Objectmath Users Guide

Version 960513

Peter Fritzson *et al.*

PELAB — Programming Environments Laboratiory
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Phone: +46 13 281484; Email: petfr@ida.liu.se

Contents

Chapter 1

What is ObjectMath?

The *ObjectMath* system is centered around the ObjectMath language which is a hybrid language, combining object-oriented constructs with computer algebra. This combination makes it a suitable language for representing and implementing complex mathematical models. Formulae and equations can be written in a notation that closely resembles conventional mathematics, while the use of object-oriented modeling makes it possible to structure mathematical models in a natural way. In fact, there are a least three aspects of ObjectMath. The system can be viewed as follows.

- An object-oriented computer algebra language for mathematical modeling.
- An object-oriented frontend to Mathematica.
- A comprehensive high-level programming environment for scientific computing, including modeling, code generation, parallelization and visualization.

The original idea and inspiration behind the development of the ObjectMath system was to raise the abstraction level of software development in scientific computing. The traditional way of developing such software is to first formulate a mathematical model by pen and paper; perform transformations of equations by hand, and manually translate into programming languages such as Fortran for numeric computation. During the translation process errors are easily introduced.

The ObjectMath environment supports several of these phases. Object orientation facilitates structuring and reuse of mathematical models; computer algebra provides support for symbolic transformations via Mathematica; automatic generation of efficient code avoids translation errors, and visualization helps interpreting numerical data produced during execution.

1.1 ObjectMath as a Language

Before explaining in more detail about the language constructs in ObjectMath that support object orientation, we would like to present the philosophy behind the ObjectMath language, what it is and what isn’t.

We have chosen to use an existing computer algebra language, Mathematica, as a basis for ObjectMath. One advantage of this approach is that users who are familiar with the widespread Mathematica system can learn ObjectMath easily. Another reason is to simplify migration of existing Mathematica software. 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.

However, the ObjectMath language emphasizes structured mathematical modeling more than object-oriented programming with operations on state, such as of variable values and object data, in contrast to object oriented languages such as Simula and C++.

The ObjectMath view of object orientation for use in mathematical modeling is very different from the Smalltalk view of object orientation of sending messages between (dynamically) created objects. An ObjectMath model is primarily a declarative mathematical description, which allows analysis and equational reasoning. For these reasons, dynamic object creation at run-time is usually not interesting from a mathematical modeling point of view. Therefore, this is not supported by the ObjectMath language. However, variable-sized (the size is determined once at run-time) sets of objects is provided by ObjectMath, which for example can be used to represent a set of similar rollers in a bearing, or a set of electrons around an atomic nuclei.

1.1.1 Object-Oriented Mathematical Modeling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other software. One way to handle this complexity is to use object-oriented techniques. Wegner defines the basic terminology of object-oriented programming:

- *Objects* are collections of operations that share a state. These operations are often called *methods*. The state is represented by *instance variables* which are accessible only to the operation's of the object.
- *Classes* are templates from which objects can be created.
- *Inheritance* allows us to reuse the operations of a class when defining new classes. A subclass inherits the operations of its parent class and can add new operations and instance variables.

Note that Wegner's strict requirement regarding data encapsulation is not fulfilled by object oriented languages like Simula or C++, where non-local access to instance variables is allowed.

More importantly, while Wegner's definitions are suitable for describing the notions of object oriented *programming*, they are too restrictive for the case of object oriented *mathematical modeling,* where a class description may consist of a set of equations which implicitly defines the behavior of some class of physical objects or the relationships between objects. Functions should be side-effect free and regarded as mathematical functions rather than operations. Explicit operations on state can be completely absent, but can be present. Also, causality, i.e. which variables are regarded as input and which should be output, is usually not defined by such an equation-based model.

There are usually many possible choices of causality, but one must be selected before a system of equations is solved. If a system of such equations is solved symbolically, the equations are transformed into a form where some (state) variables are explicitly defined in terms of other (state) variables. If the solution process is numeric, it will compute new state variables from old variable values, and thus perform operations on the state variables. Below we define the basic terminology of *object oriented mathematical modeling*:

- An *object* is a collection of equations, functions and operations that are related to a common abstraction and may share a state. Such operations are often called *methods*. The state is represented by *instance variables*.
- *Classes* are templates from which objects or subclasses can be created.
- *Inheritance* allows us to reuse the equations, functions and operations of a class when defining objects and new classes. A subclass inherits the definitions of its parent class and can add new equations, functions, operations and instance variables.

As previously mentioned, the primary reason to introduce object oriented techniques in mathematical modeling is to reduce complexity. In order to explain these concepts we use some examples from the domain of mechanical analysis. When working with a mathematical description that consists of hundreds of equations and formulae, for instance a model of a complex machine part, it is highly advantageous to *structure* the model. A natural way to do this is to model machine elements as *objects*. *Physical entities*, e.g. rolling elements in a bearing or electrons around an atomic nuclei, are modeled as separate objects. Properties of objects like these might include:

- A surface geometry description.
- A normal to the surface.
- Forces and moments on the body.
- A volume.

These objects might define operations such as:

- Finding all contacts on the body.
- Computing the forces on the body or its displacement.
- 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 properties.

The *part-of* relation is important for modeling objects which are *composed* of other objects. This is very common in practice. 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 and inheritance is used to implement composition. However, in our opinion this should be avoided as it is conceptually wrong and usually makes the model harder to understand. Also, note that multiple inheritance cannot replace composition if an object contains several parts which are instances of the same class, a situation which occurs frequently.

One way to treat *encapsulation* is that the instance variables of an object are only accessible to the operations of the object itself. For instance, in Smalltalk operations are always accessible from outside the object while the instance variables are never accessible from the outside. However, there are other models of encapsulation, e.g. the one of C++ where the programmer specifies for each operation and instance variable whether it should be completely inaccessible from outside the object (*private*), