.solidworks.com

[17] *StereoLithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[18] WITAS group, *The Wallenberg Laboratory for Research on Information Technology and Autonomous Systems*, Linköping University, http://www.ida.liu.se/ext/witas

[19] *Working Model* , MSC.Working Knowledge, http://www.krev.com

[20] F. Fischer, J. Seybold, *Modeling, Optimization and Visualization of Multibody Systems exemplary shown at the Example of a pedestrian Bridge in Kiel*, Interner Bericht Nr. 109, Institute for Computer Applications, University of Stuttgart, September 1995.

[21] Brisdata, *Indoor Climate and Energy*, http://www.brisdata.se/ice/

Paper 8

AN INTEGRATED MODELICA ENVIRONMENT FOR MODELING, DOCUMENTATION AND SIMULATION[1]

Peter Fritzson, Vadim Engelson, Johan Gunnarsson

PELAB, Programming Environment Laboratory
Department of Computer and Information Science

Linköping University, S-581 83
Linköping, Sweden

Email: {petfr,vaden,johgu}@ida.liu.se

KEYWORDS

Modelica, Mathematica, programming environments, simulation, documentation

ABSTRACT

Modelica is a new object-oriented multi-domain modeling language based on algebraic and differential equations. In this paper we present an environment that integrates different phases of the Modelica development lifecycle. This is achieved by using the Mathematica environment and its structured documents, "notebooks". Simulation models are represented in the form of structured documents, which integrate source code, documentation and code transformation specifications, as well as providing control over simulation and result visualization.

Import and export of Modelica code between internal structured and external textual representation is supported. Mathematica is an interpreted language, which is suitable as a scripting language for controlling simulation and visualization. Mathematica also supports symbolic transformations on equations and algebraic expressions, which is useful in building mathematical models.

BACKGROUND

Integrated simulation environments are advantageous in order to work effectively and flexibly with simulations. Users prepare and run simulations as well as investigate simulation results. Several auxiliary activities accompany simulation experiments: requirements are specified, models are designed, documentation is associated with appropriate places in the models, input and output data as well as possible constraints on such data are documented and stored together with the simulation model. The user should be able to reproduce experimental results. Therefore input data and parts of output data as well as the experimenter's notes should be stored for future analysis.

Traditionally, simulation and accompanying activities have been expressed using heterogeneous media and tools:

- a simulation model is traditionally designed on paper using traditional mathematical notation;
- simulation programs are written in a low-level programming language and stored on text files;
- input and output data (if stored at all) are saved in proprietary formats needed for particular applications and numerical libraries;
- documentation is written on paper or in separate files that are not integrated with the program files;
- the graphical results are printed on paper or saved using proprietary formats.

When the result of the research and experiments, such as a scientific paper, is written, the user normally gathers together input data, algorithms, output data and its visualizations as well as notes and descriptions. One of the major problem in simulation development environments is that gathering correct versions of all these components from various files and formats is difficult and error-prone.

USING MATHEMATICA NOTEBOOKS

Our approach to the integration problem is based on the Mathematica (Wolfram 1997) environment and its programmable notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can include other cells and/or arbitrary text or graphics. In particular a cell can include a code fragment or a graph with computational results. The hierarchy of cells corresponds to the traditional hierarchy of chapters, sections, and paragraphs used in textual documents and available in advanced word processors like FrameMaker or MSWord. The text can be written in different styles, including certain font families, bold, italic, color and size as well as mathematical typesetting.

The contents of cells can be

- parts of models (a formal description that can be used for verification, compilation and execution of some simulation model);
- text/documentation (used as comments to executable, formal model specifications);
- dialogue forms for specification and modification of input data;
- result tables (the results can be immediately represented in table form);

1. Published in Proceedings of The 1998 Summer Computer Simulation Conference} (SCSC 98) July 19--22, 1998, Reno, Nevada, pp. 308-313.

- graphical result representation (with 2D vector and raster graphics as well as 3D vector and surface graphics);
- 2D graphs that are used for various model structure visualizations:
  - class diagrams
  - variable dependency diagrams
  - data structure diagrams

THE MODELICA LANGUAGE

The language called Modelica (Modelica 1998) for hierarchical physical modeling has been developed in an international effort. It is an object-oriented language (Elmqvist, 1997, Fritzson and Engelson, 1998) for modeling of physical systems. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. It offers three important features: 1) non-causal modeling based on differential and algebraic equations; 2) multidomain 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 **`class`** construct.

Modelica models are built from classes. Like in other object-oriented languages, a class contains variables, i.e. class attributes representing data. The main difference compared to traditional object-oriented languages is that instead of functions (methods) the programmer uses equations to specify behavior. Equations can be written explicitly, like `a=b`, or can be inherited from other classes. Equations can also be specified by the **`connect`** statement.

Modelica model of lunar landing

As an introduction to Modelica we will present a model of a rocket landing on the moon surface.

There are two components (*Rocket* and *Planet*) used in the *Landing* class. The class `Rocket` embodies floating point (`Real`) variables, simulation-time constants (**`parameter`**) and equations for vertical motion of the rocket above the planet surface. The motion is influenced by gravitational force and rocket motor force (`Thrust`). Rocket mass varies since the rocket loses fuel mass proportional to the amount of thrust from the rocket motor. Velocity and acceleration are time-derivatives of height and velocity as specified by the **`der(...)`** construct.

The class `Landing` includes `apollo` (an instance of class `Rocket`) and `moon` (an instance of `Planet` with specified planet `mass` and `radius`).

Dynamic systems are described by models where behavior evolves as a function of time, i.e. all the variables in the model evolve as functions of time. Modelica has a predefined variable `time` which steps forward during system simulation. The simulation models the rocket behaviour from `time=0` until the rocket touches the ground.

Equations describe the thrust function, a step function with an initial thrust force level `f1` during the time interval [`0..thrustDecreaseTime`] and a second thrust level `f2` during the time interval [`thrustDecreaseTime..thrustEndTime`]. The step functions can conveniently be expressed using Modelica conditional expressions (the **`if-then-else`** construct).

The gravitational field is expressed as:

$$g(t) = \frac{GM_{planet}}{(h(t) + radius_{planet})^2}$$

The Lunar landing model expressed in Modelica

```
class Rocket "generic rocket class"
 Real height(start=59000);
 Real velocity(start=-2000);
 Real mass(start=1000);
 Real acceleration;
 Real Thrust;
 Real gravity;
 parameter Real massLossRate=0.000277;
equation
 Thrust-mass*gravity=mass*acceleration;
 der(mass)=-massLossRate*abs(Thrust);
 der(height)=velocity;
 der(velocity)=acceleration;
end Rocket;
class Planet "generic planet class"
   parameter Real mass;
   parameter Real radius;
end Planet;
class Landing "landing of a rocket onto a
planet"
   parameter Real force1=36000;
   parameter Real force2=1500;
   parameter Real thrustEndTime=210;
   parameter Real thrustDecreaseTime=43;
   parameter Real G=6.672e-11;
   Rocket apollo;
   Planet moon(mass=7.382e22,
               radius=1.738e6);
equation
apollo.Thrust=
 if (time<thrustDecreaseTime) then force1
 else if (time<thrustEndTime) then force2
 else 0;
apollo.gravity=(G*moon.mass)/
         (apollo.height+moon.radius)^^2;
end Landing;
```

Note that Modelica equations do not specify which variables are inputs and which are outputs. Thus, the causality of equations-based models is unspecified. This is fixed only when the equation system is solved.

Simulation Semantics

Classes, instances, and equations are translated by the Modelica compiler into a flat set of differential-algebraic equations, constants and variables. The initial values for `time=0`, specified by the `(start=...)` construct can be taken from the model definition. A simulation engine finds a numerical solution of the system of ordinary differential equations. All model variables are functions of (modeled) time during the simulation.

The simulation engine is a fairly complex piece of software that can be implemented in many different ways. In each case this engine sorts equations (Elmqvist, 1997), finds algebraic loops, checks consistency of ordinary differential equations (ODE), controls an ODE solver, and computes requested variable values from the solution found.

Existing Environment and Need for MathModelica

There is an environment for the Modelica language and engine for Modelica simulations, based on the Dymola system (Dymola 1998). The code, documentation, input data, graphical and numerical results are represented in different, heterogeneous formats, which is a disadvantage of the system. Furthermore, the user is not able to extend the system and introduce new software components. Additionally, the user cannot perform user-specified symbolic (algebraic) manipulations with the formulae used in the code.

For these reasons, we are developing the extensible integrated MathModelica environment which is partly based on the Mathematica system and notebooks.

A specific feature of Mathematica is that models are normally not written as free formatted text. Instead, Mathematica expressions (terms) are used. These can be conveniently written in a tree-like prefix form, or entered using standard mathematical notation. Every term is a number, an identifier or a form such as:

head **[***term*$_1$*,..., term*$_n$**]**

In order to satisfy this requirement, we designed the new *MathModelica language*[1]. Note that MathModelica has the same abstract syntax and the same semantics as Modelica, but a different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below.

The MathModelica language uses some Mathematica notation, such as:

term$_1$**;...;***term*$_n$, **{***term*$_1$*,...,term*$_n$**}**,
term$_1$ *term*$_2$, *term*$_1$**==***term*$_2$, *term*', *term*$_1$**`***term*$_2$,

and arbitrary arithmetic expressions composed from terms.

The ' (tick) means time-derivative. The ` (backtick) character in MathModelica corresponds to . (dot) notation in Modelica.

Lunar Landing Example in MathModelica

```
Class[Rocket;
 Declare[
  Real[start->59000,unit->"m"] height;
  Real[start->-2000,unit->"m/s"] velocity;
  Real[start-> 1000,unit->"kg"] mass;
  Real acceleration;
  Real Thrust;
  Real gravity;
  Parameter Real massLossRate=0.000277 ;
 ];
 Equation[
  Thrust-mass*gravity==mass*acceleration;
  mass'==-massLossRate*Abs[Thrust];
  height'==velocity;
  velocity'==acceleration;
 ]
];

Class [Planet; "**********************"
 Declare[
   Parameter Real mass;
   Parameter Real radius;
 ]
];

BeginClass[Landing];
Declare[
  Parameter Real force1=36000;
  Parameter Real force2=1500;
  Parameter Real thrustEndTime=210;
  Parameter Real thrustDecreaseTime=43;
  Parameter Real G=6.672*10^-11;
  Rocket  apollo;
  Planet[mass->7.382*10^22,
         radius->1.738*10^6]  moon;
];


Equation[
 apollo`Thrust==
  Which[time<thrustDecreaseTime, force1,
       time<thrustEndTime,     force2,
       True,                   0];
```

1. This paper presents the preliminary syntax of MathModelica.

```
  apollo`gravity==
                        G·moon`mass
                  ------------------------
                  (apollo`height+moon`radius)^2
];
EndClass[Landing];
```

Expressions used in MathModelica are "dynamic". This means that they can be created as result of symbolic and algebraic transformation performed by Mathematica. These transformations can be integration, differentiation, expansions of series, simplification, etc.

Formal Syntax of MathModelica

The formal concrete syntax of MathModelica is described by a BNF grammar[1] as shown below, where keywords are all marked with **bold**, other terminals are courier (if they are written as here) or *italic* (generic tokens, like integers or strings):

```
program::= class*
class::= shortClass | longClass
shortClass::= Class [classname;
         (ext | decls | equations )* ];
```

When the short class definition is evaluated in Mathematica, the class description is added to the model.

```
longClass::=BeginClass [classname];
        (ext | decls | equations | comment )*
            EndClass [classname];
```

The constructs **BeginClass[]** and **EndClass[]** are balanced. Class definition can be split, and written in several cells. The **EndClass[***name***];** adds (when evaluated), definition of class name to the syntax tree. These constructs allow the user to insert documentation into class code (Figure 1).

A class can extend (inherit) other classes as in Java:

```
ext ::= Extends [classname*];
decls     ::= Declare [ onedecl* ];
equations ::= Equation[ oneequation* ];
onedecl   ::= [Parameter] type
     [ [ attr *] ] [ [idx *] ]
      varname [ = value ];["short comment";]
```

Note that "parameters" are simulation time constants and their value can be stated in the simulation model.

```
idx ::= integer (used for array declaration)
attr ::=
   start->value; (bound value of variable for time=0)
  |unit->"string" (unit used for user interface only)
  |varname->value; (specifies value for component)
comment::= text written in a separate notebook cell
type ::= Real | Integer | classname
oneequation ::= expr == expr ;
expr ::=  varname' | expr+expr
  | expr-expr | expr/expr | expr*expr
  | builtinfunction(expr) | expr[[expr]]
  | expr`varname | number | expr^expr
```

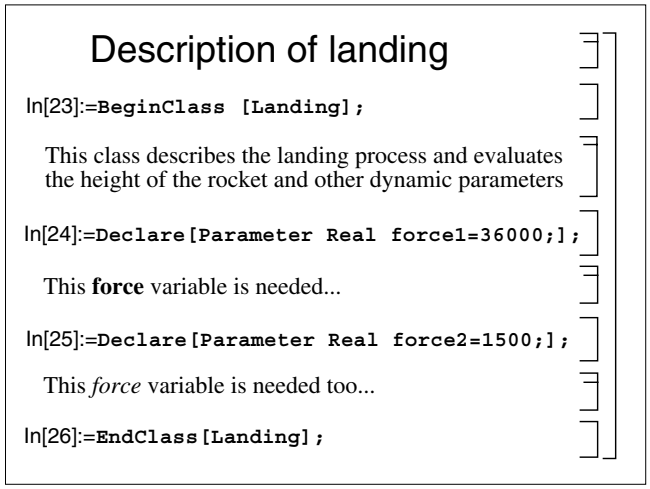


Fig. 1. Notebook fragment with MathModelica class declaration and comments.

In Figure 1 the symbols in the right margin denote different kinds of cells and the hierarchy constructed from the cells. The bracket ⊐ means a cell with code (an executable command); the bracket with the horizontal bar ∃ means a cell with text (a fragment of documentation). A larger ⊐-bracket marks a cell that contains eight smaller cells. A hierarchy of cells of unlimited depth can be easily created.

THE EXPERIMENTAL ENVIRONMENT

MathModelica is both a language and an environment.

The Mathematica built-in functions and MathModelica environment functions can be used in order to manipulate the model. Naturally, the most important operation using the model is simulation. This can be performed in different ways:

- translation of the model to Mathematica, and simulation using the Mathematica built-in ODE solver;
- translation to Modelica, and using the Dymola-based environment for simulation;
- translation to C++ using the MathCode system (MathCode 1998, Fritzson 1997), a Mathematica to C++ compiler.

1. In BNF notation [F] means optional text F; F|G is an alternative between F and G; F* is repetition of F 0 or more times (Aho *et al.* 1986).

TRANSLATION FROM MATHMODELICA TO MATHEMATICA

In order to convert a MathModelica model to Mathematica the function `ModelicaToMathematica[]` is called. Variables and functions declared in the model then become interactively accessible by the user working in the Mathematica environment.

Classes, variables and equations in the MathModelica model are converted to a flat list of variables and equations. First, the constants are set:

```
setparam[]=(apollo`massLossRate = 0.000277;
            thrustDecreaseTime=43;
            thrustEndTime=210;
            moon`mass=7.382*10^22;
            moon`radius=1.738*10^6;
            force1=36000;
            force2=1500;
            G=6.672*10^-11);
```

The Mathematica differential equation solver (`NDSolve`) requires that only differential equations are specified in the equation list, so algebraic equations have to be handled separately.

There is an equation (here named `eq1`) with the variable `apollo`acceleration` the translator has to extract:

```
eq1:=
 apollo`Thrust[t] - apollo`mass[t]*
 apollo`gravity[t]  ==
   apollo`mass[t]*apollo`acceleration[t];
```

To extract the variable `apollo`acceleration` the algebraic equation solver is called:

```
 Solve[eq1,apollo`acceleration[t]];
```

The `Solve` function expresses the variable `apollo`acceleration[t]` in terms of other variables. A new function definition for the acceleration is created:

```
apollo`acceleration][t_]:= - ( (
  apollo`gravity[t] * apollo`mass[t] -
   apollo`Thrust[t]) / apollo`mass[t]);
```

Two other functions are already given in the MathModelica code:

```
apollo`gravity[t_]:=(G*moon`mass) /
        (apollo`height[t]+moon`radius)^2;

apollo`Thrust[t_]:=
  If[t < thrustDecreaseTime ,force1,
    If[t < thrustEndTime, force2, 0]];
```

Now, three ordinary differential equations are specified:

```
eqs={
 apollo`mass'[t]==- apollo`massLossRate*
                Abs[ apollo`Thrust[t]],
 apollo`height'[t]==apollo`velocity[t],
 apollo`velocity'[t]==apollo`acceleration[t]
};
```

Initial conditions are necessary for the solution:

```
initcond={
   apollo`height[0]==59000,
   apollo`velocity[0]==-2000,
   apollo`mass [0]==1000};
```

Solutions are required for these functions:

```
vars={apollo`height[t],apollo`velocity[t],
   apollo`mass[t]}
```

When the function `ModelicaSimulate[time`max`]` is called, initial conditions, equations, and functions are passed to an ordinary differential equation solver which is a Mathematica built-in function:

```
res=NDSolve[Join[eqs,initcond],var,
            {t,0,time`max`}];
```

As a result, Mathematica creates polynomial approximations for all functions we want to find, known as interpolate-functions.

Result visualization

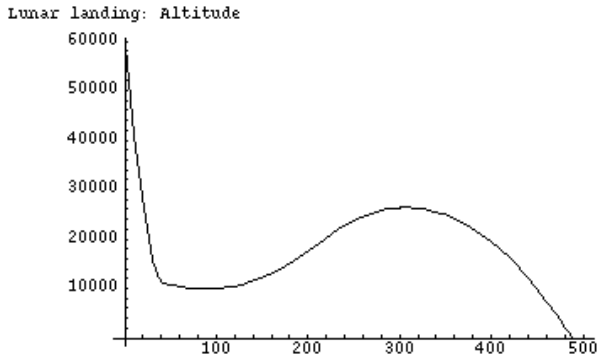


Fig. 2. The `apollo`height` value for time in [0,`time`max`] (`time`max`=500)

The functions computed by `NDSolve` can be visualized graphically. For instance, the height of the rocket can be visualized (see Figure 2) by the call

```
 ModelicaPlot[ apollo`height[t],
               "Lunar landing: Altitude"];
```

Other Scripting Utilities

Three scripting utilities – `ModelicaToMathematica[]`, `ModelicaSimulate[]` and `ModelicaPlot[]` have been illustrated above.

`ImportModelica[]` accepts textual traditional Modelica code, parses it and converts to MathModelica code. `ExportModelica[]` performs the opposite translation. It can be used with a Modelica implementation, available in the Dymola environment.

Input Data Specification

Another script command, `ModelicaInData[]` allows selecting parameter values and specifying boundary conditions interactively. The dialogue window is created automatically from specification of data structures (Engelson and Fritzson, 1996) in the Modelica code (see Figure 3). Variable values can be edited and new simulation runs can produce results immediately. In the future this can be used for computational steering of Modelica simulations.

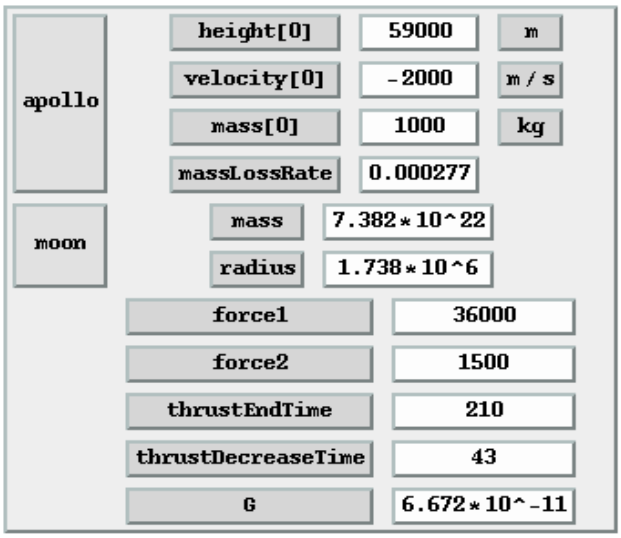


Fig. 3. A Mathematica dialogue box for setting input variable values of a MathModelica simulation. The boxes `apollo` and `moon` represent cell instances with several components.

CONCLUSION

The MathModelica is an extension of the Modelica language targeted for work within the Mathematica environment. This language is object-oriented (the programs consist of collections of classes). The language is equation based: instead of traditional functions and procedures we use non-causal equations which specify algebraic and differential relations between numerical variables. Input-output causality is not specified, and therefore these equations can be used in multiple ways.

The environment integrates most activities needed in simulation design and use: documentation, modeling (coding), symbolic processing and transformation of formulas, input and output data visualization. This advanced programming environment can be applied in various simulation applications.

ACKNOWLEDGMENTS

This work had been supported by the Wallenberg Foundation in the WITAS project.

REFERENCES

Aho, A., Sethi, R., Ullman, J., *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

Dymola Home Page, *http://www.Dynasim.se*, 1998.

Elmqvist, H., Mattson, S.E., Modelica - The Next Generation Modeling Language - An International Design Effort, in *Proceedings of First World Congress of System Simulation*, Singapore, September 1-3, 1997.

Engelson, V., Fritzson, P., Fritzson, D., Automatic Generation of User Interfaces From Data Structure Specifications and Object-Oriented Application Models. In *Proceedings of ECOOP96,* Linz, Austria, 8-12 July 1996, Pierre Cointe (Ed.), pp. 114-141. Springer-Verlag, 1996 .

Fritzson, P., Static and Strong Typing for Extended Mathematica. In *Innovation in Mathematics*. Proceedings of the Second International Mathematica Symposium, Rovaniemi, Finland, July 1997, V.Keränen, P. Mitic, A. Hietamäki (Ed.), pp. 153-160.

Fritzson, P. and Engelson, V., Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. Accepted for publication in *Proceedings of ECOOP-98*, Brussels, July 1998.

Fritzson, P., and Fritzson, D., The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers and Structures,* Vol 45, No 2, pp 387-395, 1992.

Modelica Home Page, *http://www.Modelica.org* 1998.

MathCode Home Page, http://www.mathcore.com, 1998

.Wolfram, S., *The Mathematica Book*, Wolfram Media, 1997.

Paper 9

3D Graphics and Modelica - an integrated approach.

Vadim Engelson

Abstract

The Modelica[8] standard library and available Modelica tools [4, 7] contain some facilities for specification of 3D geometry and 3D graphics. Geometry and graphics is associated with physical objects included in simulated Modelica models. However, important graphics properties are missing from this model. In particular, physical objects cannot change their shape (geometry) and rendering features (graphics) dynamically. The physics of simulation, is often not affected by geometry of physical objects. For instance, a body is often approximated by its center of mass under certain conditions. Either simple predefined shapes or specifications of geometry via external files are used. The last facility leads to separation between the model and the corresponding graphics and geometry.

Our proposal is to integrate 3D geometric and graphical features with Modelica models of physical objects. The 3D graphics information is specified explicitly via annotations containing certain graphics primitives or using instances from a specially designed geometry class library. The motivation, syntax and implementation outline for this approach are discussed in this report.

1 Background

Modelica[8] is an object-oriented language for modeling dynamic systems. When Modelica models are simulated, variables constrained by algebraic and differential equations evolve in time. The current tools for visualization of Modelica simulation results [4, 7, 6] have facilities to render *static* objects only. These objects are rigid, i.e. they cannot change their shape. Modelica is a perfect tool for modeling physical processes, especially mechanical systems. However, the bodies in these processes are modeled using very simplified geometries. One of the reasons is that there are limited facilities for specification of body geometry and 3D graphical information in Modelica. Either a fixed set of primitive shapes should be used, or external files with shape and graphics description are referenced.

Modelica is a convenient tool for modeling mechanical systems. The Multibody Simulation library (MBS)[10] is intended for this purpose. The dynamic properties of bodies are obviously in dependent on their geometry. This circumstance is only used in MBS for a limited set of bodies with primitive shapes (box, cylinder, etc.). Furthermore, the shape of the bodies is not utilized for collision detection and contact force computation.

Our solution to the problem of integrating Modelica and graphics is to *develop a standard notation* for 3D geometry data in Modelica, and to *incorporate* this information into appropriate classes used in Modelica models. This information will be created during model design, updated when the model changes, used for computation of certain physical quantities (which can vary) of the bodies during simulation, and used for rendering of the bodies during online or offline visualization. Initially this information can either be created manually, or imported from a modeling tool, e.g. a CAD system.

This solution is motivated by several factors:

- Modelica models and their visualizations have the same structure. Therefore this structural consistency should be supported. The geometry of a whole Modelica model is the same as the union of the shapes of all class instances in the model.

- Values can be reused between a Modelica class of a physical object, corresponding geometric construction and 3D graphical model. In particular, constants defined in a geometric model are reused as constants in the Modelica physical model since they are related to the same physical object. Parameters from the Modelica model affect the visualization of the corresponding 3D graphic model.

- Graphics and model code can be kept together by including all the necessary information into the same document. Such documents, together with textual documentation which is already included as a textual annotation in Modelica models, can be used for information exchange between developers.

There are also some disadvantages of inserting geometry descriptions into Modelica models. If a complex geometry is used the model becomes very large, having the following consequences:

- It can be cumbersome to edit the textual representation of the model. However the geometry descriptions can be hidden when connection diagrams are edited, using special model editing environments (MathModelica[7]) or their customization (e.g. special Emacs[5] mode).

- Compilation of models with complex geometries can take too much time. A solution to this problem can be placing geometry descriptions in Modelica code such way that these can be ignored during normal compilation. This issue is discussed in Section 3.

It is important to note that annotations in general, and geometric annotations in particular, *do not change the semantics of Modelica models*. They do not create any new equations or new variables. However, they can affect the execution of Modelica code (if the user specifies function calls that request data based on geometry), in a similar way as input data can affect execution.

2 Simulation and Visualization Requirements

There are several requirements which are essential for the simulation of physical models with non-trivial geometry:

- The volume, mass (when density is given), position of the center of mass (assuming that bodies are homogeneous), and the matrix of inertia should be automatically evaluated for each body.
- In the context of simulations where contacts between bodies are important, collisions can be detected. Collisions between bodies can be detected only if the complete geometric information is given for all the bodies.
- Contact force evaluation: if a collision of bodies is detected, an external force caused by the collision should affect the physical model behavior.
- Partial differential equations (PDEs) are not part of Modelica yet. However it is clear that geometry information related to domain, distribution and grid generation is necessary for solving PDEs. This geometry data should be described in Modelica model somewhere.

For some application areas the geometry associated with physical models can be used for simulation; in some others there is no relation between simulation and geometry at all. In particular, geometry is ignored in electrical modeling. The body geometry is partially used in dynamic analysis of mechanical systems (the kinematic outline is usually sufficient). In chemical processes a few simple shapes, e.g. cylinders are sufficient. Complete and exact geometry descriptions are, however, necessary for computing contact forces between bodies, i.e. friction of surfaces and collision. Complete geometry information is necessary for visualization of physical processes.

The most important features that should be available in the visualization environment are:

- primitive graphic objects with vertex coordinates (point, line, triangle, quadrilateral[1] and polygon) in 2D and 3D; These coordinates can be constants or variables computed during simulation.
- colors;
- objects with fixed shape that change position and orientation over time (rigid body visualization).

Other features that should be available in the environment are:

- normal vectors to surfaces (necessary for correct rendering);

[1] Quadrilateral is a polygon with four vertices in 3D.

- material illumination and transparency properties;
- texture raster and texture mapping;
- coordinate transformations, such as translation, rotation, scaling (including transformations and rotation matrices such as those available in the MBS library);
- predefined non-primitive objects with size and position attributes, such as box, cylinder, beam, sphere, cone, etc.;
- surfaces defined as splines[2];
- trees with nodes for transformations, switches[3] , levels-of-detail etc.;
- visualization of forces, speed, acceleration etc., e.g. in the form of arrows;
- objects that are able to refer to graphic data in other formats (such as VRML[12] or STL[1]) and/or other files;
- objects that change their shape during dynamic visualization (non-rigid, flexible body visualization). The shape is specified by some constants and variables, and the values for the variables is computed during simulation.

Visualizations defined in Modelica models can be highly dynamic since all the numerical values used in graphic definitions can be not only constants, but arbitrary Modelica variables or numerical expressions.

3 Graphic Object Representation in Modelica Code

An important question is *how* to add geometric or graphic information to a Modelica model. These are not equations, not functions, not variables, but tree-like graphs with structural information, numerical constants and variables. The structure should match certain syntax rules to be readable by simulation and visualization software.

Two alternatives can be considered (see Figure 1):

Using classes: User-defined models of physical objects should inherit certain general graphic classes (from a new special graphic Modelica library) or include components of those classes. For instance, currently for visualization with DymoView[4] or MVISLIB[6] a special class `VisualMbsObject` should

[2]Sometimes there is a need to visualize surfaces defined via parametric functions given in a Modelica model. This case requires a very special treatment of these functions, and using function names in the same way as variables. Since this deviates too far from Modelica syntax and complicates implementation we don't cover this topic in this report.

[3]*Switch* nodes allow the user to specify several completely different graphic objects, and alternate between them depending on the switch variable value.

```
model C "using classes"
 PrimitiveGeometry x(c={Sphere(p={0,0,0},radius=1)});
 ...
end C;

model A "using annotation"
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=1)}))
 ...
end A;
```

Figure 1: Example of Modelica classes with graphical annotations and with graphical classes.

```
Icon(
 Rectangle(
  extent=[-90, 15; 0, -15],
  style(color=8 , gradient=2 ,
        fillColor=8 , fillPattern=1 )),
 Rectangle(
  extent=[0, 50; 100, -50],
  style(color=8 , gradient=3 ,
        fillColor=8 , fillPattern=1 )),
 Text(
  extent=[-100, 60; 100, 122],
  string="%name")),....
```

Figure 2: Example of Modelica annotation defining an icon.

be used (or classes that use or extend this class). Each class instance can include a description of some corresponding graphics.

Using annotations: The classes might contain special *annotations* which define geometric and graphic properties. Annotations in Modelica are tree-like structures with labeled branches and attribute values, e.g. as the example shown in Figure 2

Currently annotations are used for icons, diagrams (here a particular grammar for the structure of the annotations is used) and for documentation (a string with free contents).

3.1 Choice Between Classes and Annotations

The first alternative (using classes) requires creating a Modelica visualization library (with a set of Modelica classes) and adding new classes for particular geometric structures to this library. The second variant might require a specific analyzer (parser) for graphic annotations, which can be an extension to the Modelica parser.

Sometimes there is a need to include some format (e.g., VRML[12]) directly (verbatim) into the Modelica model code. In this case annotations can conveniently

be used since they are effectively ignored when the model is compiled for the purpose of simulation.

Sometimes there is also a need for dynamic modification of parameters of some geometric features during simulation or visualization. For instance, the height of a box increases as a function of time; its mass and center of mass should be computed for simulation; the shape of the box should be updated for online and offline visualization [6] as well. This can be done by placing some variables into geometric classes and setting up some equations to connect these variables with the rest of the simulation. This cannot be done easily in the annotation approach, since annotations do not affect the simulation and cannot affect the visualization of simulation results.

The architecture of a special environment for handling shape annotations is described in Section 7.

It would be convenient to ignore all information related to graphics when the model is simulated just to obtain numerical values of model dynamics. In such a case classes and instances with graphic information should be "detached" from the model in order to reduce the number of classes, equations and variables. This can be needed for higher performance, memory reduction and speed-up of the Modelica compiler. It is easy to detach all the annotations at once, but it is more difficult to switch off all geometry-related classes of an application.

Each Modelica class or instance might have an annotation defining some geometric and/or graphic properties. Obviously, there can be classes that contain graphic annotations only, and these can be instantiated or extended (inherited) in other classes. Therefore both syntax alternatives have roughly the same capabilities.

There are annotations defining icon geometry in Modelica [9]. The icons can be displayed graphically in Modelica implementations[4, 7] (Figure 2). These icons are displayed as 2D graphic primitives when appearing in Modelica connection diagram editors. These annotations are not used for simulation purposes and not stored in the simulation output data.

In the second alternative the proposed graphic annotations will represent 3D graphic primitives. Some special component in the Modelica implementation should care of the task of analysis and appropriate handling of these specific annotations.

A problem arises with geometry definition in annotations since sometimes the geometry contains *connectors* (such as `MbsCutA`) and there is no place in the Modelica syntax to specify connection between a class instance inside an *annotation* with the rest of the model. However, it is possible to write a variable name in the annotation.

For describing the structure of graphics there is a need for non-homogeneous arrays, i.e. arrays consisting of instances of different classes. This is not possible in standard Modelica class syntax, but it is allowed in annotations.

Thus, both approaches have some drawbacks. The rest of this report is based on the *second* alternative (using annotations). The syntax of annotations, however,

is described by Modelica class notation.

3.2 Shape Structure and Modelica Model Structure

A Modelica model refers directly and indirectly to a number of classes. These classes have components (variables) of types defined by some other classes. Instantiation of variables and equations always starts from the last (the main) class of the model (this class corresponds to the root node in the tree of Modelica classes), and continues recursively to the components.

The total collection of geometric bodies included in a Modelica model (when used for simulation or visualization) is specified recursively. The shape corresponding to a whole Modelica application is a combination of the shapes of all instances declared in the last (the main) class of the textual presentation of the application model.

If a class defines its own geometric annotations it can refer (within its attribute structures) to the annotation of the parent class (for this purpose we introduce a special keyword `Super`) and attributes of the components of the class (we introduce the keyword `Component(`*name*`)`). By default the annotation of the parent class is overridden and annotations of all components are ignored.

If a class does not define its own shape attributes, then the shape of the superclass and the shape of components are just joined together:

```
Super, Component(name1), Component(name2), ...
```

Figure 3 gives definitions of five classes with geometric information. The actual shapes associated with each class are given in the comments.

Visualization and geometric properties are inferred from the shape properties. The visualization of a complete Modelica application consists of the shapes of all the object instances. If geometric information is enough for computation of object volume, then the volume associated with a whole Modelica application is just the sum of volumes of all the instances (it is assumed that they do not penetrate each other).

4 Geometry Definition Syntax in Modelica

The library of data structures that can be used for geometric data description is presented in Figure 4.

Generally, graphics can be defined using two main approaches:

- Specification of a set of primitives (including triangles, spheres, etc.) and operations (translation, rotation, extrusion, etc.).
- Using an external graphics format (such as STL[1], VRML[12], DXF[2], etc). The names of the files that contain the graphics are specified *or* the graphics in its external format is included directly (verbatim) into the model.

```
model A
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=1)}))
end A;
model B extends A;
 Real r;
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=r)}))
 // This annotation overrides the annotation inherited from A
end B;
model C extends A;
 A a;
 B b;
 annotation(PrimitiveGeometry({Super,Box(...),Component(b)}))
 // This annotation overrides the annotation inherited from A.
 // C contains two spheres and a box.
end C;

model D extends A;
 A a;
 B b;
 // No annotation. D contains three spheres.
end D;

model E;
 C c; D d; // contains five spheres and a box.
end E;
```

Figure 3: Example of Modelica annotations defining geometries.

4.1 Syntax for External Graphic Format Specification

Some components or classes might have their graphics and geometry stored in an external graphics format.

If an external graphics format is used for a Modelica model, we need to specify the following:

- the format identifier,
- the external file name of the graphic file,
- (optional) file component identification.

In these cases an annotation is attached to a component or a class, e.g. as below:

```
annotation(ExternalGeometry(format="STL",
                            file="arm001.stl"))
```

There exist many different 3D graphics formats. The STL format is just one of the available portable formats for description of static rigid bodies.

This format may include sub-parts in the geometry description. Typically, assemblies of components are described as trees. It is possible to access subcomponents via a unique name or a sequence of names.

```
annotation(ExternalGeometry(format="STL",file="robot.stl",
                            component="17"))
```

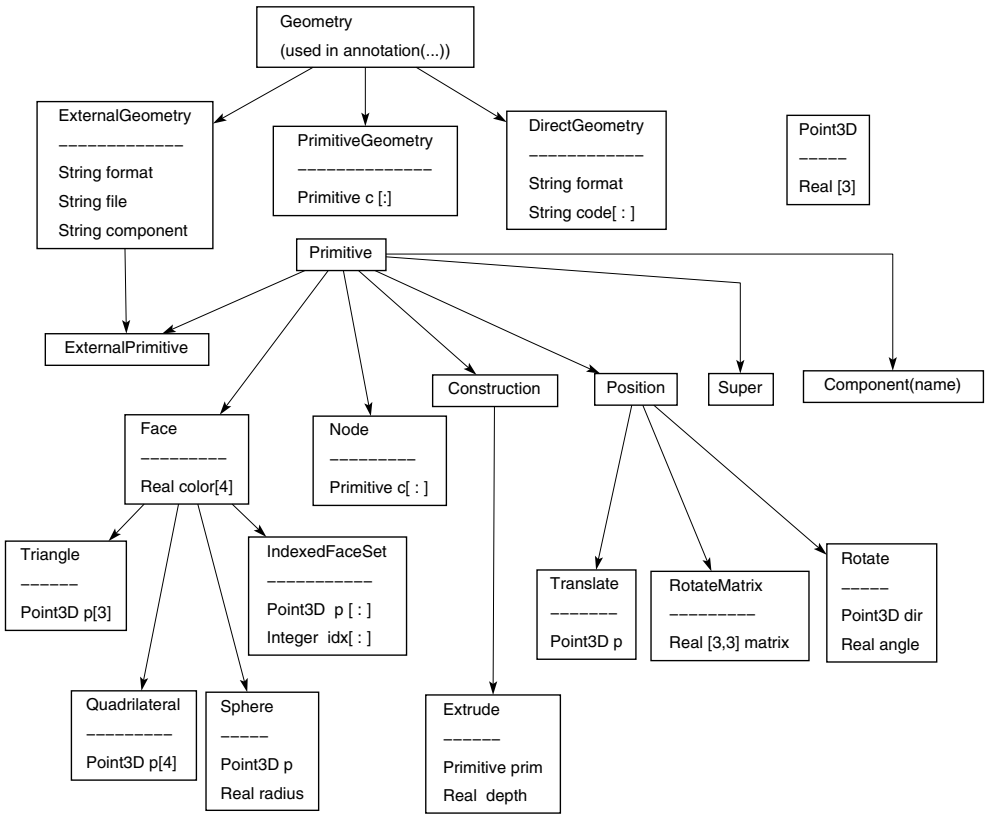


Figure 4: Relationship diagram for data structures describing geometric annotations.

Graphic information in an external format can be included into the Modelica model directly (verbatim), as an array of strings:

```
annotation(DirectGeometry(
    format="STL",
    code={"solid arm",
          "facet normal 0.0 -1.0 5.9e-08",
          "outer loop",
          "vertex -1.9e+01 5.1e-07 1.08e+01",
          "vertex -1.9e+001 -6.2e-007 -8.3",
               ...}))
```

5 Primitive Geometric Objects

A geometric model might contain many different primitive objects like triangles, quadrilaterals, spheres, boxes, and parametric surfaces.

In particular the following types should be available:

```
type  PrimitiveGeometry=Primitive [:];

record Primitive;
// has no specific components
end Primitive;

type Point3D = Real [3];

record Face
    extends Primitive;
    Real color[4]; /* RGBA */
    /* other features can be here */
end Face;

record Triangle
    extends  Face;
    Point3D  p[3];
end Triangle;

record Quadrilateral
    extends  Face;
     Point3D  p[4];
end Quadrilateral

record Sphere
     extends  Face;
     Point3D  p;
     Real radius;
end  Sphere

record IndexedFaceSet
     extends  Face;
     Point3D  p[:];
     Integer idx[:];
```

An annotation defining a geometry might look like:

```
annotation(
 PrimitiveGeometry({
    Triangle(p={{0.3, 0.4, 0.2},
                      {1.4, 0.4, 0.2},
                      {1.4, 0.5, 0.3}}),
    Sphere(p={0.2,0.4,0.6}, radius=1.2  )
  }))
```

6 Position Specification

Generally geometry is specified as a tree. The semantics of `Translate`, `Rotate` and `Scale` re the same as in OpenGL[11]. Other OpenGL-like constructs can be added in the same way. These primitives define transforms from an old coordinate system to the new coordinate system, and all further drawing in the current list of primitives is applied in the new coordinate system.

Corresponding definitions are:

```
record Position extends Primitive
end Position;

record  Translate
    extends Position;
    Point3D  point;
end  Translate;

record  Rotate
    extends Position;
    Point3D  direction;
    Real     angle;
end  Translate;

record  RotateMatrix
    extends Position;
    Real [3,3]  matrix;
end  Translate;

record Scale
   extends Position;
   Point3D size;
end Scale;
```

A `Node` is a `Primitive` that might contain a number of other nodes:

```
record Node
    extends Primitive; /* it is a primitive */
    Primitive [ : ];
     /* contains some other nodes and other primitives */
end Node
```

The position specification is stored in the tree, together with the faces. The example below defines two spheres. As result of combination of two translations, the center of one of them is shifted to (1,0,1), the other becomes (1,0,-1) .

```
annotation(
  PrimitiveGeometry( {
     Translate( point={1,0,0}),  // Translates both nodes
     Node({
        Translate( point={0,0,1}), // Translates only one sphere
        Sphere( point={0,0,0}, radius=1)
               })
              ,
       Node({
        Translate( point={0,0,-1}), // Translates only one sphere
        Sphere( point={0,0,0}, radius=1)
                })
     }))
```

6.1 Specification of Level of Detail

In case of rendering we can consider specification of levels of detail (LOD). The primitive list can differ depending on the LOD required. Different primitive lists will be chosen from a particular branch, depending on the current detail level. Different LOD levels can be specified for the attribute and for the class.

There can be a `Switch` nodes that selects a particular child node (from several alternative nodes) according to some parameter or variable. For example, the variable p can only take the value 1 (a sphere will be seen) or 2 (a box will be seen).

```
annotation(
 PrimitiveGeometry(
    Switch(
      name=p ,
      alternatives={
          Sphere( p={0,0,0}, radius=1.0 ),
          Box ( p1={-1,-1,-1}, p2={0,0,0})
          })))
```

There is a similar `Switch` construct in the VRML language.

7 Implementation Outline

In this section we present an outline of a future implementation of environment components necessary for integration of 3D geometry and 3D graphics with Modelica.

The structure of this implementation is shown in Figure 5.

Usually the Modelica model is supplied to the compiler, and all the annotations are ignored. In order to extract necessary information a special *pre-compiler* should be designed. It takes Modelica model with 3D geometry and graphical annotations and transforms it to labeled tree representation (LTR). The labels at the top level of the tree correspond to the variable paths of the variables containing some graphical annotations. For instance. model `E` in Figure 3 corresponds to the tree in Figure 6:

The labeled tree representation (LTR) should be supported as a tree in memory. Additionally, it should be possible to write and read such structures to and from a file.

When a simulation is running, external functions can be called in order to obtain some geometry data. For instance, the current volume of some body, or collision force that currently occur between the bodies. These functions (e.g. `GetVolume(''d'')`) find the corresponding branch or set of branches in the LTR. If some variables (such as `d.b.r`) are needed during the computations, the values can be obtained from the runtime data table. This table contains the names of all available variables and their current values.

Some functions (e.g. to find the volume of a rigid body, which never changes during simulation) should be called just once, before the actual simulation starts. The result of these functions can be used in the same way as `parameters` in Modelica.

The same LTR is used when the geometry is required during visualization. Necessary variable values are fetched from the file with simulation results.

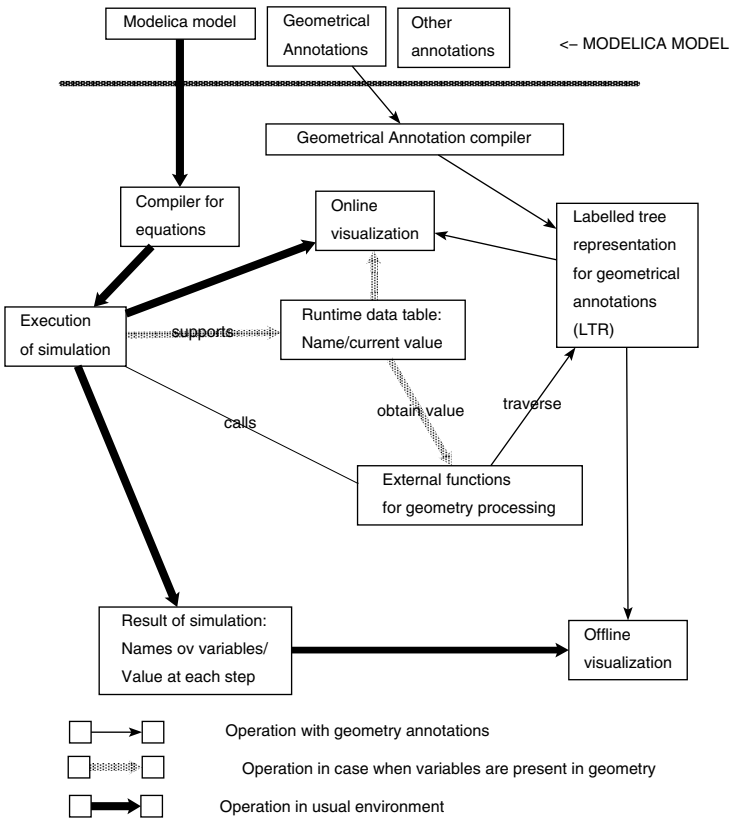


Figure 5: Implementation structure for Modelica environment working with graphical annotations.

8 Conclusions

In this report a flexible notation for geometry description is discussed. In order to obtain the necessary flexibility and short notation, some deviation from the standard Modelica syntax is necessary. This is one of the reasons in using annotations instead of classes where a new standard can easily by defined and old standards can be extended. The proposed standard requires implementation of some extra tools, including an annotation translator. These tools do not break the current implementation of Modelica [4, 7] and furthermore require no essential changes in it. Therefore it can be considered as a pure Modelica extension.

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] AutoDesk, *AutoCad documentation. DXF Format*, http://www.autodesk.com

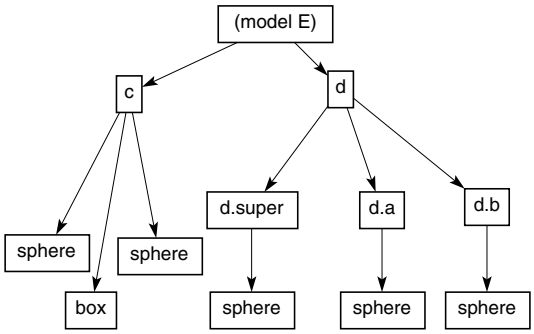


Figure 6: Labeled tree representation for geometric annotations.

[3] Cycore AB, *Cult 3D Home Page*, http://www.cult3d.com

[4] Hilding Elmqvist, Dag Brück, Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual, Version 4.0*, from Dynasim AB, Research Park Ideon, Lund, Sweden, http://www.dynasim.se

[5] Emacs, http://www.emacs.org

[6] Vadim Engelson, *Tools for Design, Interactive Simulation and Visualization for Dynamic Analysis of Mechanical Models*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 007. Available at: http://www.ep.liu.se/ea/cis/2000/007/

[7] MathCore AB, *MathModelica* , software tool for modeling, simulation, and visualization. Available from MathCore AB, http://www.mathcore.com.

[8] Modelica Design Group, *Modelica WWW site*, http://www.modelica.org

[9] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial and Rationale. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[10] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[11] SGI, *OpenGL Web Site*, http://reality.sgi.com/opengl/

[12] VRML Consortium, *VRML Repository*, http://www.sdsc.edu/vrml/

Paper 10

Integration of Collision Detection with Multibody System Library in Modelica

Vadim Engelson, PELAB, IDA, Linköping University

Abstract

Collision detection and response is one of the most difficult areas in simulation of multibody systems. Two known approaches, the impulse-based method and the force-based (penalty) method, can be applied for multibody simulation in Modelica. The impulse-based method requires instantaneous modification of some variables, but such modification is not always possible in Modelica. The force-based method leads to stiff ODE, which can be handled by solvers used with Modelica. We suggest a new way to express the penalty coefficients. The force-based method, however, requires computation of penetration depth which is time-consuming.

We suggest a combined method for computation of penetration depth.

Calling external functions is a preferable method integrate collision detection algorithms with in practical physical models, since body geometry is stored exernally. We describe an interface with collision detection tool SOLID.

Keywords: Mechanical modeling, Simulation, Modelica, Collision Detection

1 Introduction

In mechanical systems certain machine elements usually interact with each other. When a mathematical model of such a system is designed, the interactions between the parts can be divided into the two following categories:

- *Mechanical joints* are used for definition of permanent constraints of motion.
- *Mechanical contacts* are almost instantaneous, typically short-time interactions caused by non-penetration contact forces arising between the bodies in the model. The forces occur when the body surfaces touch each other.

  Two major phenomena occur in mechanical contacts:

  - friction contacts (causing static or dynamic friction forces) and
  - collision contacts (causing collision response forces).

Computation of contact forces is a difficult task. The bodies might move in a complicated way, and they might have a complex geometry. When at some instant they touch each other, penetration of the bodies should be prevented. There is a tradeoff between efficiency and accuracy. One of the goals of Modelica simulations is interactivity, therefore the computation should have at least the same speed as the processes in the real mechanisms. There exist accurate methods for contact force computation based on finite element methods and other methods using subdivision of bodies into very small fragments. Currently these cannot run at interactive speed.

The most accurate and realistic methods used in mechanical simulations of contacts available in the area of mechanical analysis called tribology. This theory contains equations that take hydrodynamic properties of the lubricant that occurs between bodies into account. Contact forces appear already when bodies have some distance from each other. These forces are caused by compression and decompression of the lubricant.

One of the difficulties with the computation of contact forces is the variety of surface geometries. For certain kinds surfaces (plane, spline of 2nd order) there exist collision checking methods and approaches to generalize the forces that arise. Such systems in general use higher order polynomials to compute forces from geometrical relations. Theory and applications of contacts situations are discussed in [13].

An accurate method for contact (e.g. a FEM method) force computation requires that the surfaces of two colliding bodies are covered by a mesh, and that the relevant contact force is computed for each point on the mesh. The resulting force is then found by integrating of all the forces acting on the contact surface. Experiments with contact computation of rolling bearing models [4] show that this method is accurate, but requires tremendous computing resources.

However, many simulation applications do not require extreme accuracy. Their goal is to demonstrate qualitative characteristics of the behavior, not numerical ones. Therefore in many cases additional assumptions are taken into account, that may reduce accuracy, but provide high simulation speed. Often different assumptions lead to different computation methods but to the same (or nearly the same) computation results. In that case this it makes no major difference for the application what assumptions and methods were used.

The Multibody System (MBS) library in Modelica is used for mechanical model simulation. Currently this library supports simulation of models with rigid bodies and joints. Friction occurring in the joints can optionally be taken into account in the models. However, collision detection and collision response is not supported. As a result, the mechanical parts may penetrate each other freely during simulation. This significantly decrease the realism of mechanical simulations. The simulation results might even be wrong.

The goal of this report is to identify the ways to add collision response to mechanical simulation models based on MBS.

Our approach to modeling collisions is based on combining several components. In order to do that, collisions between bodies should be detected, collision

response should be evaluated, and this response should some way affect the simulation. Therefore the following basic components of collision simulation models have been identified:

- Mechanical models use certain classes describing physical bodies. Mechanical models including collision response force should extend these classes to describe colliding physical bodies. Such bodies should be distinguished from non-colliding bodies.

- There should be routine that can detect collisions in the simulated mechanisms and return detailed information regarding collision parameters, such as collision points and their velocities.

- A special routine should calculate the collision response from collision parameters. The collision response is calculated as force and torque applied to the point of collision at the colliding body (or bodies). Alternately, this routine might change the velocity of the bodies or some other model variables. This component seems to us the most problematic and controversial, since there exist many approaches to collision response computation which require different input information and may produce quite different numerical results.

Each of these three components should have an interface that allows replacing its implementation without doing major redesign of the other components. In particular, the collision detection package as well as the force computation functions shioukd be easily replaceable when necessary.

The overall architecture of mechanical simulation models with collision detection and response is presented in Figure 1.

Thiss report reflects ongoing research and development. Therefore some fragments of work related to this report will be done in the future. The structure of the report and the relation between its parts is shown in Figure 2.

The two major collision models used in the simulation are impulse-based and force-based models. Both assume that the bodies are rigid. The impulse-based approach uses collision impulses between the bodies (Section 2). The force-based approach computes a non-penetration force (Section 3). A traditional variant of the force-based approach is the penalty method which assumes that bodies in contact behave like objects connected by a spring. We consider these methods in application to Modelica. A new method for computing force from penetration depth is given in Section 4. There is a number of common properties of collision detection tools (Section 5). A the particular tool, SOLID, is used for finding the penetration depth (Section 6). Section 7 illustrates how the routines for computing the forces are integrated with mechanical models in Modelica.

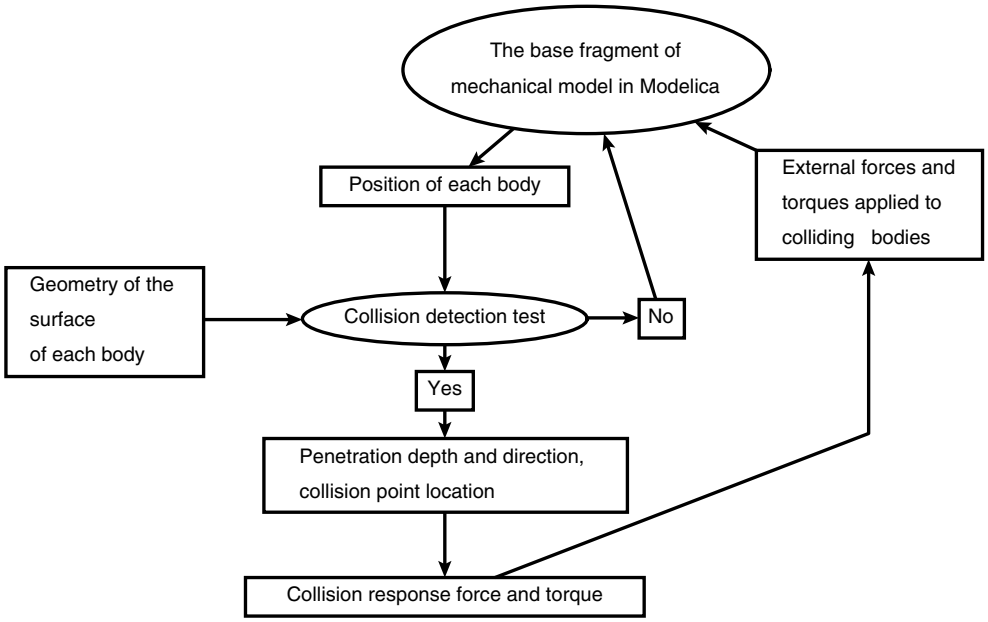


Figure 1: The overall architecture of mechanical simulation models for computing collision detection and response.

2 Impulse Model

A standard way of handling collisions in mechanics is based on the linear momentum preservation law.

2.1 Impulse and Velocity Equations

The following notation is used in the equations below:

- m_a, m_b – masses of bodies A and B;
- v_a, v_b – velocity vectors of bodies A and B before the collision; v'_a, v'_b – the velocity vectors of the bodies after the collision;

The movement of a body is described as movement of its center of mass (linear velocity) and rotation of the body around its center of mass (angular velocity). The linear momentum is the mass multiplied by the linear velocity. The total linear momentum of a system consisting of A and B is preserved if there is no external impact. This is also true for colliding point masses and it can be expressed by the equation

$$m_a v_a + m_b v_b = m_a v'_a + m_b v'_b.$$

In order to describe the collision further, we need to build a collision plane and the vector $\vec{n}$ which is normal to the collision plane.

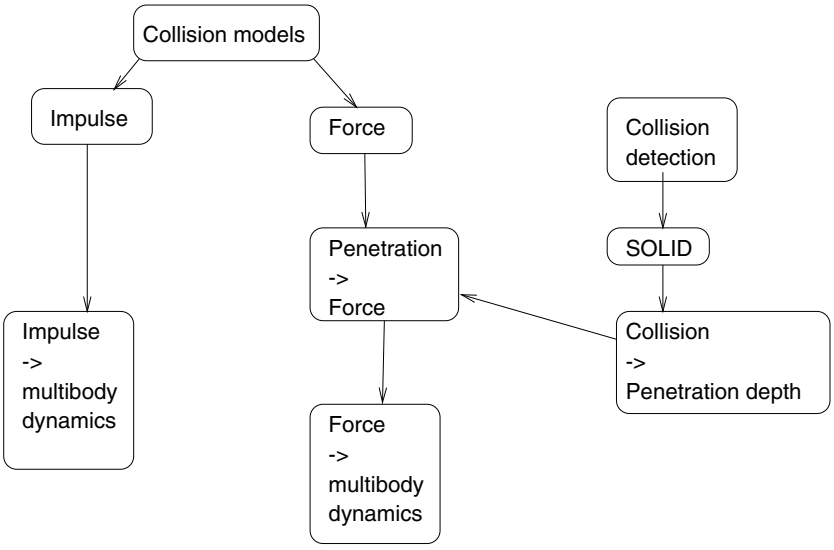


Figure 2: The structure of the report.

The change of the projections of the velocities of point masses on $\vec{n}$ can be expressed using the restitution coefficient ε:

$$\varepsilon = \frac{v'_a - v'_b}{v_b - v_a}.$$

In the case of absolutely elastic collision $\varepsilon = 1$. In the case of absolutely non-elastic (completely damped) collision $\varepsilon = 0$. Actual physical values of ε always belong to the interval $0 < \varepsilon < 1$. Experimental measurements show that the restitution coefficient ε depends mainly on material properties of bodies A and B.

The object B can be rigidly attached to the inertial system. It can be a static obstacle, such as a ground plane or a wall attached to the ground. In this case ($m_b \to \infty, v_b = 0$), the equations for point masses are simplified, and they yield the following equation:

$$v'_a = -\varepsilon v_a.$$

The simplest case of one-dimensional collision is shown in Figure 3. The case of two- or three-dimensional collision is similar. The trajectory of collision of point masses in 2D is shown in Figure 4.

However, when bodies collide (see Figure 5), the collision points differ from the center of mass, which should be taken into account.

Several assumptions (see e.g. [9, 15]) are taken into account when the law of linear momentum preservation is used for physics-based simulation:

- Collision duration is negligible.

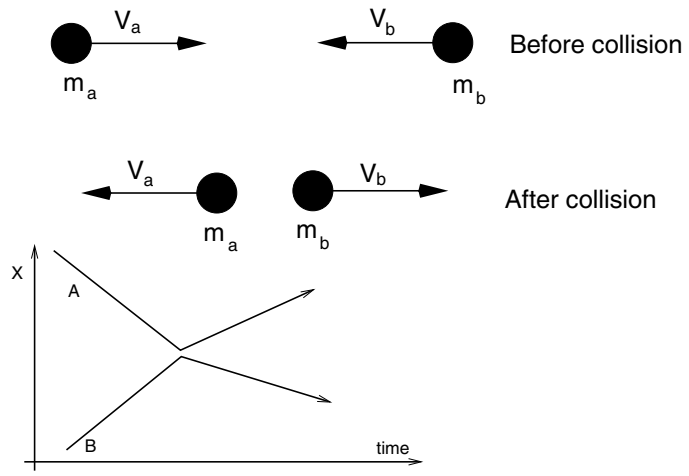


Figure 3: One-dimensional collision of two point masses A and B.

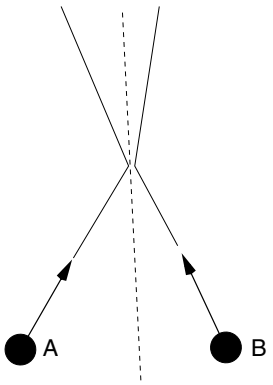


Figure 4: Two-dimensional collision of two point masses A and B. The collision plane is shown as a dashed line.

- There exists just one point of collision.
- The colliding bodies don't move during the collision.
- No other forces than collision force act on the bodies.
- The impulse gives instantaneous change to the linear and angular velocities of the colliding objects.

A derivation of the equations we use here is available in [13]. Assuming that p_a and p_b are points of collisions, x_a and x_b are positions of the centers of mass, and ω is the angular velocity of the body, the velocities of the points can be found as

$$\dot{p_a} = v_a + \omega_a \times (p_a - x_a) \text{ and } \dot{p_b} = v_b + \omega_b \times (p_b - x_b),$$

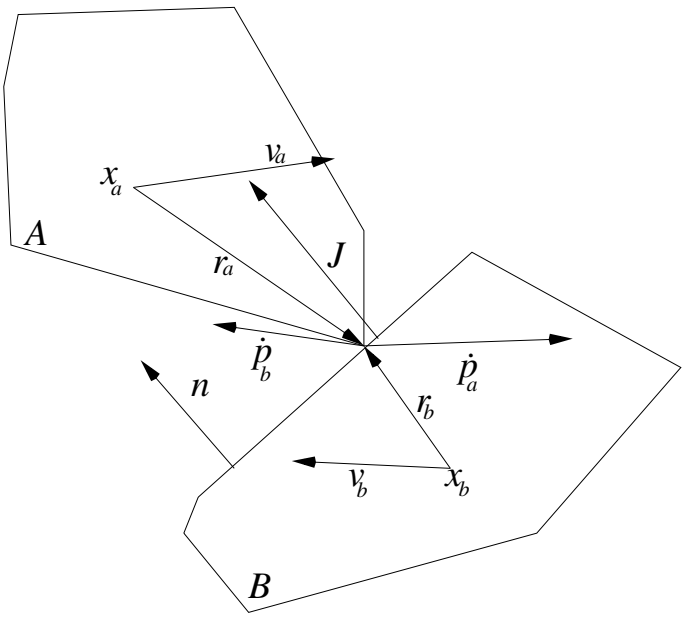


Figure 5: Two dimensional collision of bodies A and B.

the relative velocity is defined as $v_{rel} = \vec{n} \cdot (\dot{p_a} - \dot{p_b})$,
and the restitution coefficient is expressed as $\varepsilon = -v'_{rel}/v_{rel}$.
Also, the angular momentum of two colliding bodies is preserved:

$$I_a\omega_a + I_b\omega_b = I_a\omega'_a + I_b\omega'_b$$

where I is inertia tensor.

The collision impulse J is the change of momentum. Knowing the velocities of the bodies before the collision, the point of collision, and the restitution coefficient we can find the velocities after the collision. These velocities can for each body be expressed from

$$(v'_a - v_a)m_a = J \text{ and } (v'_b - v_b)m_b = -J$$

$$(\omega'_a - \omega_a)I_a = r_a \times J \text{ and } (\omega'_b - \omega_b)I_b = r_b \times J$$

where $r_a = p_a - x_a$ is position of the collision point relative to the center of mass of body A.

In order to find J some algebraic manipulations are needed. If we ignore the angular velocities it can be expressed as

$$J = -(1 + \varepsilon)v_{rel}/(m_a^{-1} + m_b^{-1})$$

But if we care about correct angular velocities, the expression is more complex [13]:

$$J = \frac{-(1+\varepsilon)v_{rel}}{m_a^{-1} + m_b^{-1} + \vec{n} \cdot (I_a^{-1}(r_a \times \vec{n})) \times r_a + \vec{n} \cdot (I_b^{-1}(r_b \times \vec{n})) \times r_b}.$$

2.2 Simulations Using Impulse-Based Model

In order to use the impulse-based approach,the computation model should assume that collision takes place if the distance between the closest features of two bodies is less than some threshold T.

Some systems [15] are constructed so that

- The exact time of collision (if it happens at some instant between two time frames) is not computed.
- Penetration is avoided at all by choosing a safe time step (Δt) and estimating the maximum speed (v_{max}), so that $T > v_{max}\Delta t$.

Nevertheless, if penetration occurs (because of erroneous estimations) the time step should be changed and some backtracking in the solution process must be performed. Of course, this requires fine-grained control over the differential equation solver that is used for numerical simulation.

Other systems [13] search for exact time of contact, using the method of bisection. The time interval normally used in the solver is divided to two, and the contact is searched in one of two time intervals. This subdivision continues until some tolerance bound is reached. This requires even more control over the solver.

The collision plane for disjoint objects is defined as follows. We assume that bodies consist of such features as vertices, edges and faces. If one of the colliding features is a face, this face is used as a collision plane. If vertices or edges are the closest features, the shortest line between them is used as a normal to the collision plane.

The situation when a body is resting on a surface is treated as a constraint (giving additional equation) or as series of micro-collisions (as proposed in [9]).

Since the velocity is not continuous in the impulse-based model, it is not very appropriate for use with traditional ODE solvers. Actually, the solver should stop at the instant of collision and continue with the new velocity, as it is done, for instance, in Modelica.

An alternative approach is based on writing a system of non-differentiable equations and applying a Newton method specially devised for such equations [12]. This method have successfully been applied for body impact with friction by Johansson and Klarbring [5].

Variations of the same impulse-based model [9] can describe rolling, sliding, resting and bouncing. This mathematical model has been combined by Zhang [15] with collision detection algorithm I_COLLIDE [6] to form an ODE-based simulation tool. This model, however, assumes that between the collisions the objects have ballistic trajectories i.e. they are not constrained by revolute and translational joints.

2.3 Impulse-based Approach and Modelica

Modelica has capabilities for modeling the impulse-based collision response algorithm.

The Modelica language, like some other modeling and simulation tools, has support for instantaneous change of some variable values during simulation. This change might happen at special simulation steps, called *events*. At these events the continuous integration (i.e. the process of solution of the system of differential and algebraic equations) stops, specific equations valid for this event are solved, variables obtain new values, and then the integration continues. This is the way Modelica carries out *hybrid* modeling, where continuous and discrete behavior of the system are described in the same model.

In a simple mechanical model, described in Modelica, it is possible to change the velocity variable `v` when some condition triggers an event.

The `when` statement may contain a call to collision detection routine. When collision happens, Modelica event occurs. The angular and linear velocity of bodies can be measured and instantly changed at specific points in time (events) during the simulation. In this case the integration stops, the initial conditions for velocity are changed, but all other variables keep their values as they were before the collision. The continuation of the simulation is completely consistent from the point of view of the MBS library.

We illustrate this possibility with two examples: a bouncing ball and a simplified pendulum.

2.4 Bouncing Ball Example

The bouncing ball example illustrates a model constructed using explicit equations defining 1D movement and collision.

The model below describes a bouncing ball falling down from the height 10 meters. When the ball reaches the ground, the condition `x<0` becomes true, an event is triggered, the exact instant of this event is automatically found, and the variable `v` gets the new value `-`ε`*v`. The ball rises from the ground and falls down again. The plot of the height `x` is shown in Figure 6.

```
model BouncingBall "bouncing ball, 1-D model"
  Real v "velocity";
  Real x (start=10) "height";
  parameter Real g=9.8;
  parameter Real eps=0.8 "restitution" ;
equation
  der(v)=-9.8;  // permanent acceleration
  der(x)=v;      // velocity
  when (x<0)  then // when ball crosses the ground level
    reinit(v,-eps*v); // the velocity is re-initialized
  end when;
end   bouncingBall;
```

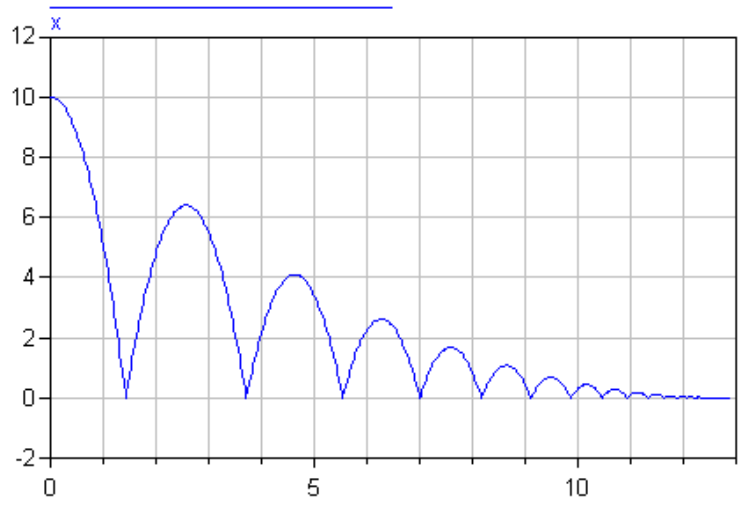


Figure 6: The height of the bouncing ball computed using the impulse model.

2.5 Colliding Pendulum Example

It is also possible to use the **`when`** and **`reinit`** in the MBS library context.

The pendulum example (see Figure 7) illustrates use of the MBS library for a model with impulse-based collision response. The state variable `q` in this model is the angle of the revolute joint of the pendulum, and the model makes an assumption that the velocity (that changes instantaneously at the moment of collision) is proportional to the angular velocity of the change of this state variable. This assumption is wrong in more complicated case.

The trajectory of the pendulum end is depicted in Figure 8. The Change of the angle of rotation $R1.q$ is shown in Figure 9.

```
model Pendulum
  "Impulse-based model of collision of pendulum end with an obstacle"
  Inertial i;
  RevoluteS R1(n={0,0,1},q(start=1));
  BodyBase M1 (m=50,rCM={0,0.5,0},I={0,0,0;0,0,0;0,0,0});
  Bar B (r={0,1,0});
  parameter Real restitution=0.5;
  parameter Real obstacle_x = 0.5 " x coordinate of obstacle";
equation
  connect(i.b, R1.a);
  connect(R1.b, M1.a);
  connect(R1.b, B.a);
  when (B.r0b[1]>obstacle_x) then
    reinit(R1.qd,-restitution*R1.qd);
    // A problem is how to propagate this change to velocity of B
  end when;
```

```
end Pendulum;
```

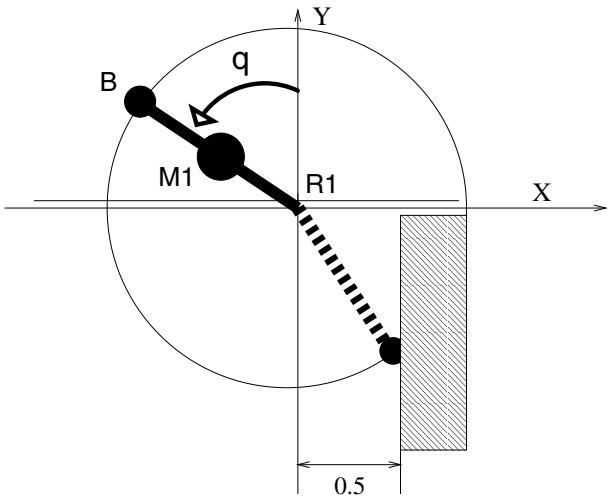


Figure 7: Simple pendulum colliding with an obstacle at point $Obstacle_x = 0.5$.

.

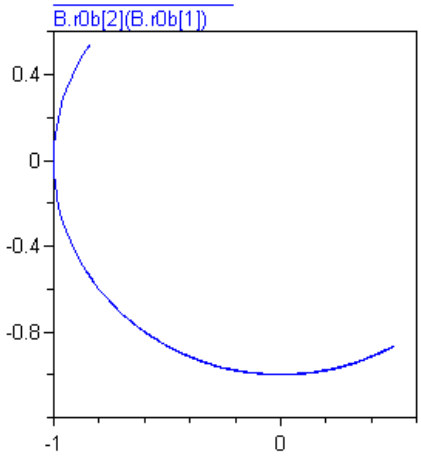


Figure 8: Trajectory of the endpoint of a simple pendulum colliding with an obstacle at point $Obstacle_x = 0.5$

2.6 Problem of Non-State Variables

There is one major difficulty in applying the impulse-based model to mechanisms with rotating parts.

In order to apply the impulse-based approach, the velocity should be changed, according to the rules, $v := g(v)$, where g can be a complicated function depending

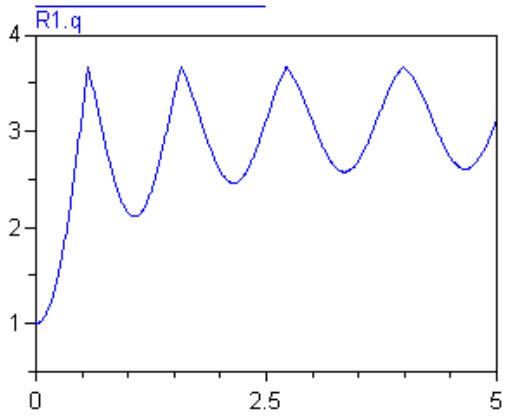


Figure 9: Angle of rotation of a simple pendulum colliding with an obstacle. The restitution is $\varepsilon = 0.5$. The collision happens when the angle reaches `R1.q`$=\pi + \arcsin(Obstacle_x) \approx 3.66$.

on collision details. Modelica allows only instantaneous change of state variables. In the MBS models, that include rotating bodies, the linear velocity is not a state variable. It is a dependent variable that can be computed from state variables. There are only two ways out:

- Restructure the model so that velocities become state variables, and apply impulse-based approach to this model.
- Re-initialize the state variables q such way ($q := f(q)$) that corresponding velocities change ($v- > g(v)$) according to the impulse-based approach. The difficulty is to find the function f from g.

2.7 Restructuring the Model of Colliding Double Pendulum Example

It is possible to use kinematic loop in order to restructure the double pendulum example (see Figure 10).

Model can be restructured in some case, but this approach is not general enough. We can use kinematic loops, i.e. loops in the structure of the multibody model. In particular, double pendulum model with 2 revolute joints (containing 2 state variables) (Figure 10(a)) can be replaced by closed kinematic loop with 2 prismatic (with state variables), one rotational joints for cutting kinematic loops and 2 revolute joints (without state variables) (Figure 10(b)) .

Both models (a) and (b) result in the same motion when no collisions occur, but they use different state variables. The Modelica model for the construction in Figure 10(b) is given below. The connection diagram of this model is shown in Figure 11.

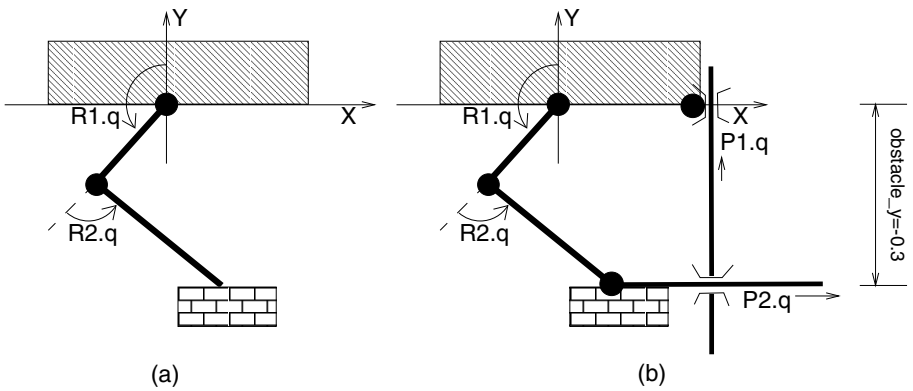


Figure 10: Double pendulum (a) combined with additional joints to form a kinematic loop(b).

```
model Pendulum
  parameter Real obstacle_y=-0.3;
  output Real x;
  output  Real y;
  output  Real xv;
  output  Real yv;
  output  Real dist;
  parameter Real restitution=0.5;
  BodyV M1 (mass=50,r=\{0.5,0,0\},
    Shape="box", Size=\{1,0.1,0.1\});
  BodyV M2 (mass=50,r=\{0.5,0,0\},
    Shape="box", Size=\{1,0.1,0.1\});
  RevoluteCut2D rc;
  // some objects presented in the diagram are omitted here
equation
  x=P1.r0b[1];
  y=P2.r0b[2];
  xv=P1.qd;
  yv=P2.qd;
  dist=obstacle_y-y; // positive when collided.
  when (dist>0) then
    reinit( P2.qd ,- restitution *  P2.qd);
  end when;
  // connections presented in the diagram are omitted here
end Pendulum;
```

When simulated and visualized this model demonstrates a correct bouncing behaviour.

The difficulties with this approach are

- Each bodies in the multibody system requires a special kinematic loop, which leads to huge amount of additional objects.
- It is difficult to find correct components for the loop, since all potential freedom degrees should be taken into account and analyzed.
- Computations with kinematic loops easily reach singularity points where solvers cannot find an appropriate solution (in the case above it happens

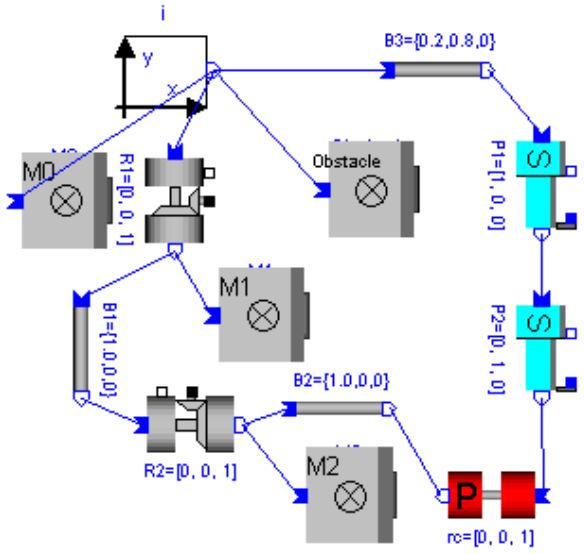


Figure 11: Double pendulum model with kinematic loop, graphical presentation.

when the angle `R2.q` crosses 0).

2.8 Using Dependencies Between State Variables and Body Velocities

Computing these dependencies can be difficult. It should be noted that in a tree-like structure of a multibody system the collision of a body in one node can cause the change of velocities in all joints between this body and the ”root” of the tree. The technique for this propagation of velocities is developed in [10] as well as [11] (p. 146). It is a sequential algorithm based on sending three ”test impulses” through the links of the multibody. It seems hard to implement the algorithm in the connection-based MBS model.

2.9 Limitations of the Impulse-Based Method in Modelica Models

MBS-based models might contain static objects, free floating objects, and multi-bodies (objects consisting of several bodies connected by revolute and prismatic joints).

The impulse-based approach can easily be used in MBS-based models if impact of collision on the multibodies is negligible. For instance, systems of free floating bodies, colliding with the walls of some volume, as well as robot manipulators that may collide and affect the floating bodies, but such collision does not affect the robot. This is limit the number of applications for dynamic analysis. If an appropriate solution to the problems mentioned in Sections 2.8 and 2.7 is found, arbitrary MBS models can take collisions into account.

3 The Force Model of Collision

Our approach to collision processing in mechanical systems is based on the force and torque model of collision. We assume that colliding bodies penetrate that and a separation force created is caused by this penetration. This force tries to prevent further penetration and to separate the colliding bodies.

It is well known that during the collision a relatively large force occurs between the two colliding bodies for a very short period of time. The value and the direction of the force can be computed approximately computed for each simulation time step. The following properties of the collision force should be taken into account:

- The collision force and collision torque acting on an object is zero if the object does not collide.
- Between the start of collision and the end of collision a force occurs to prevent further penetration.
- If an ideal collision is modeled (collision of points masses), the resulting velocities after the collision are given by the law of preservation of linear momentum.
- A contact force acting on a body resting on a horizontal platform compensates the gravity force applied to the body. Therefore such an object does not move (its vertical acceleration and velocity is zero).

A collision force satisfying the above properties can be found in many different ways, with different degree of accuracy. It appears that the physics of collision between elastic and homogeneous (isotropic) bodies matches the mathematical model. However this model (penalty-based collision response) is usually applied to arbitrary rigid bodies.

In practice it is important that the force is differentiable and the total mechanical energy of the system is consumed (not produced) during the collision. Some energy is transformed to the thermal energy.

In order to balance the required accuracy and available computational power of Modelica simulations, we derived the following rules for collision force computation (some of the terms are discussed in the following sections):

- The collision force acting on a body is zero if the body does not penetrate with any other body.
- If body A penetrates body B, collision forces are created to act on the objects A and B and are applied at the point of contact on each body. The force is directed so that A and B are pushed away from each other due to this force. The direction of this form corresponds to the shortest displacement that can separate the bodies.

- The magnitude of the force is proportional to the depth of penetration of A and B. This depth is the length of the shortest displacement that can separate the bodies. This corresponds to the model of spring and damper, inserted between the bodies. The bodies are rigid, but the spring and the damper are not rigid.

These rules regarding the computation of collision force contain several terms that will be further discussed, clarified and refined.

3.1 Penetration

We assume that A and B are defined using description of their surfaces which separate their inner volume from the outer world. Objects with this property are called also orientable manifolds. In practice such surfaces are described as a set of connected triangles (or arbitrary polygons). In this case the bodies are called *polytopes*. More complex surfaces can be described as spline surfaces, e.g. NURBS.

Formally, two objects A and B penetrate each other if the volume of their intersection is larger than zero. This is equivalent to the statement that at least one point on the surface of body A occurs within body B. However, in practice, the volumes are not computed. The geometry description of bodies A and B is normally considered by collision detection software as a so-called "triangle soup", i.e. a set of arbitrarily placed triangles in 3D. The bodies A and B penetrate if some of their triangles intersect[1]. In some cases the collision detection software assumes that the considered bodies are convex ones (i.e. any line between the points belonging to the body belongs to the body).

3.2 The Bodies and Their Shells

Perfectly rigid bodies do not penetrate. However, physical bodies always penetrate for a small fraction of their size in practice. When more accurate results are necessary, and more complicated computation model is used, bodies are subdivided to small fragments and penetration between these fragments is considered.

The collision methods normally assume that the depth of penetration is negligible in comparison with the size of the body.

We can also consider a body A_{shell} which is A joined with all the points on the distance less than Δ_{shell} from A (see Figure 12). Impulse based systems take the geometry of A and B into account and consider that collision takes place if A_{shell} and B_{shell} intersect. The bodies A and B can never intersect in the models using this approach. Here the value Δ_{shell} is used as a threshold.

In our force-based model, the geometry of A_{shell} and B_{shell} can be taken into account instead of A and B. Collision occurs if A_{shell} and B_{shell} intersect. The

[1] It might happen that the bodies A and B just touch each other; the volume of intersection is zero and the collision force is zero in this case.

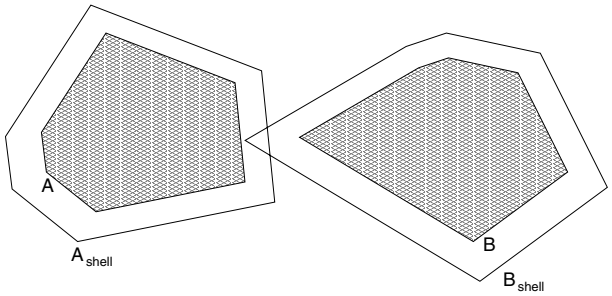


Figure 12: Penetrating shells of objects A and B.

forces generated due to collision response are so large that time when A and B intersect is negligible.

3.3 The Point of Contact

The point of contact can be defined in many different ways. The naive definition states that this is the point where two bodies touch each other the very first time (at the first instant of collision). Such a definition is only good for very short-duration contacts. If the bodies have longer contact (i.e. the separation force should be computed during several time steps), then the point of the very first contact can differ from the point of contact few steps later. An example of such behaviour is two bodies colliding and then keeping sliding contact. In this case the point of the first contact cannot be used for evaluation of contact force for the next steps.

Collision detection software packages usually do not find the point of the first contact. They just determine a point which belongs to the intersection of the objects A and B if they collide. If the objects do not collide, the closest pair of their points can be determined. Obviously, these points may become irrelevant for further computation in the case of a long duration contact between the bodies.

A good integral (average) point of collision would be the center of volume of the intersection between the bodies A and B. The geometry of an intersection can, however, be quite complex. Finding the volume is computationally intensive task. The volume cannot be in general found at all if the body geometry is stored by the collision detection software just as ”triangle soups” (since the soup is composed of a set of triangles with zero thickness, it has no volume, and intersection of two soups has no volume.) Finding the center of the volume does not help in finding the direction of the collision force.

3.4 Direction of Force

The direction of the collision force can be determined in several different ways. For a short contact it would be natural to define the direction of the collision force as opposite to the velocity of the contact point, i.e. the point of the first touch between

the bodies. However, for similar reasons as for the above mentioned approaches, this does not work well for long contacts.

For accurate determination of force direction a mesh based on colliding surfaces is constructed, and a normal vector to the surface in each mesh point is used as local force direction. The resulting force direction is then found by integrating of vectors of all the forces acting on the contact surface.

3.5 Penetration Prevention Model

In our approach the collision force is computed using the assumption that it prevents further penetration of the bodies. The computations during the contact are not dependent on the duration of the contact. The method can handle both short collision times (the collision force causes termination of collision soon after it starts) and long collision times (the collision force compensates some other forces and therefore the contact can take indefinitely long time).

If the objects A and B collide, body A forces body B out (pushes it away) from its interior. We assume that the bodies behave as elastic and isotropic medium. Therefore the direction of this force should be such that B is pushed out in the shortest possible way. Therefore the *shortest separation vector* should be chosen. The shortest separation displacement is the shortest of all displacements of body B that if applied to B, the bodies will separate.

We formulate the definition in mathematical notation:

Assume that S is a set of all vectors in R^3. If $\vec{m} \in S$, we define $B(\vec{m})$ as the body B being displaced according to the vector $\vec{m}$. The distance between the closest points of bodies A and B is $d(A, B)$; these closest points are $P_{AB} \in A$ and $P_{BA} \in B$. The set of translation vectors that can separate B from A is $S_{AB} = \{\vec{m} \in S \mid A \cap B(\vec{m}) = \emptyset\}$. The shortest separation vector $\vec{c} \in S_{AB}$ is the shortest vector in the set S_{AB}.

The separation force is applied at the corresponding closest points. The force F in direction $\vec{c}$ is applied to the body B at the point P_{BA}; it pushes out the body B from A. The opposite force $-F$ in direction $-\vec{c}$ is applied to the body A at the point P_{AB}, and pushes the body A away from B. The ways to evaluate the magnitude of F from $\vec{c}$ are discussed in the next section.

Our method works best if a single vertex (with some surrounding surface fragments) of body B penetrates in the middle of a face of body A. The forces are directed so that this vertex P_{BA} ($P_{BA} \in B$) is pushed away by the force directed as a normal to the face (see Figure 13). Note that the 2D examples are given just for illustration; the actual software works with 3D models.

The approach works well even if the center of mass of B is far away from P_{BA} (see Figure 14). In this figure body $B(c)$ is oriented the same way as body B. Note that due to the force being applied to point P_{BA} body B will actually rotate during the collision.

In the case of vertex–to–vertex collisions (see Figure 15) the algorithm computes the shortest direction, c (the vertical direction in this case) and ignores the

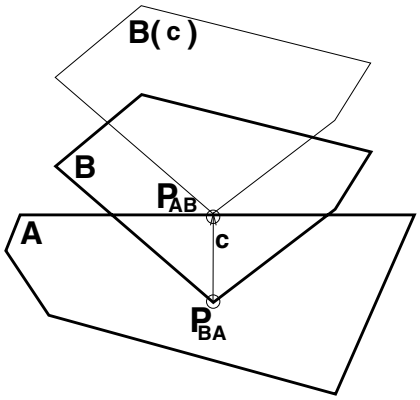


Figure 13: Collision geometry in the simple case: a single vertex of B is located within A.

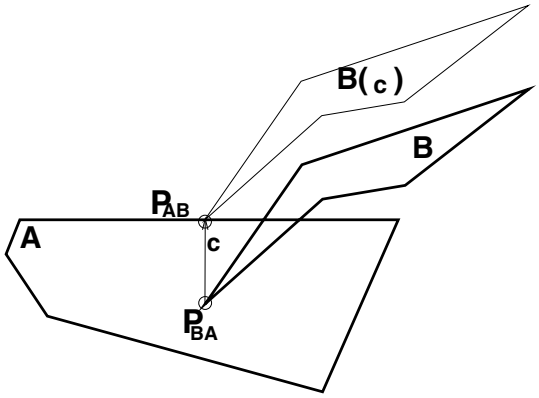


Figure 14: Collision geometry in the case of a sharp angle.

longer one (c_1). Such collisions, however, are rather rare, specially if the penetration depth is much smaller than the size of the bodies.

In the case of two–vertices–to–plane collision (a box resting on the ground) the force is applied to the deeper vertex of the box (see Figure 16). This force raises the box up, and another vertex of the box becomes the deepest one. After some interaction the box and the platform separate, or the box will be lying on the platform. The collision forces compensate the force of gravity.

4 Computation of the Force from Penetration Measurement

The source of collision forces is in the following physical phenomena. Initially, the bodies are compressed with each other and therefore deformed. This deformation causes reaction force and restitution (restoration) of the shapes. The bodies there-

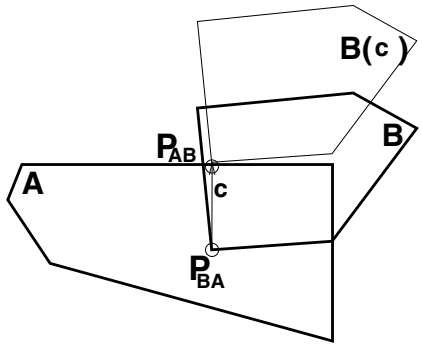


Figure 15: Collision geometry in the case of several vertices penetrating.

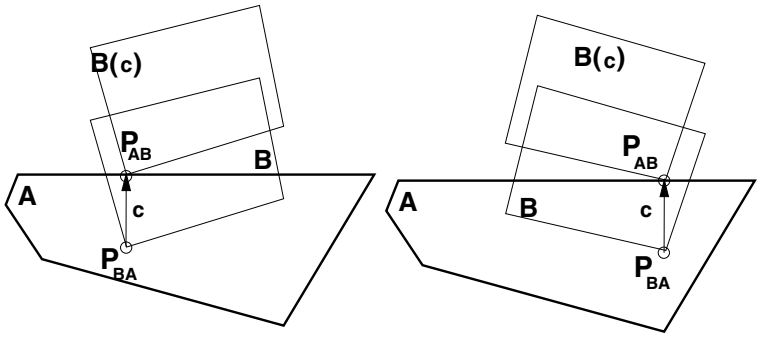


Figure 16: Collision geometry in the case of collision of two vertices and a horizontal platform.

fore separate from each other. This phenomena can be modeled in different ways. The traditional approach is to model it as a spring with a damper.

At each instant during the collision the force values should be computed. The collision duration is very small, the velocity of objects changes rapidly during the collision, and the forces needed to change the velocity can be very large.

The experiments with various approaches to the computation of the collision force F have been done in the following stages:

- An equation for F with some unknown coefficients is chosen.
- A series of simulations is performed to find the dependency between the coefficients, the body velocity before the collision, and the velocity after the collision.
- The inverse dependency between the velocities and the coefficients is computed.
- Since the velocities can be computed using the law of conservation of linear momentum the coefficients can be found.

- An equation for F with known coefficients is available for computation.

The experiments describe a collision of a point mass and a static obstacle. Therefore we do not prove that F is appropriate for collisions in all cases. However it can serve as an approximation to the force in the broader set of cases.

The formula for F can be found from the laws of dynamics. During the collision ($0 < t < \tau$) the velocity changes from $v(0) = v_0$ to $v(\tau) = v'$. The acceleration could be computed from the force divided by mass. The acceleration, however, should not be constant during the collision. The actual dependency can be expressed as

$$\int_0^\tau F(t)\,dt = m(v' - v_0)\ .$$

This definition, however, cannot be used for modeling the force in dynamic simulation since the time τ is not known at the start and during the collision. Furthermore, if some other forces are applied to the body, the collision can go over to a stable contact. In this case $\tau = +\infty$.

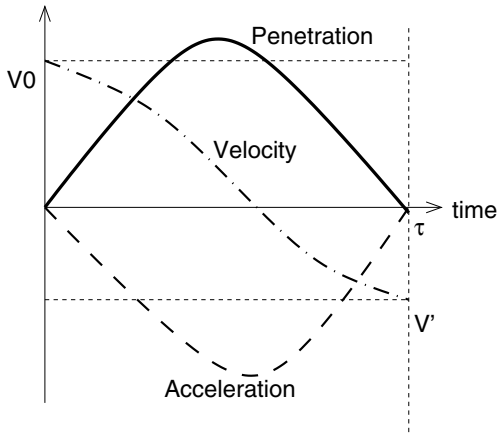


Figure 17: Collision depth, velocity and acceleration in case of elastic collision.

Figure 17 displays what might happen with penetration depth, velocity and acceleration during the collision time in case of elastic collision.

Penetration reaches some maximum value and returns to the zero value. The speed gradually changes from v_0 to v'. The acceleration and force either immediately, or gradually, achieve the minimal value, and later return to zero at the time τ.

In case of non-elastic collision (Figure 18), the penetration, velocity and acceleration become zero after the time τ. If the body has a non-elastic collision with a horizontal platform, and gravity is present, the penetration (directed downwards) is slightly more than zero, the collision force (directed upwards) is slightly below zero, which compensates the gravity force (directed downwards).

The following notation will be used below:

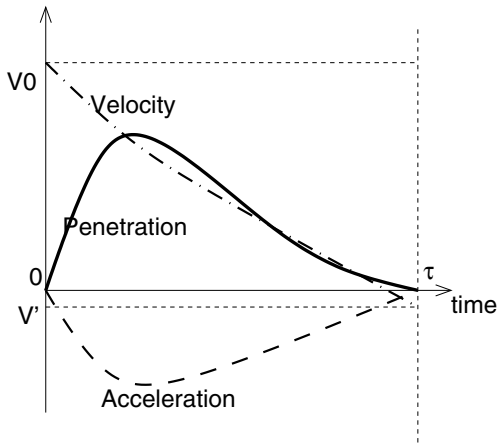


Figure 18: Collision depth, velocity and acceleration in case of non-elastic collision.

- $F(t)$ – collision force during the collision between a body and a fixed immovable obstacle.
- k, K, q – collision coefficients, they will be used in further computations.
- $x(t)$ – penetration depth.
- $v(t) = \dot{x}(t)$ – speed of change of the penetration depth.
- m – mass of the object penetrating the obstacle.

4.1 Constraints on Force Equations

The following constraints should be used in order to derive the equation for the force magnitude.

- Conditions on the start of collision (constraints C_1)
  - $F(0) = 0$ – force should be zero at the start of collision.
  - $x(0) = 0$ – the penetration depth should be zero at the start and the end of collision.
  - $v(0) = v_0 > 0$ – the actual speed is known at the start of collision.
- During the collision (C_2)
  - $F(t) < 0$ – force should all the time push the object away the obstacle.
  - $x(t) > 0$ – the penetration depth is positive (the object penetrates the obstacle).
  - $v(t)$ – the speed is reduced to zero and, probably, to negative.

- If no external force act on the colliding body (or this force is negligible), it behaves exactly according to the impulse law (C_3):
  - Collision end time is τ.
  - $F(\tau) = 0$ – force should be zero at the end of collision, and after that.
  - $x(\tau) = 0$ – the penetration depth should be zero at the end of collision.
  - $v(\tau) = v' = v_1 = -\varepsilon v_0 < 0$ – the speed at the end of collision can be predicted using the restitution coefficient $0 < \varepsilon < 1$.
- If a constant external force F_{ext} acts on the colliding body, it either behaves as above, or, in case F_{ext} is large enough, it rests on the obstacle, and collision never ends (C_4):
  - $\lim_{t\to\infty} F(t) = -F_{ext}$ – collision force should compensate the external force.
  - $\lim_{t\to\infty} x(t) = x_{rest} > 0$ – the penetration depth should stabilize on some value.
  - $\lim_{t\to\infty} v(t) = 0$ – the body rests, so it does not move anymore.

There can be many definitions for F satisfying these equations. However, in most cases, difference between them (difference between the overall effect they cause) is negligible in comparison with the effect caused by the constraints. Therefore definition for F might have or might have not sound motivation from the physical point of view.

4.2 Definition of the Collision Force Using Polynomial of Time.

The simplest kind of expression that could satisfy the constraints C_1 and C_3 is a polynomial.

Initially we can assume that $v(t)$ is a polynomial (of t) of order 3. In this case

$$\begin{aligned} x(t) &= a_0 + t\,a_1 + t^2\,a_2 + t^3\,a_3 + t^4\,a_4; \\ Solve[&\{ x(0) = 0, x'(0) = v_0, x''(0) = 0, \\ x &(\tau) = 0, x'(\tau) = v_1, x''(\tau) = 0\}, \\ &\{ v_1, a_4, a_3, a_2, a_1, a_0, \tau\} \\ &] \end{aligned}$$

The Mathematica solver finds one solution only. In this solution $v_1 = -v_0$.

Therefore we should assume that $v(t)$ is a polynomial (of t) of order 4. In this case the acceleration is a third order polynomial, and penetration depth is a fifth order polynomial. Lower order polynomials

The equation system is

$$\begin{array}{rcl} x[t\_] & = & a_0 + t\,a_1 + t^2\,a_2 + t^3\,a_3 + t^4\,a_4 + t^5\,a_5; \\ Solve[& \{ & x(0) = 0, x'(0) = v_0, x''(0) = 0, \\ & x & (\tau) = 0, x'(\tau) = v_1, x''(\tau) = 0\}, \\ & \{ & a_5, a_4, a_3, a_2, a_1, a_0\} \\ &] & \end{array}$$

The solution is

$$x(t) = t\,v_0 - \frac{3\,t^5\ (v_0 + v_1)}{\tau^4} - \frac{2\,t^3\ (3\,v_0 + 2\,v_1)}{\tau^2} + \frac{t^4\ (8\,v_0 + 7\,v_1)}{\tau^3}$$

The force can be found from the acceleration expression:

$$F/m = \ddot{x}(t) = \frac{-60\,t^3\ (v_0 + v_1)}{\tau^4} - \frac{12\,t\ (3\,v_0 + 2\,v_1)}{\tau^2} + \frac{12\,t^2\ (8\,v_0 + 7\,v_1)}{\tau^3}$$

Given collision duration τ as well as velocity v_0 before the collision and v_1 after the collision the simulation model can apply the appropriate collision force $F = m\ddot{x}$. However this approach cannot work for bodies resting on the obstacle (condition C_4) since for any polynomial $x(t)$ goes to infinity and it is impossible to have $\lim_{t\to\infty} x(t) = x_{rest}$

4.3 Linear Collision Force Model of the First Order

Traditional penalty methods are based on a linear dependency between collision force and the penetration depth. Therefore the collision model considered assuming that the force F depends on $x(t)$ only:

$$F(t) \quad = \quad \begin{cases} -Kx(t), & \text{if } x(t) > 0 \\ 0, & \text{if } x(t) \le 0 \end{cases}$$

where K is a positive constant, called penalty coefficient. Physically this corresponds to a stiff spring, temporarily placed between the objects during the collision. The expression $-Kx(t)$ corresponds to an ideal spring. The expression contains $x(t)$ in the first power only, therefore this method is called the linear model of the first order.

For brevity of the solution, K might be replaced by another positive constant, $q^2 m$. In this case, taking into account the constraints C_1 and C_3, the system of equations (assuming that no external forces are acting on the colliding bodies) appears as follows:

$$\begin{array}{rcl} F(t) & = & -q^2 m x(t) \\ F(t) & = & m\ddot{x}(t) \\ \dot{x}(0) & = & v_0 \\ x(0) & = & 0 \end{array}$$

This results in the equation

$$\ddot{x} + q^2 x = 0$$

which has a solution

$$x(t) = \frac{v_0 \sin qt}{q}$$

The velocity is

$$\dot{x}(t) = \frac{v_0 q \cos qt}{q}$$

When the collision ends (i.e. $x(t) = 0$ giving $qt = \pi$) the velocity $\dot{x}(\pi/q)$ is equal to $-v_0$.

This means that resulting velocity does not depend on the mass, and does not depend on the penalty coefficient in front of $x(t)$.

But what we need is $\dot{x} = v' = -ev_0$

Therefore the simple penalty-based model above is not good in the general case and another model should be chosen.

4.4 Collision Force Model Based on Position

It can be shown that for any collision force model that is based in position only, the purely elastic contact can be modeled only We consider the equation

$$\ddot{x} + p(x) = 0$$

where p is positive during the collision. We denote $\dot{x} = y(x)$. This results in

$$\ddot{x} = \frac{dy}{dt} = \frac{dy}{dx}\frac{dx}{dt} = \dot{x}\frac{dy}{dx} = y\frac{dy}{dx} .$$

The initial equation can be rewritten now as

$$y\frac{dy}{dx} + p(x) = 0 .$$

If all parts of the equation are integrated, the result is (C is unknown constant)

$$\int y\, dy + \int_0^x p(u)\, du = C .$$

The first component is transformed:

$$\int y\, dy = y^2/2 = \dot{x}^2/2 .$$

We know that at the start of collision $x(0) = 0$ and $\dot{x}(0) = v_0$. The equation should be valid at $t = 0$, and substitution gives

$$v_0^2/2 + \int_0^0 p(u)\, du = C .$$

Now C is found as $C = v_0^2/2$. The equation with the replaced C is

$$\dot{x}^2/2 + \int_0^{x(t)} p(u)\,du = v_0^2/2\ .$$

We know that at the end of collision $t = \tau$, and $x(\tau) = 0$. Therefore the substitution gives:

$$\dot{x}^2(\tau)/2 + \int_0^{x(\tau)} p(u)\,du = v_0^2/2\ ,$$

or simply $\dot{x}^2(\tau) = v_0^2$. From we can conclude that either $\dot{x}(\tau) = v_0 = \dot{x}(0)$ or $-\dot{x}(\tau) = v_0 = \dot{x}(0)$. But we know that p is positive, therefore $\ddot{x}(t) = -p(x(t)) < 0$. We conclude that $\dot{x}$ decrease and since $\tau > 0$ and $\dot{x}(0) > 0$, the only equation valid is $-\dot{x}(\tau) = v_0 = \dot{x}(0)$. Therefore $v' = -v_0$, which means that collision was purely elastic ($\varepsilon = 1$).

4.5 Collision Force Model Based on Spring and Damping

Since the definitions for collision force discussed above are not good for our collision model, some other variable should be taken into account. The goal is to reduce the magnitude of velocity at the end of collision. Assuming that the collision takes several seconds, the momentum should be smaller during the last second of the collision, and larger at the first second. Then the magnitude of velocity at the end of collision can be reduced. Therefore some factor is needed that gradually changes during the collision.

There are several variables that can be used in new definition of collision force. However, only t (time after the start of collision) and $\dot{x}$ (velocity of the body) are gradually changing from one value to another during the time of collision.

The following discussion explains the choice of equation for F in more details. In this discussion it makes no difference whether t or $\dot{x}$ is chosen as additional variable.

First, consider what can be done with the original expression $F = -Kx$. We know that $F(0) = F(\tau) = 0$. The velocities differ: $\dot{x}(0) \neq \dot{x}(\tau)$. Therefore the velocity (or any expression based on velocity) cannot be added to $-Kx$.

Another operation that can be applied to $-Kx$ is multiplication. Our hypothesis is that function H is applied to $\dot{x}$ and the result is multiplied by $-Kx$, i.e. $F(t) = -KH(\dot{x})x$.

The equation $F = -1Kx$ produces correct F in the case when the restitution coefficient is $e = 1$. This function H should be equal to 1 in the case of $e = 1$. Therefore it is convenient to represent $H(u)$ as $1 + G(u)$. But H is a function of velocity, therefore (if we choose the simplest alternative) $H = 1 + k\dot{x}$. The complete expression is $F(t) = -(1 + k\dot{x})Kx$. Since K is a constant and body mass m is a constant it is possible replace K by Km (for brevity of the solution). The equation is then $F(t) = -(1 + k\dot{x})Kmx$.

This equation is taken as work hypothesis and it is discussed further.

4.6 Modeling Collision Force as A Function of Penetration Depth and Time

The definition for F using another argument variable (t instead of $\dot{x}$) is $F(t) = -(1+kt)Kx$.

Various values of k, K, v_0 (in interval $\pm 10^{-3} ... \pm 10^{+3}$) have been tested using Mathematica, but none of them allows to control v' such way that $v' = -ev_0$ for desired e.

Furthermore, this equation cannot satisfy the constraint C_4, since $\lim_{t\to\infty} F$ cannot have finite limit.

4.7 Properties of Collision Force Model Based on Spring and Damper

The model can be expressed as a system of equations:

$$\begin{aligned} F(t) &= \begin{cases} -(1+k\dot{x}(t))Kmx(t), & \text{if } x(t) > 0 \\ 0, & \text{if } x(t) \le 0 \end{cases} \\ F(t) &= m\ddot{x}(t) \\ \dot{x}(0) &= v_0 \\ x(0) &= 0 \end{aligned}$$

The first two equations are reduced to $\ddot{x} = \text{If}[x > 0, -(1+k\dot{x}(t))Kx(t), 0]$.

The function $G(\dot{x}) = k\dot{x}$ is called a damping factor. In practice it is often chosen as a linear function. However in order to provide differentiability of F other functions are used too. The topic of our future research is to define a such smooth function F that Modelica solvers can use it as an external function during continuous integration.

The equation above can be solved numerically by Mathematica. The values of v_0, K, k have been chosen and $v(t_{end})$ was analyzed, where t_{end} is some point in time after the collision is finished.

When $v_0 = 1$ and $k = 1$ the resulting velocity v' changes this way: if $K = 1$, $v' = -0.593628$; if $K = 10^{-3}$, $v' = -0.593625$; if $K = 10^{6}$, $v' = -0.593627$. Similar effect appear for other values of v_0 and k. Therefore v' almost does not depend on K. In the further discussion we assume that $K = 1$.

The resulting restitution coefficient can be computed from the solution of the equation as $\varepsilon = -v'/v_0$. How can we find an appropriate k depending on the velocity at the start of collision v_0 and known restitution coefficient of materials ε ?

First we notice that ε resulting from the solution does not change if we multiply v_0 by some number and divide k by the same number.

Proof: Assume x(t) is a solution of the system. Consider $y(t) = px(t)$ (where p is a constant). Then $\dot{y} = p\dot{x}$, therefore $\dot{y}(0) = p\dot{x}(0) = pv_0$ and $\dot{y}(\tau) = p\dot{x}(\tau) = pv'$. We compute $\ddot{y}$ from the equation:

$$\ddot{y} = p\ddot{x} = -p(1 + k\dot{x})Kx = -(1 + (k/p)\dot{y})Ky$$

Therefore y is a solution for the equation $\ddot{y} = -(1 + k/p\dot{y})Ky$ with start condition $\dot{y}(0) = pv_0$. The restitution coefficient is $\varepsilon(v_0, k) = -v'/v_0 = -\dot{x}(\tau)/\dot{x}(0) = -(\dot{x}(\tau)p)/(\dot{x}(0)p) = -\dot{y}(\tau)/\dot{y}(0) = \varepsilon(pv_0, k/p)$.

Now we can check the properties of $\varepsilon(v_0, k)$ in order to extract $k(v_0, \varepsilon)$. It is hard to do without having an explicit equation relating these values. This equation is too complex to be solved symbolically, therefore numerical experiments have been done. Since $\varepsilon(v_0, k)$=$\varepsilon(v_0 k, 1)$ it is enough to consider the function $\varepsilon(u, 1)$, where $u = v_0 k$.

Various plots for $\varepsilon(u, 1)$, $10^{-2} < u < 10^2$ has been obtained using Mathematica. We found two ways to approximate this function, using an fraction-based and a polynomial-based approximation.

4.8 Fraction-based Approximation

By looking at the plots we guessed that $\varepsilon(u, 1) \approx 1/(u + 1)$. We make an assumption that $\varepsilon(u, 1) = 1/(u + 1)$ and then extract k. Since $\varepsilon(v_0 k, 1) = 1/(v_0 k + 1)$, we can extract $k = 1/(v_0\varepsilon(v_0, k)) - 1/\varepsilon(v_0, k)$. This expression with *required* ε can be inserted into the original equation:

$$F(t) = -(1 + (1/(v_0\varepsilon_{required}) - 1/\varepsilon_{required})\dot{x}(t))Kmx(t)$$

This equation is solved numerically. It appears that now resulting ε differs from the required one not more than for 0.093. This value does not depend on v_0, m and K.

4.9 Polynomial-based Approximation

The plots do not resemble a polynomial function. However the function $\varepsilon_1(m) = \varepsilon(e^m, 1)$ very resembles a cubic polynomial of m. Four "typical" points of the curve have been guessed, and a polynomial

$$\varepsilon_1(m) = a_3 m^3 + a_2 m^2 + a_1 m + a_0$$

is found using Mathematica. How to find m? The equation $\varepsilon_1(m) = n$ has three solutions which can be expressed in algebraic form. Some of them are complex numbers or do not correspond to the piece of the curve we consider now. The appropriate solution is denoted as $m(n)$. The computations show that m can be chosen as one of solutions of the equation:

$$m(n) = 0.620898 + \mathrm{Re}\left(\frac{-0.163279 + 0.282808\,i}{T^{\frac{1}{3}}} - (15.6309 + 27.0735\,i)\ T^{\frac{1}{3}}\right)$$

where

$$T = -0.000833598 + 0.00193378\, n + 0.00193378\, \sqrt{-0.983167 + n}\, \sqrt{0.121024 + n}$$

The imaginary part of the expression within Re() is zero. According to definition of ε_1,

$$\varepsilon(v_0, k) = \varepsilon(v_0 k, 1) = \varepsilon_1(\ln(v_0 k))$$

Therefore k can be expressed as

$$k = \frac{1}{v_0} e^{m(\varepsilon(v_0,k))}$$

Again k is inserted into the original differential equation system. When The equation

$$F(t) = -(1 + \frac{1}{v_0} e^{m(\varepsilon_{required})} \dot{x}(t)) K m x(t)$$

is solved for $0.001 < \varepsilon < 0.999$ and resulting ε is found, the absolute value of the difference between required and resulting restitution is always less than 0.05. Smaller deviation can probably be achieved by choosing other four or five points for interpolation.

Polynomials of order 5 and higher (they require six points) cannot be used for interpolation in this method because there is no way to express the inverse function (i.e. find the root symbolically).

5 Collision Detection Software

5.1 General Properties of Collision Detection Software

Collision detection is widely used in simulation of multibody systems, in design of virtual environments, and in general 3D graphics. Given coordinates of two or several bodies, collision detection functions determine whether the objects share common points in space, and if they are close enough, determines the distance between them. Each time when collision is examined, the three steps are performed:

Choice of candidates for collision. On this stage the bodies that cannot collide due to simple geometrical relations between them are rejected. This selection is based on bounding boxes of the bodies. A bounding box is usually a good approximation of body position and size. It is easy to compute the bounding box from body geometry. There exist efficient algorithms [6] to check whether any bounding boxes in the 3D space intersect.

The bounding boxes are always axis-aligned. Static bounding boxes are computed when the body geometry is specified. They preserve their size and move when the body is moving. Dynamic bounding boxes are determined from the geometry, position and the current rotation angle of the body, and they are more accurate.

Low-level collision detection and distance determination. When bounding boxes of a pair of bodies intersect, the components (features) of the bodies are checked for intersection. It is easy to check whether triangles intersect, and find the distance between them. There exist methods for more complex features, such as spline surfaces. However it can be difficult to define a body using such surfaces in a consistent way. Also, collision detection for complex surfaces might be time consuming.

If the bodies are disjoint, the low-level collision detection algorithms find the distance between the closest features of the objects. They also determine the closest points on these features.

In the bodies intersect, the algorithms determine a pair of features that intersect.

Response handling In order to compute reaction forces more detailed geometrical information about the collision is necessary. Often the collision detection package cannot provide the information that is needed, for instance, the penetration depth, collision plane, time of collision, collision volume and area of collision surface.

Sometimes this information is available, but just for certain time steps.

6 Using SOLID for Collision Detection

We chose SOLID [3] as collision detection package for our experiments. In this section we discuss

- Using SOLID interface functions (Section 6.1) and difficulties in obtaining correct collision plane (Section 6.2).
- How geometry of bodies is specified when SOLID is used with Modelica (Section 6.3).
- How the shortest separation vector (defined in Section 3.5) is computed using SOLID callback function (Section 6.4). The shortest separation vector defines direction and magnitude of penetration depth. The penetration depth is then used for computation of collision force (Section 4).

6.1 Using SOLID interface functions

When SOLID is used the following steps are performed:

- Object shapes are created. There exist predefined primitive shapes (box, cone, cylinder and sphere). A complex shape can be constructed from one or several so called polytopes. Each polytope is a point, a line, a triangle, a tetrahedron, a convex polygon or a convex polyhedron. Polytopes are always defined by list of coordinates of all their vertices.

- The bodies are created (instantiated) based on the corresponding shape. There can be several bodies of the same shape.
- The collision detection is performed for each time frame. In order to describe the state of the bodies at each time frame, their rotation, translation and scaling factor are specified. The tool uses frame coherence when determines collision, i.e. it reuses information about the state of the bodies from the previous time frame.
- Collision details in SOLID are obtained using a *callback* function. Each time when collision between a pair of bodies is detected (i.e. the bodies penetrate already), some user-defined callback function is called.
- If the current state of the bodies will be used in a future step (for closest feature determination) it should be stored by function call `dtProceed()`.
- The function `dtTest()` is called in order to check all pairs of the bodies and it calls the callback function in case of collision.
- The callback function written by the library user obtains data about the colliding features as parameters.

6.2 Collision Plane Definition Problem

For physics-based simulation the "smart response" option of SOLID is used. When collision is detected, the closest point pair of the objects at placements from the previous time frame is reported by the callback function. In mathematical notation, assume that the time frame $t+1$ is analyzed and objects $A(t)$[2] and $B(t)$ were disjoint, and objects $A(t+1)$ and $B(t+1)$ penetrate. The points $P_{A(t)B(t)}$ and $P_{B(t)A(t)}$ are obtained as parameters by the callback function.

Vector defined by these two points can serve as approximation to the normal to the collision plane. The direction of the collision force can be set as equal to the direction of this normal vector.

This approach, however leads to some difficulties when the objects move during the collision and continue to be in the penetration for several time steps.

Assume that bodies $A(t+i)$ and $B(t+i)$ are disjoint when $i=0$ and then intersect for $i=1,...,m$. The points $P_{A(t+i)B(t+i)}$ cannot be obtained for $i=1,...,m$. Therefore we cannot obtain collision plane. From the other side, point $P_{A(t)B(t)}$ cannot be used for finding the collision plane for $i=2,...,m$ since the plane could change during the collision.

Figure 19 demonstrates how the collision plane (that should be perpendicular to the vector of the object separation force) can change during the collision. At Figure 19(a), at time step t the objects are disjoint. At figure 19(b) at time step $t+1$ the object start to intersect, and this fact is noticed by SOLID. The package finds the

[2]The notation $A(t)$ means body A at time frame t.

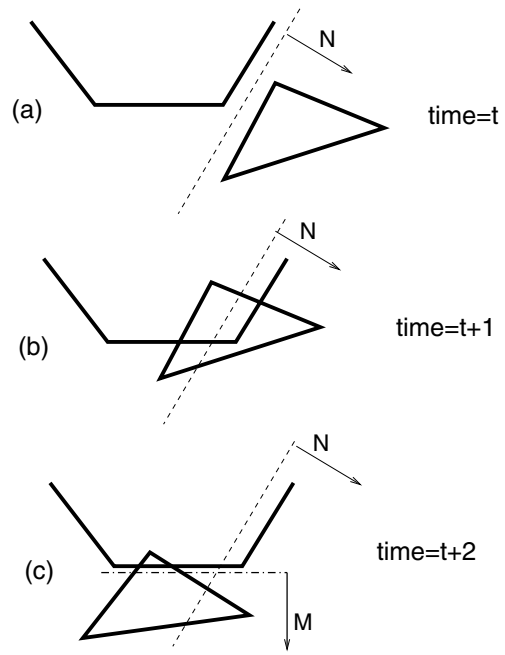


Figure 19: Variations of the collision plane during the collision

position of the closest features at the previous time frame (t) and determines the collision plane, as well as normal vector to this plane, N . Finally, at Figure 19(c) at time step $t + 2$ the position of the bodies have changed and the actual collision plane has changed too. The vector M should be used instead of N.

Our solution to this problem is presented in the section 6.4.

6.3 Geometry Specification

In order to transmit geometry specifications used in mechanical modeling in Modelica to SOLID user interface functions two ways have been chosen. Modelica bodies may have a predefined shape such as box, cone, cylinder, sphere, etc. These shapes correspond to predefined objects in SOLID (box, cone, cylinder and sphere). Alternatively, Modelica bodies may have custom geometry, described using some external format.

When SolidWorks to Modelica translation is performed, STL[1] is used as a file format for geometry description. This format contains triangles with coordinates of their vertices as well as normal vectors. This presentation is translated into collection of SOLID triangles to form a shape.

6.4 Collision Response Detail Handling

Our force-based approach to contact handling requires to obtain collision points, separation force direction and penetration depth from collision detection software.

In our approach we currently consider the worst case assumption. In the worst case the collision may take many time steps (frames) and during the collision the separation force direction changes. It is also possible that the contact never ends.

Assume that collision between the objects A and B is detected at time frame t. In order to find the shortest separation vector $\vec{c}$ (as defined in Section 3.5) we use a search algorithm.

The algorithm is implemented as two nested loops. In the external loop the length for $\vec{c}$ is chosen. Initially a very small estimated c^0 is chosen[3], and after that it is incremented at each loop iteration, so that $c^i = (1 + \kappa)c^{i-1}$. The value c^0 should be chosen so that it is negligible in comparison with typical penetration depth. The value κ should be small if high accuracy is required. The reasonable choice interval for it is $0.1 < \kappa < 0.4$. The smaller κ is the slower and more accurate is the search.

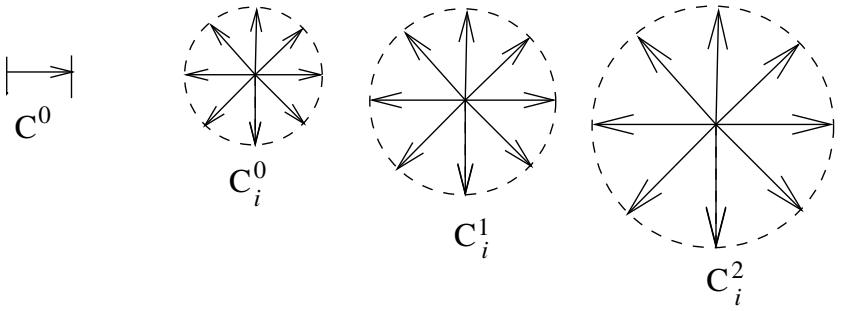


Figure 20: Sequence of trial vectors c^0, c^1, etc.

In the nested internal loop (over variable m) the direction for $\vec{c_m^i}$ is chosen. In order to sweep through all possible directions the 27 vectors (x_i, y_j, z_k) are selected, where the components can be -1, 0 and 1. The vector (0,0,0) is excluded from the list. All the direction vectors are normalized and then multiplied by the current length, i.e. c_i. As result, the vectors $\vec{c_m^i}$, $1 \leq m \leq 26$ are obtained. A 2D variant (8 vectors) is shown at Figure 20.

There is no way to ask SOLID directly about the distance and the closest points of two bodies. However, if the bodies penetrate it is possible to make a request about the closest points in the previous step. We use the following way around this problem.

The algorithm checks if bodies A and $B(\vec{c_m^i})$ (i.e. B displaced by $\vec{c_m^i}$) are disjoint or they penetrate. If they are disjoint then routine `dtProceed()` is used in order to store object's position. Again the positions of A and B are passed to the package. The `dtTest` routine finds that they intersect, and therefore the callback routine delivers to the algorithm the closets points between the bodies A and $B(\vec{c_m^i})$. These points can be referred as $P_{A,B(\vec{c_m^i})}$ and $P_{B(\vec{c_m^i}),A}$. The vector

[3]Here and further the superscript should be regarded as an index, not as an exponent in the *power* operation.

between the points is referred as $\vec{d_m^i}$.

If there is at least one vector that makes the bodies disjoint in the set of vectors $\vec{c_m^i}$, $1 \leq m \leq 26$, the external loop of the algorithm stops. The internal loop should find the most appropriate vector if there are several of them. We know that all $\vec{c_m^i}$ that made the bodies disjoint are slightly longer than necessary. All of them can be shortened by some distance. This distance is length of projection of the vector $\vec{d_m^i}$ on $\vec{c_m^i}$. The length of projection is $p^m = \vec{c_m^i}\vec{d_m^i}/|\vec{c_m^i}|$, and the m giving the largest projection should be chosen.

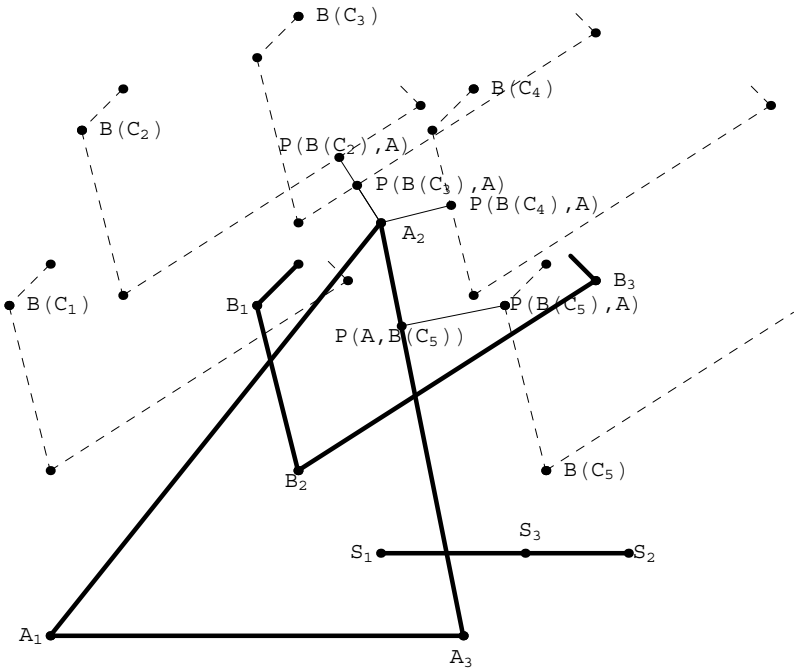


Figure 21: Computation of the shortest separation vector

The inner loop of the algorithm can be demonstrated on the two-dimensional illustration (Figure 21). It shows the body A with vertices A_1, A_2, A_3. The body B has vertices B_1, B_2, B_3 as well as some other vertices that are currently ignored. The current separation vector length c^i (we further skip the index i) is chosen as $|S_1S_2|$. There are five vectors $\vec{c_1}, ..., \vec{c_5}$ of the same length. Other vectors are currently ignored. These vectors are not shown on the picture; instead, the results of displacement of the body B according to these vectors are shown: $B(c_1), ..., B(c_5)$. The collision detection algorithm finds that $B(c_1)$ intersects A, but other bodies $B(c_2), B(c_3), B(c_4), B(c_5)$ are disjoint with A. For every disjoint objects the collision detection algorithm finds the closest points:

- For $B(c_2)$ and A the closest points are $P(B(c_2), A)$ and A_2 (vector $\vec{d_2}$).

- For $B(c_3)$ and A the closest points are $P(B(c_3), A)$ and A_2 (vector $\vec{d_3}$).
- For $B(c_4)$ and A the closest points are $P(B(c_4), A)$ and A_2 (vector $\vec{d_4}$).
- For $B(c_5)$ and A the closest points are $P(B(c_5), A)$ and $P(A, B(c_5))$ (vector $\vec{d_5}$).

The largest of the lengths of $\vec{d_i}$ is the length of $\vec{d_5}$. Therefore the direction of the separation vector is $\vec{c_5} = \vec{S_1 S_2}$. The length of the actual shortest separation vector is shorter than $|S_1 S_2|$. The projection of $\vec{d_5}$ on $\vec{c_5}$ is subtracted from $|S_1 S_2|$. The result is $|S_1 S_3|$. This is the penetration depth used in the computation of collision force.

6.5 Special Cases for Speedup of the Search

The algorithm we consider for finding the penetration depth is rather slow. It can be greatly optimized if we know that the bodies are convex, only one single vertex (P_B) of body B is within A, and for all features of B the face F_A is the closest face. With such assumption it is enough to move the vertex (together with the body B) to the closest face of A. i.e. F_A.

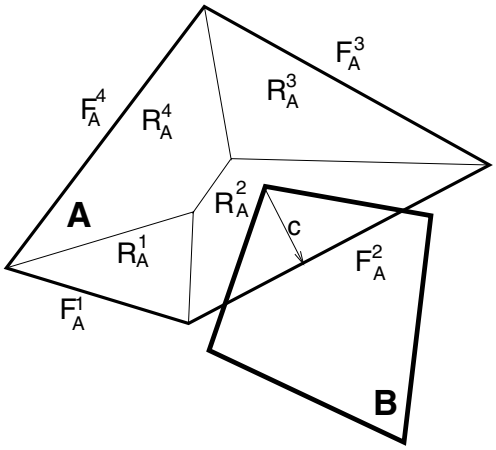


Figure 22: The closest face can be found in advance.

Finding the closest face can be optimized. The object A consisting of faces $F_A^1, ..., F_A^M$ can be divided during the preprocessing stage to interior regions $R_A^1, ..., R_A^M$ (one region per face) such way that for all the points in the region R_A^i the face F_A^i is the closest one (see Figure 22). These regions are sometimes called *pseudo Voronoi regions* [15]. Preprocessing, however, can be a difficult problem.

The Figure 23 demonstrates a case when this assumption is violated. When vertex P_B is moved to the closest face F_A^1, the bodies are still penetrating.

The algorithm can be made more exact (and slow) by adding more loops for fine-grained search and more steps in the outer loop. Some hints can be given to the algorithm in order to choose the initial approximation.

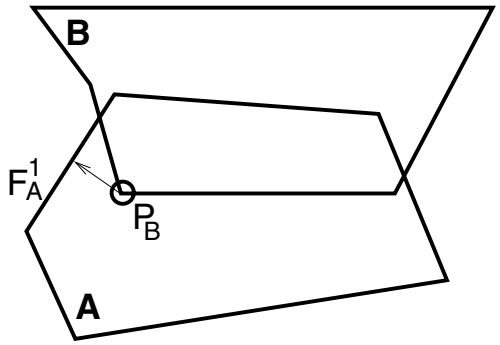


Figure 23: Collision involving several vertices

6.6 Combining Penetration Depth and Distance

The collision detection algorithms based on impulse model (Section 2) utilize a different way of using distance between the bodies. The bodies are considered as intersecting if the distance between their closest features is less than certain threshold. It is not very accurate model either, since the bodies start to interact when they are still on some distance from each other. The impulse-based algorithms do not allow the bodies intersect at all. This is done either by estimating the maximum velocity and choosing an appropriate time step, or by reducing the time steps before the collision moment. The approach based on distance threshold can be combined with our force-based model.

In this case we use both the penetration depth and the distance between the bodies in computation of forces. The vector $\vec{x}$ used for collision force definition can be defined as follows.

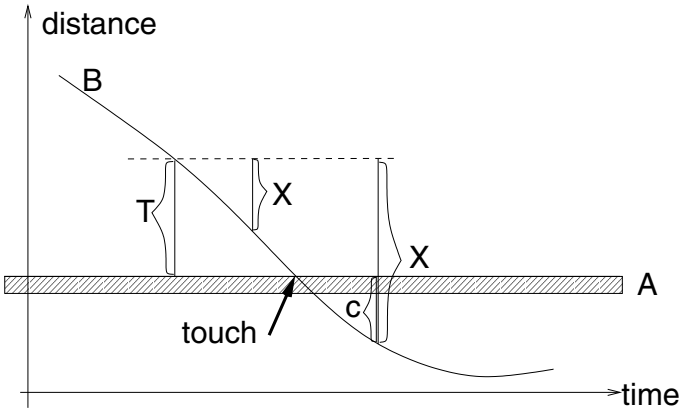


Figure 24: Combined approach to evaluation of x.

For instance, the vector x used for computation of the force (see Figure 24) can be defined as

- $\vec{x} = 0$ if $d \geq T$,
- $|x| = T - d$ and $\vec{x}$ is directed as $\vec{d}$ if $d < T$ and objects are disjoint,
- $|x| = T$ but its direction is undefined if $d = 0$ (degenerate case which should almost never happen),
- $|x| = T + |c|$ and $\vec{x}$ is directed as $\vec{c}$ if objects intersect,

where

- d is distance between the two closest features of two disjoint objects.
- $\vec{d}$ is vector between the closest points on them.
- T is threshold, chosen depending on the average velocities and computation time step.
- $\vec{c}$ penetration depth and direction.

Unfortunately the SOLID tool for collision detection cannot be used for this combined model. This tool does not evaluate the distance between the bodies ($|d|$) if they do not collide at the next time frame. Therefore other collision detection packages such as I_COLLIDE [6] can be used for this purpose.

7 Applications Using Force-based Model

This section shows examples of modeling the force-based collision in Modelica.

The first examples use the MBS library but do not use collision detection package. The last example uses this library, collision detection tool SOLID, as well as some interface and integration functions.

7.1 Pendulum Colliding with an Obstacle

This pendulum is the same mechanical construct as one in Section 2.5, Figure 7. For finding the magnitude of the force it uses the equation derived in Section 4.8. It occurs that the penalty coefficient K (Section 4.8) (variable `penalty`) can be chosen arbitrarily large:

- If $K < 10^3$ the collision looks too weak and penetration is non-realistically large.
- If $10^3 < K < 10^5$ a soft collision occurs.
- If $10^5 < K < 10^15$ a hard collision occurs.
- If $K > 10^15$ it makes the integrators unstable.

The desired restitution coefficient (variable `e`) almost matched the actual restitution (variable `eres`), the computation error was $(e - eres)/eres < 0.05$.

```
model Pendulum
  Inertial i;
  RevoluteS R1(n=\{0,0,1\},q(start=1));
  BodyBase M1 (m=50,rCM=\{0,0.5,0\});
  Bar B (r=\{0,1,0\});
  ExtForce EF1  "collision force";
  parameter Real penalty=50000;
  parameter Real e=0.5 "desired restitution, 0<e<1";
  parameter Real obstacle_x = 0.5 " x coordinate of obstacle";
  Real depth "penetration depth";
  Real depvel "penetration velocity";
  Real k "velocity coefficient for penalty";
  Real force_magnitude "magnitude of collision force";
  output  Real x "x of pendulum end";
  output  Real y "y of pendulum end";
  output Real eres "resulting restitution coef" ;
  output Real v0(start=9.999) "velocity at start of collision.";
  // Need start value just to avoid zerodivision
  output Real vout "velocity at the end of collision";
equation
  connect(i.b, R1.a);
  connect(R1.b, M1.a);
  connect(R1.b, B.a);
  connect(B.b, EF1.b);
  x=B.r0b[1];
  y=B.r0b[2];
  depth = B.r0b[1]-obstacle_x;
  depvel= -B.vb[1];
  when (depth>0) then v0=depvel;  end when;
  when (depth<0) then vout=depvel; end when;
  eres=vout/v0;
  k=(1-e)/(e*v0); // approximation derived for typical k
  force_magnitude=
   if (depth>0) then (1 + k*depvel  )*penalty*depth
   else 0;
  EF1.fb={force_magnitude,0,0};
end Pendulum;
```

7.2 Pendulum Resting on an Obstacle after the Collision

This section describes an experiment with the same model as above, but it models the situation when the pendulum is bouncing and is finally *resting* on the obstacle, see Figure 25.

The only equation replaced in this model is

```
 depth = -B.r0b[2]-0.4;
```

where `-0.4` is the position of the obstacle.

The result (assuming that $K = 5 \cdot 10^4$) is shown in Figure 26. It demonstrates a collision with small penetration. If penalty K is set to 10^6 the value of the Y coordinate of the pendulum end practically never becomes less than `-0.4` in this experiment.

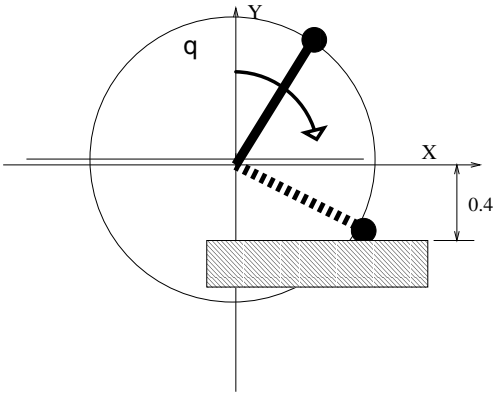


Figure 25: Pendulum resting on an obstacle after the collision.

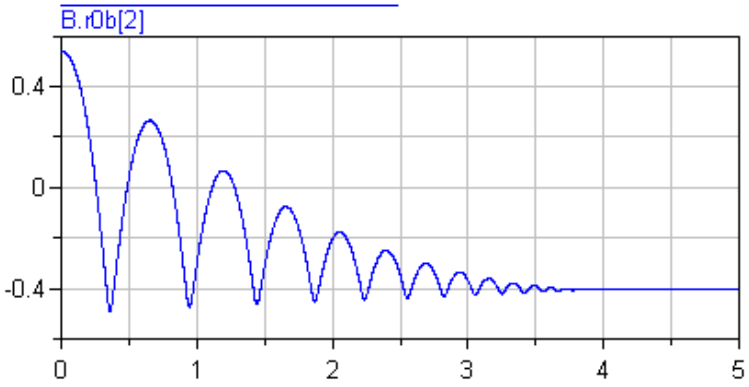


Figure 26: Height of the endpoint of the endulum resting on an obstacle after the collision.

7.3 Interface to Collision Detection Package

The Modelica model interfaces with collision detection package through function calls. The mechanical simulation based on MBS sends current positions of all bodies to the collision package (CP) and receives vectors of forces and torques that occur due to collision.

Since the number of the bodies can be large, and the data should be send to collision package at once, and received at once, all the variables needed are packed into arrays `collisionInput` and `collisionOutput`.

The function `doCollide` accepts the array `collisionInput` as an argument and returns the array `collisionOutput` as result.

The example below demonstrates a purely elastic collision, but it can also be extended for different restitution coefficient values. Also, the example demonstrates free flying bodies, but it it can also be extended for arbitrary constructs that use MBS, consisting of bodies, joints and additional external forces and torques.

```
function doCollide "Function to be called to obtain the collision force"
  input Integer bodies  "number of bodies";
  input Real collisionInput[bodies,27] "position and other data";
  output Real collisionOutput[bodies,6] "forces and torques";
external
end doCollide;
```

```
model BodyME "Body that participates in collision and can be visualized"
  extends MbsOneCutA;
  parameter Real id;
  parameter Real r[3]=[0; 0; 0] "Vector from cut A to center of mass [m]";
  parameter Real mass=0 "Mass of bar [kg]";
  parameter Real I11=0 "(1,1) element of inertia tensor [kgm^2]";
  // other inertial tensor elements are skipped for brevity
  parameter Real r0[3]=[0; 0; 0] "Origin of visual object.";
  parameter Real nx[3]=[1; 0; 0] "Vector in x direction.";
  parameter Real ny[3]=[0; 1; 0] "Vector in y direction.";
  parameter Real Material[4]=[1; 0; 0; 0.5] "Material properties";
  parameter Real parmStlIndex=0 "Index of STL file";
  parameter Real noCollision=0 "Whether the body participates in collisions";
  Real collisionInput[26] "position and other data";
  Real collisionOutput[6] "force and torque";
  output Real StlIndex "Index of STL file";
  BodyME2 ME2(r=r,mass=mass,
    II=[I11, I12, I13; I12, I22, I23; I13, I23, I33])
    "A help class - wrapper for physical body model";
  MbsOneCutB collResp;
  VisualMbsObject B(r0=r0,nx=nx,ny=ny,Material=Material);
equation
  connect(a, ME2.a);
  connect(a, collResp.b);
  connect(a, B.a);
  StlIndex = parmStlIndex;
  B.fa = [0; 0; 0];
  B.ta = [0; 0; 0];
  collisionInput=cat(1, { id,
      B.shape, B.Form, B.extra, parmStlIndex, noCollision},
      B.size, B.rxvisobj, B.ryvisobj,
      B.rvisobj, Sa[1,:], Sa[2,:], Sa[3,:]);
  collisionOutput=cat(1, collResp.fb, collResp.tb);
end BodyME;
```

```
model minibox "An MBS construction containinig a single body"
// This construction can be extended and contain
// more complex model consisting of bodies and joints
  parameter Real nx[3]=[1; 0; 0];
  parameter Real ny[3]=[0; 1; 0];
  parameter Real id;
  Real ac[24];  Real th[6];
  MbsCutA a(across=ac, through=th);
  SubInertial I(nx=nx, ny=ny);
  BodyME box_1(r=[0; 0; 0],I11=0.133333,I22=0.133333,I33=0.133333,
    I12=0,I23=0,I13=0,mass=1,
    r0=[0; 0; 0],nx=[1; 0; 0],ny=[0; 1; 0],
    Material=[0.8; 1.0; 0.8; 1.0],
    parmStlIndex=20,id=id);
equation
  connect(a, I.a);
  connect(I.b, box_1.a);
end minibox;
```

```
model IntegratedBox "A glue class representing a free flying element bx"
  extends MbsOneCutA;
  parameter Real id;
  minibox bx(id=id);
  FreeCardan2S free;
equation
  connect(a, free.a);
  connect(free.b, bx.a);
end IntegratedBox;
```

```
model world
  IntegratedBox ib1(id=1) "free flying box no. 1";
  IntegratedBox ib2(id=2) "free flying box no. 2";
  IntegratedBox ib3(id=3) "free flying box no. 3";
  IntegratedBox ib4(id=4) "free flying box no. 4";
  Inertial I(g=0)  "The roor of the MBS tree";
  parameter Integer bodies=4;
  Real wholeInput [bodies,27];
  Real wholeOutput [bodies,6];
equation
  connect(I.b, ib1.a) ;
  connect(I.b, ib2.a) ;
  connect(I.b, ib3.a) ;
  connect(I.b, ib4.a) ;
  ib1.bx.box_1.collisionInput=wholeInput[1,:];
  ib2.bx.box_1.collisionInput=wholeInput[2,:];
  ib3.bx.box_1.collisionInput=wholeInput[3,:];
  ib4.bx.box_1.collisionInput=wholeInput[4,:];
  ib1.bx.box_1.collisionOutput=wholeOutput[1,:];
  ib2.bx.box_1.collisionOutput=wholeOutput[2,:];
  ib3.bx.box_1.collisionOutput=wholeOutput[3,:];
  ib4.bx.box_1.collisionOutput=wholeOutput[4,:];
  wholeOutput=doCollide(bodies,time,wholeInput);
end world;
```

The Figure 27 demonstrates dynamic visualization of collision between four free flying cubes. The lines represent the trajectories before and after the collision. The Figure 28 contains a plot of forces acting on the first box. The three components (X, Y, Z) of the force are represented by three curves. Two major collisions occur at time 1.0 and 1.3. The trajectories of four bodies (projected on the X axis) are displayed in Figure 29. Initially two boxes move towards the zero point, and all four bodies are separated after the collision occurs.

8 Conclusion

In this report we attempt to find ways to connect MBS-based models written in Modelica with collision detection and response routines. We also derive the equations and the coefficients that can be used as collision response functions.

As the first step in this direction different collision response models were identified and experiments with Modelica simulation were performed. We demonstrate that impulse based approach for non-trivial models requires a new library of equations for propagation of impulses into joints. This new library can be a topic for

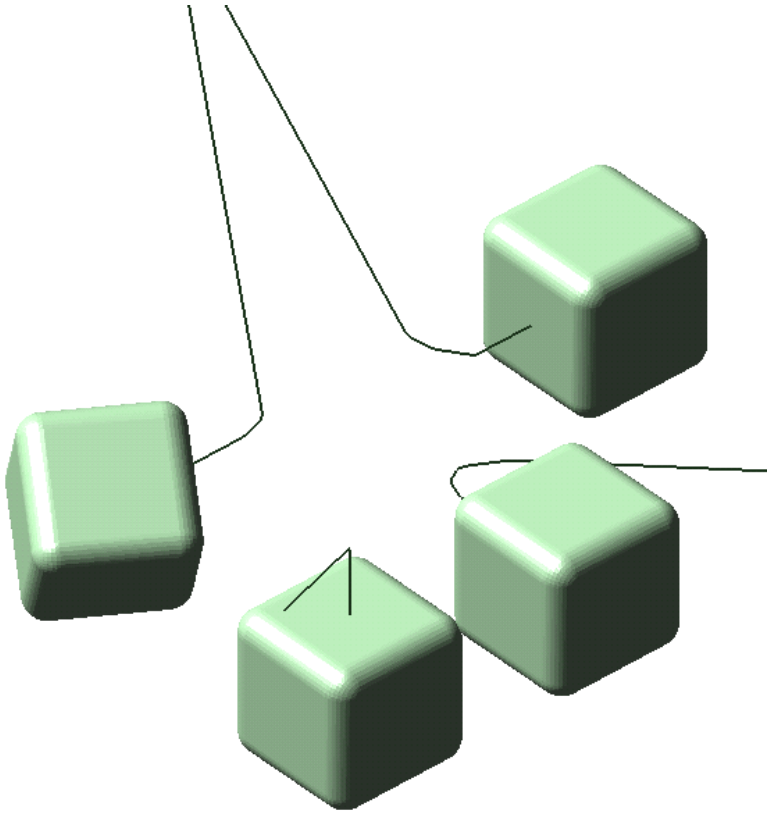


Figure 27: Dynamic visualization of collision between four free flying cubes. The lines represent the trajectories before and after the collision.

new research and experimentation. The force-based (penalty) approach can work with any MBS models, but the performance and stability should be further studied. Our currently used force model is continuous, but actually not differentiable at the time of the start and the end of collision. In future more smooth force models should be derived and used for simulation.

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] David Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies*, Ph.D. thesis, Department of Computer Science, Cornell University, 1992, Available via http://www.cs.cmu.edu/ baraff/

[3] Gino van den Bergen, *SOLID, Software Library for Interference Detection*, http://www.win.tue.nl/cs/tt/gino/solid

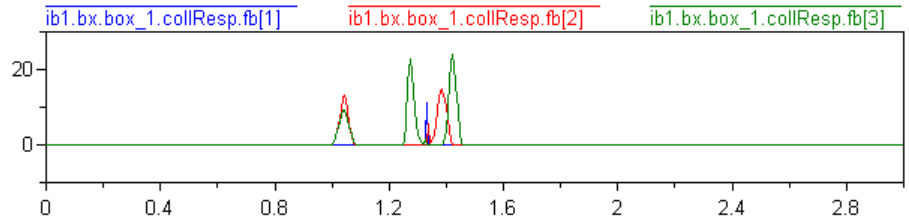


Figure 28: The forces acting on the first box. The three components (X, Y, Z) of the force are represented by three curves.

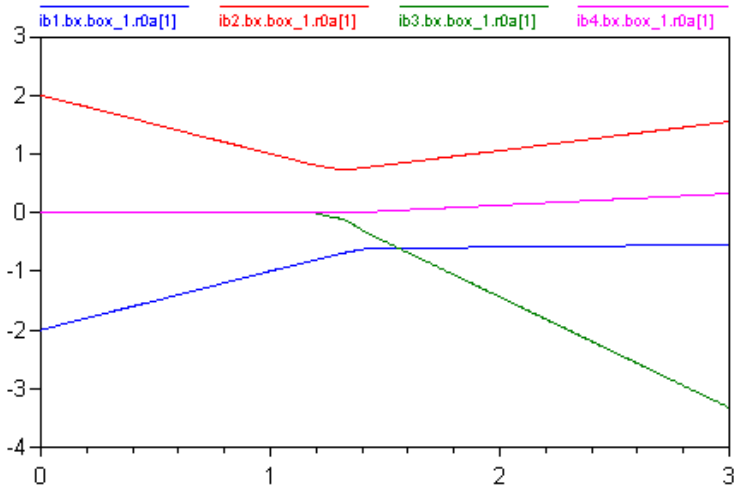


Figure 29: The trajectories of the four bodies projected on the X axis.

[4] Dag Fritzson, Peter Fritzson, Patrik Nordling, Tommy Persson, Rolling Bearing Simulation on MIMD Computers, *International Journal of Supercomp. Appl. and High Performance Computing*, 11(4), 1997.

[5] Lars Johansson, Anders Klarbring, Study of Frictional Impact Using a Non-smooth Equations Solver, to appear in *Journal of Applied Mechanics*

[6] Ming Lin and Dinesh Manocha, *Collision Detection Packages RAPID, PQP, V-COLLIDE, I-COLLIDE*, http://www.cs.unc.edu/ geom/collision_code.html

[7] Ming Lin and Stefan Gottschalk. Collision Detection between Geometric Models: A Survey. *In the Proceedings of IMA Conference on Mathematics of Surfaces 1998.* Available via http://www.cs.unc.edu/ dm/collision.html

[8] Brian Mirtich, *V-Clip Collision Detection Library* http://www.merl.com/projects/vclip/

[9] Brian Mirtich. Impulse-based Simulation of Rigid Bodies, In *Symposium on Interactive 3D Graphics*, ACM Press, 1995.

[10] Brian Mirtich. Hybrid Simulation: Combining Constraints and Impulses, in *Proceedings of the 1st Workshop on Simulation and Interaction in Virtual Environments*, ACM Press, July 1995, Available via http://www.merl.com/people/mirtich/

[11] Brian Mirtich, *Impulse-based Dynamic Simulation of Rigid Body Systems*, Ph.D. thesis, University of California, Berkeley, December 1996.

[12] Jianping Pang, Newton's Method for B-Differentiable Equations, *Mathematics of Operation Reserach*, Vol. 15, pp. 311-341.

[13] Friedrich Pfeiffer and Christoph Glocker, *Multibody Dynamics With Unualteral contacts*, Wiley Series in Nonlinear Science, 1996.

[13] Andrew Witkin and David Baraff, Physically Based Modeling: Principles and Practice(Online Siggraph '97 Course notes). Available as http://www.cs.cmu.edu/ baraff/sigcourse.

[14] Wolfram Research, *Mathematica 4.0*, Wolfram Research, 1999.

[15] Peng Zhang, *Physically Realistic Simulation of Rigid Bodies*, Thesis, Department of Computer Science, Tulane University, 1996, Available via http://www.eecs.tulane.edu/www/Zhang/

Paper 11

Lossless Compression of High-volume Numerical Data from Simulations*

Vadim Engelson, Dag Fritzson, Peter Fritzson
IDA, Linköping University S-58183, Sweden, {vaden,dagfr,petfr}@ida.liu.se

Abstract

Applications in scientific computing operate with high-volume numerical data and the occupied space should be reduced. Traditional compression algorithms cannot provide sufficient compression ratio for such kinds of data. We propose a lossless algorithm of delta-compression (a variant of predictive coding) that packs the higher-order differences between adjacent data elements. The algorithm takes into account varying domain (typically, time) steps. The algorithm is simple, it has high performance and delivers a high compression ratio for smoothly changing data. Both lossless and lossy variants of the algorithm can be used. The algorithm has been successfully applied to the output from a simulation application that uses a solver of ordinary differential equations.

1 Introduction

Applications in scientific computing often operate with large volumes of input and output data. Despite the huge capacity of the modern disk devices, the space required for data storage is often larger than the hardware allows. Data transfer over communication networks is another bottleneck in scientific computing. Text compression tools cannot compress binary numeric data. Image compression algorithms are not intended for such data. Signal compression algorithms do not work directly with numerical data of time series, when time steps are varying. Also, they are typically designed for measured values, not for simulation results and therefore not intended for lossless compression. Therefore new data compression algorithms must be designed.

*An abstract of this paper is published in *Proceedings of the 2000 IEEE Data Compression Conference,* Snowbird, Utah, March 28-30, 2000.

We assume that application data is stored as one or multiple arrays, i.e. time[1] series. The elements of the arrays are certain quantities that change *smoothly* (see Section 1.1). Informally, a smooth function is a function that is close to some polynomial. Smooth arrays contain a sequence of values of this function. Smooth arrays are typical for numerical dynamic simulations of physical phenomena where scalar values change in time. An investigation of these values reveals several things: properties of the solver, properties of mathematical model, and of course, the physical phenomena themselves. The values are consequently computed after each other. Often these quantities change so slowly that nearby elements differ in the few last digits only. Sometimes the elements of an array are computed from respective elements of another array so that the correlation between the values can be found. These numerical properties might result in a very high compression ratio. The problem is how to disciver all the features of the data and how to use them for data compression.

Algorithm		Good for time steps:	Ratio
Delta-compression	Differences	fixed	good
	Extrapolation with fixed step	fixed (same as above)	good
	Extrapolation with varying step	varying	best
Wavelet algorithms		fixed	poor or N/A

Table 1: Comparison of algorithms described in the paper.

In section 1.2 we discuss why general-purpose compression algorithms do not work with numerical data from simulations. In section 2 we introduce the simplest delta-compression algorithm (see Table 1) based on fixed time steps. Properties of memory representation for real numbers are discussed. In section 3 we reformulate the algorithm by using extrapolation formulas. In section 4 we extend the algorithm for varying steps, because such data arrays are more typical for simulations.

Experiments with artificial data sequences as well as data samples from a realistic application have been carried out and compression ratios are reported in Section 5 .

1.1 Smoothness of the data

The compression algorithm input is a sequence a consisting of values a_i, $i = 1, ..., n$. It works with arbitrary sequences of numerical values. However, it can deliver some considerable compression ratio for smooth data sequences only, where

[1]We use *time* as the domain for the steps; however in PDE-based simulations it can be a space axis.

substantial correlation between adjacent data values can be found.

Assume that a function $f : [1, n] \to R$ is evaluated on time interval $[1, n]$. The values a_i are stored in the sequence, a so that $a_i = f(i)$.

An array a is called *smooth of order* m if every a_j $(j > m)$ can be well approximated by the extrapolating polynomial based on previous m values[2]. In the simplest case, if a function that has very small and slow changes (is close to a polynomial of order 0, i.e. a constant) then the corresponding array a can be compressed very well.

In practice smooth functions represent solutions of ordinary differential equations and various continuous quantities that are computed in simulations.

1.2 Compression and data representation

The traditional text compression algorithms cannot compress numerical data because these do not utilize correlation between adjacent floating point values. Text compression algorithms represent the data as a string in an small alphabet (0..255) and attempt to find equal substrings, producing lossless compression.

Image compression algorithms represent the data as small rectangular blocks of pixels (usually three values in the 0..255 alphabet) and if the pixels in the block have similar color then the algorithm replaces color information in all the pixels by a single color, giving lossy compression. Such algorithms utilize correlation between adjacent data values, but do not use similarity of the gradient (speed of the changes) of the data.

Another approach to the problem consists of signal compression algorithms. These use data smoothness but do not consider the fact that time steps vary. Very few of them can provide lossless compression, and compression ratio is insufficient in this case.

Representation of numerical data in computer memory is important for our compression algorithm. It is assumed that 64-bit real numerical values are used. This data representation corresponds to the type `double` in most C compilers, operating systems and processor architectures used for numeric computations (Intel, Alpha, Sparc families, etc.).

Let us consider how the numerical data are represented in the memory. If p is a floating point 64-bit number, $\texttt{Int}(p)$ is defined as an integer that is represented by the same 64-bit string as p (see Table 2 and more details in Section 2.1). In this paper we use hexadecimal notation for such numbers to emphasize the byte-level representation.

Three real numbers a_1, a_2, a_3 differ in the 5-th digit after the decimal point

[2]Formally, each polynomial φ_j is created from $a_{j-m}, ..., a_{j-1}$ and $\varphi_j(j) \approx a_j$

		byte number	1	2	3	4	5	6	7	8
a_1	2.3667**1**76745585676	$\mathtt{Int}(a_1)$	40	02	ef	**09**	**ad**	**18**	**c0**	**f6**
a_2	2.3667**2**76745585676	$\mathtt{Int}(a_2)$	40	02	ef	**0e**	**eb**	**46**	**23**	**2f**
a_3	2.3667**3**76745585676	$\mathtt{Int}(a_3)$	40	02	ef	**14**	**29**	**73**	**85**	**6a**

Table 2: Integer representation of three real numbers

only. It is a linearly growing sequence. Each number occupies 64 bits. The first three bytes are almost the same, and the general-purpose algorithms for compression of byte sequences can use this fact for compression. However, the problem is that the five last bytes (in **bold**) are perceived as completely random. There is no correlation between these five bytes of one number and five bytes of another number. Traditional compression algorithms cannot compress these bytes.

2 Fixed-step Delta-compression

In this section we consider the simplest variant of delta-compression algorithm. Our algorithm computes the first (and higher-order) differences using 64-bit integer arithmetic; subsequently meaningless 0s or 1s are truncated in the result.

2.1 Internal Representation of Double Values.

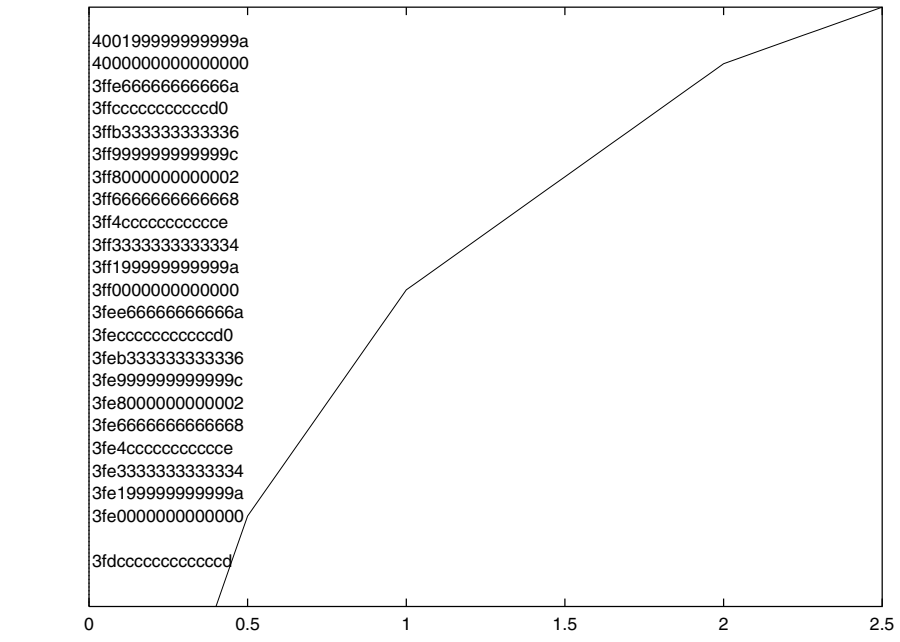
The function $\mathtt{Int}(p)$ has been defined as the *integer* representation of the 64-bit memory area allocated for a floating point number p. Here p can be any machine number between $mindouble$ and $maxdouble$[3]. This memory contains[4] concatenation of one sign bit, 11 exponent bits and 52 mantissa bits. The function $\mathtt{Int}\colon [mindouble, maxdouble] \to \{0, ..., 2^{64}-1\}$ is continuously growing everywhere except in 0. This function is very close to linear on every interval such as $[2^k, 2^{k+1}]$, $-1023 \le k \le 1023$. A fragment of the function graph is shown in the Figure 1(a), where $\mathtt{Int}(x)$ is given in hexadecimal notation.

The function $\mathtt{Real}(x)$ is opposite to $\mathtt{Int}(x)$, i.e. the equality $\mathtt{Real}(\mathtt{Int}(x)) = x$ occurs.

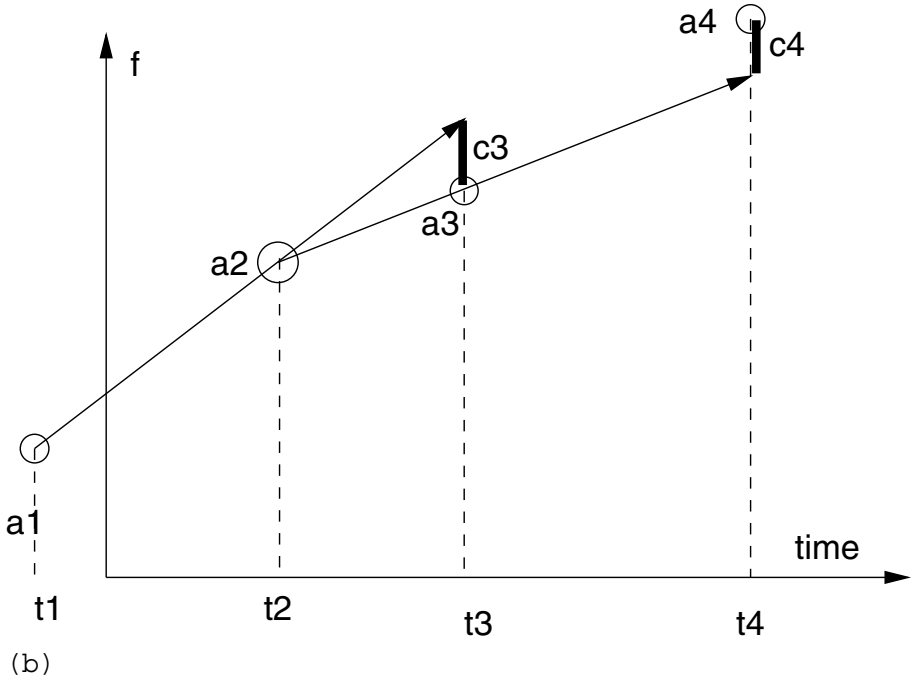
For integer p, the $\mathtt{Bin}(p, r)$ is an r-bit sequence of zeros and ones of the binary representation of p ($1 \le r \le 64$, $|p| < 2^r$).

[3]Normally $mindouble \approx -10^{309}$ and $maxdouble \approx 10^{309}$ are predefined compiler constants.

[4]This description can be processor-dependent; it holds for Intel, Alpha and Sparc families. The numbers can be investigated by a trivial C program. Order of bytes is, however, different: Intel and Alpha are "little-endian", whereas Sparc is "big-endian". This is taken into account by compression/decompression routines.



(a)



(b)

Figure 1: (a) The graph of function $\texttt{Int}(x)$. (b) Finding first order differences c_3 and c_4.

2.2 Definition of differences

Let us assume that an array b containing 64-bit integers b_i, $i = 1, ..., n$ is given. The first difference is defined as $\Delta^1 b_i = b_i - b_{i-1}$. The m-th difference is defined in a similar way as $\Delta^m b_i = \Delta^{m-1} b_i - \Delta^{m-1} b_{i-1}$.

Instead of storing the whole array b we can just store the m-th differences for b. In this case we need storage for n values:

$$(b_1, b_2, ..., b_m, \Delta^m b_{i+1}, \Delta^m b_{i+2}, ..., \Delta^m b_n)$$

The sequence b can be unambiguously restored from the m-th differences. Since b elements are integers, it is restored without loss of precision.

The difference between two 64-bit integers can be stored in 64 bits. In general case 65 bits would be needed, but we utilize the wrap-around feature of computer integer arithmetics. This feature can be illustrated by computation:

If $b_1 = 2^{63} - 1$ and $b_2 = -(2^{63} - 1)$ then $b_2 - b_1$ produces 2. Also, $b_1 + 2 = b_2$.

From above it can be concluded that a sequence b of length n can be stored as m-th differences for b , and it will occupy not more than the same memory i.e. $64n$ bits.

2.3 Truncating meaningless bits.

We can use the fact that the difference between the array elements is small relatively to the element values.

Small integer numbers have many initial zeroes (in positive numbers) or ones (in negative numbers) in their binary representation. These meaningless bits can be truncated. For instance, zeroes in `0000`**`0101`** can be truncated and just `0101` is stored. Formally, truncating of bit sequences can be described as follows:

Assume, s is a binary digit sequence of k elements $(s_1, ..., s_k)$ where $s_i \in \{0, 1\}$. Then $\texttt{Drop}(s)$ is defined as such substring $(s_l, ..., s_k)$ that all digits at the beginning of the original sequence are equal: $s_1 = s_2 = \ldots = s_l$ and after that some other digit follows: $s_l \neq s_{l+1}$.

Two extreme cases are defined: $\texttt{Drop}(\texttt{00}\ldots\texttt{0}) = \texttt{Drop}(\texttt{0}) = \texttt{0}$ and $\texttt{Drop}(\texttt{11}\ldots\texttt{1}) = \texttt{Drop}(\texttt{1}) = \texttt{1}$.

For instance, `Drop(0000`**`0101`**`)=0101`, `Drop(1111111`**`101001`**`)=101001`.

2.4 The difference compression algorithm

The algorithm compresses a sequence of real numbers a to bit sequence e using differences of order m. It consists of the following steps:

— The integer values are taken instead of real: $b_i = \texttt{Int}(a_i)$, $i = 1, ..., n$;

— First m values are copied: $c_i = b_i$ for $i = 1, ..., m$;

— The m-th order differences are computed: $c_i = \Delta^m b_i$ for $i = m + 1, ..., n$.

See c_3 and c_4 on Figure 1(b). Further the *systematic coding* method is applied to c, i.e.:

— Only necessary bits are selected $d_i = \texttt{Drop}(\texttt{Bin}(c_i, 64))$;

— The bit string length[5] and the bit string itself are stored:

$$e_i = \texttt{Concatenate}(\texttt{Bin}(\texttt{Length}(d_i), 6), d_i)$$

where `Concatenate` is the bit string concatenation operator.

— All e_i are concatenated to the single bit string $e = \texttt{Concatenate}(e_1, ..., e_n)$.

The sequence a can be restored again from e unambiguously by reverse operations[6].

For example (see b_i in Table 2),

$\Delta^1 b_2 = b_2 - b_1$ = `00 00 00 05 3e 2d 62 39`

$\Delta^1 b_3 = b_3 - b_2$ = `00 00 00 05 3e 2d 62 3b`

$\Delta^2 b_3 = \Delta^1 b_3 - \Delta^1 b_2$ = `00 00 00 00 00 00 00 02`

It can be noted that the first difference requires 5 bytes (more exactly, 36 bits). The second difference requires no more than 3 bits (`010`). Also, 6 bits are used to encode the length. Assuming that the algorithm stores b_1, b_2 and $\Delta^2 b_3$ and compression ratio $(64 * 3)/(64 * 2 + 6 + 3) \approx 1.401$ is achieved.

Normally there is no smoothness in the sequence e, therefore it cannot be compressed anymore[7].

3 Using fixed step extrapolation

The algorithm of differences of order m can be formulated differently in terms of extrapolation of order $(m - 1)$. When this reformulation is done, just slightly different[8] computations take place, and these can be seen from different point of view. We introduce the extrapolation technique here (instead of differences) in order to proceed later to varying step extrapolation algorithm in Section 4.

[5]Note that there can be various approaches for length storage, for instance $e_i = \texttt{Concatenate}(d_i, \texttt{ENDMARKER})$, but we found that our solution is close to optimal. This length can be coded in 6 bits because $2^6 = 64$.

[6]It can be done since result of 64-bit integer addition and subtraction is identical on all processors working with 64-bit integers.

[7]Relatively small additional smoothness can be found in the sequence of $\texttt{Length}(d_i)$, but we ignore this for brevity.

[8]Discussed in more detail in [5] .

The fixed step difference algorithm works well if the sequence $\texttt{Int}(a)$ can be approximated by polynomials. In real applications, however, it would be better to assume that a can be approximated by polynomials[9].

To explore this approach traditional Lagrange extrapolation form [2] should be used. The Lagrange rule for extrapolation of order $m-1$ states that for function $f(x)$ and extrapolation points $x_1, ..., x_m$ there exist a polynomial $\varphi_m(x)$ such that $\varphi_m(x_i) = f(x_i)$ for $i = 1, ..., m$. The polynomial[10] can be found as $\varphi_m(x) = L_1(x)f_1 + \ldots + L_m(x)f_m$, where $f_i = f(x_i)$ and

$$L_i(x) = \frac{(x-x_1)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_m)}{(x_i-x_1)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_m)}$$

The compression algorithm using fixed step extrapolation of order $m-1$ first takes sequence of real numbers a and produces a sequence of integers c.

First, m first values are copied: $c_j = \texttt{Int}(a_j)$ for $j = 1, ..., m$.

After that every c_j where $j = m+1, ..., n$ is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = j-m, ..., x_m = j-1$.

2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.

3. The predicted value $\varphi_m(x)$ for $x = j$ is computed using the Lagrange formula.

4. The extrapolation residual (difference between actual and predicted value) is stored (therefore our method is a variant of *predictive coding*). Since we expect that this difference is very small, the values are first converted to the integer representation and then subtracted: $c_j = \texttt{Int}(a_j) - \texttt{Int}(\varphi_m(j))$

After that the necessary operations with c are performed just like in the Section 2.4.

3.1 Decompressing

The original sequence a can be unambiguously restored from the compressed sequence:

First, sequence of integers c is restored from the bit string e.

Then first m values are copied: $a_j = \texttt{Real}(c_j)$ for $j = 1, ..., m$.

[9]The correlations between two and more arrays $(a^{[1]}, a^{[2]}, a^{[3]}, ..., a^{[r]})$ of the same length can be taken into account. It might produce high compression ratio, specially if appears that $a^{[k]} \approx p(t, a^{[1]}, a^{[2]}, a^{[3]}, ...)$ and p is a polynomial. These correlations can be revealed automatically, however this is rather time consuming.

[10]Note that the term $x_i - x_i$ is always skipped in the divider. If $x_1 = j-3$, $x_2 = j-2$, $x_3 = j-1$ then $\varphi_3(j) = f(j-3) - 3f(j-2) + 3f(j-1)$.

After that from every c_j where $j = m+1, ..., n$ the value a_j is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = j-m, ..., x_m = j-1$.

2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.

3. The predicted value $\varphi_m(x)$ for $x = j$ is computed using the Lagrange formula.

4. The actual value is computed as sum of predicted value and the residual:

$$a_j = \texttt{Real}(\texttt{Int}(\varphi_m(j)) + c_j)$$

Evaluation of $\varphi_m(j)$ includes double precision arithmetics that potentially can give different results on different processors, since they use different technique to round up multiplication or division result to fit it into 64-bit space. To guarantee lossless decompression, it should be made on the same processor family as compression. Otherwise an error in the last bit might appear, accumulate and lead to losing numerical accuracy[11].

The algorithm is fast since the coefficients for Lagrange formula are computed efficiently and only once (see [5] for details)

4 Varying step extrapolation algorithm

The previous algorithms assumed that the sequence a can be approximated by polynomials with fixed steps between extrapolation points. However in practice, simulations use adaptive ODE solvers and produce state variable values for varying, non-equidistant time steps. Therefore we should consider smooth functions with values taken with varying steps, and adapt the compression algorithm for such application data.

Assume that a function $f : [t_{min}, t_{max}] \to R$ is evaluated during the simulation.

Finite number (n) of function values is produced by the solver for time steps $(t_1, ..., t_n)$ (where $t_{min} = t_1$, $t_{max} = t_n$, $t_i < t_{i+1}$) and these are stored in the sequence, a so that $a_i = f(t_i)$.

The sequence t is used for compression and decompression of a. The sequence t itself should be compressed by the fixed step difference algorithm.

The compression algorithm using varying step extrapolation of order $m-1$ first takes the sequence of real numbers a and t and produces a sequence of integers c.

[11] Our experiments with Sparc and Alpha processor families show that difference is never larger than 2-3 last bits when extrapolation of 3rd order is used and $n = 100$.

First, m first values are copied: $c_j = \texttt{Int}(a_j)$ for $j = 1, ..., m$. After that every c_j where $j = m + 1, ..., n$ is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = t_{j-m}, ..., x_m = t_{j-1}$.

2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.

3. The predicted value $\varphi_m(x)$ for $x = t_j$ is computed using the Lagrange formula.

4. The residual (difference between actual and predicted value) is stored.

Since we expect that this difference is very small, the values are first converted to the integer representation and then subtracted: $c_j = \texttt{Int}(a_j) - \texttt{Int}(\varphi_m(t_j))$

After that the necessary operations with *c* are performed just like in the Section 2.4.

The original sequence a can be unambiguously restored from the compressed sequence under conditions described in the Section 3.1.

5 Experiments

In this chapter we describe the experimental application of both our algorithms — higher order differences (suitable for fixed steps) (orders 2, 4, 6, 8, 10) and varying step extrapolation (of orders 1, 3, 5, 7, 9). The compression ratios are compared with two wavelet algorithms (Section 5.1). There were two major tests: artificially designed test sequences and output from high-precision numerical simulation of mechanical model using ODE solver.

5.1 Experiments with wavelet-based algorithms

A widely used family of algorithms for numerical data compression are wavelet transforms. Without going into details about wavelet theory and taxonomy of transforms we just describe two transforms we experimented with.

Assuming that a sequence has some correlation between neighbor elements, wavelet transform computes an "average" value s and "difference value" d. For arbitrary integer $q > 0$ the transform compresses a sequence $a_0, ..., a_Q$, where $Q = 2^{q+1} - 1$, by running through levels $r = q, q - 1, ..., 0$. On the level r the sequence considered is $a_0, ..., a_R$, where $R = 2^{r+1} - 1$. Furthermore there are certain rules defining how to compute the sequence for level $r - 1$.

The simplest wavelet transform, **Haar** wavelet [3] computes on level p by formulae

$$s_i = (a_{2i} + a_{2i+1})/2, \quad d_i = a_{2i+1} - a_{2i} \quad i = 0, ..., 2^r - 1$$

The number of bits needed for d_i is relatively small; lossy compression algorithm using wavelets might ignore it; the lossless algorithm stores them. The elements s_i become a_i on the next level of transform.

The sequence a can be recovered by $a_{2i} = s_i - d_i/2, a_{2i+1} = s_i + d_i/2$.

This algorithm is specially successful for sequences which change very slowly and close to a polynomial function. Mainly wavelet algorithms are used for lossy compression.

There are, however, modification, called **TT-transform** [4], used for lossless compression: $s_i = \lfloor (a_{2i} + a_{2i+1})/2 \rfloor, d_i = a_{2i} - a_{2i+1} + p_i$,

where p_i is defined by $p_i = \lfloor (3s_{i-2} - 22s_{i-1} + 22s_{i+1} - 3s_{i+2} + 32)/64 \rfloor$,

and sequence can be restored by $a_{2i} = s_i + \lfloor (d_i - p_i + 1)/2 \rfloor, a_{2i+1} = s_i - \lfloor (d_i - p_i)/2 \rfloor$

Both Haar and TT transforms were applied to compression and decompression. Just like in Section 2.4 six bits were always used to encode the length of the bit strings. The experiments show that compression ratio for lossless compression is insufficient.

5.2 Artificially designed test sequences.

The test sequences for testing the algorithms were designed. These sequences contain 64-bit double precision numbers. We took into consideration that the sequence for the test cases should be rather smooth as a whole, but it should also contain small local non-smooth variations.

The size of the sequence N is chosen as 2^8 or 2^{16} (see Table 3). The longer sequence has smaller difference between adjacent elements and therefore is compressed better.

The sequence a with *fixed* time steps is defined as $a_i = F(i/N)$ where $i = 1 \ldots N$ and

$$F(x) = 0.2 + 0.7x - 0.5x^2 + 0.007\cos(100x) + 0.00007\cos(10000x) + 0.1\sin(10x)$$

The time step is constant 1, i.e. $t_i = i$. The table shows that sequences with fixed time step can be compressed equally well by both our algorithms (see repetitions in columns under "*fixed*")). The best ratio achieved is 3.68.

For testing of a sequence with *varying* time step we use $t_i = t_{i-1} + i \bmod 4 + 1$ and $a_i = F(2^{-16} t_i / 2.5)$ where i and F are as above. Here time steps vary from 1 to 4. Therefore the algorithm using varying step extrapolation shows better result than the algorithm using differences (3.73 versus 1.3).

		Ratio			
Time step:		fixed		varying	
Number of values:		2^8	2^{16}	2^8	2^{16}
Differences of order m (Section 2.4)	$m = 2$	1.58	1.64	1.45	1.53
	4	1.81	1.9	1.43	1.51
	6	2.12	2.27	1.33	1.42
	8	2.52	2.8	1.27	1.35
	10	3.13	**3.68**	1.23	1.3
Varying step m-th order extrapolation (Section 4)	$m = 1$	1.58	1.64	1.58	1.65
	3	1.8	1.9	1.81	1.91
	5	2.11	2.27	2.12	2.29
	7	2.51	2.8	2.54	2.83
	9	3.11	3.68	3.16	**3.73**
Wavelet	TT	1.7	1.86	1.23	1.67
	Haar	1.39	1.47	1.19	1.4

Table 3: Compression ratios for various data sequences and various algorithms.

5.3 Application to simulation results

The compression algorithms were applied to the output from a numerical solver of ordinary differential equations, which serves as a component in our software for dynamic simulation of bearing [1]. The program produces some quantities for every time step and writes them to the output file for analysis and further simulation. Every quantity (position, force etc.) changes very slowly from one step to another. Extreme accuracy and lossless compression is necessary, since a relative error of order 10^{-10} can substantially change the simulation results.

To chose a particular algorithm and its order the compression routines estimate achievable compression ratio by subsampling, trying different algorithms and choose the best one.

The 2837 arrays from a single simulation were analyzed. Compression ratio varies between 2.5 and 10. The algorithms were automatically chosen as follows:

— difference, first order - 20% of all arrays[12], second order - 3%, 3rd order - less than 1%.

— varying step extrapolation, first order -10 %, second order - 17%, third order - 51%. If the 4th order extrapolation is suggested, it takes 30%, but compression ratio is almost the same as in the 3rd order extrapolation.

[12]These arrays are not smooth. It took 40 bits per 64-bit number to compress them. Compression ratio was 1.6.

5.4 Lossy compression

There were four different applications of data saved by the compression algorithm:

— simulation can be restarted from any time step;

— simulation results are used in another computation;

— intermediate simulation results are sent between nodes in parallel simulation;

— simulation results are visualized in form of 2D function graphs and 3D model animations.

Only the last application allows using lossy compression. Other applications require lossless one.

The lossy compression is an extension of the basic algorithm. It can be parameterized in order to adjust the trade-off between the precision and the compression ratio.

The lossy compression can be achieved by cutting away some c bits at the end of the bit string representation. To compensate for the error one exact value is followed by some p lossy compressed values.

The user would be interested to choose the pair (c,p) for given sequence a in such a way that during decompression the absolute and relative error does not exceed ε_{abs} and ε_{rel} correspondingly.

There is a straightforward way to do that, but it requires some extra computations during compression.

First we use an interval $[a_i^{min}, a_i^{max}]$, where $a_i^{min} = \min(a_i - \varepsilon_{abs}, a_i - \varepsilon_{rel}\|a_i\|)$, $a_i^{max} = \max(a_i + \varepsilon_{abs}, a_i + \varepsilon_{rel}\|a_i\|)$.

After that interval arithmetics is used in order to obtain $[d_i^{min}, d_i^{max}]$. Then d_i is easily chosen from this interval such way that it occupies the minimal possible number of bits.

6 Conclusion

A lossless algorithm for floating-point data compression has been developed. It has similarities to image compression, since it works on bit level. It resembles wavelet compression since it uses floating-point computations for compression and decompression. The algorithm works best if the data are values of a function in some points, and this function is close to a polynomial.

The algorithm uses subtraction of one 64-bit integer representation of floating-point value from another ($\texttt{Int}(a_j) - \texttt{Int}(\varphi_m(t_j))$). If the difference would be computed between *floating-point* representations ($a_j - \varphi_m(t_j)$) there would be no win in data storage.

The algorithm is implemented as a C++ class and linked to an industrial-level application. The measurements show high compression ratio (in comparison with traditional tools) as well as high speed[5]. In the future we are going to test and measure the algorithm with the data samples from other applications.

Acknowledgments

Professor Robert Forchheimer (ISY, Linköping University) contributed many suggestions regarding the discussed algorithms.

References

[1] D. Fritzson, P. Fritzson, P. Nordling, T. Persson. Rolling Bearing Simulation on MIMD Computers. *Int. Journal of Supercomp. Appl. and High Performance Computing*, 11(4), 1997.

[2] Råde, L., Westergren, B., *Beta - Mathematics Handbook*, Studentlitteratur and Chartwell-Bratt, 1988, p.336.

[3] A. Certain, J. Popovic, T. DeRose, T. Duchamp, D. H. Salesin, W. Stuetzle. Interactive multiresolution surface viewing. *Proceedings of SIGGRAPH 96*, in *Computer Graphics Proceedings*, Annual Conference Series, 91-98, August 1996, http://www.cs.washington.edu/research/projects/grail2/www/pub/abstracts.html#InterMultSurfView

[4] M. J. Gormish, E. L. Schwartz, A. Keith, M. Boliek, A. Zandi, Lossless and nearly lossless compression for high quality images *Proc. of IS&T/SPIE's 9th Annual Symposium*, Vol. 3025, San Jose, CA, February 1997.

[5] Vadim Engelson, Dag Fritzson, Peter Fritzson. On Delta-compression Algorithm for Numerical Data from ODE-based Applications in Scientific Computing
Technical reportPELAB, Linköping University, 1999.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea / ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.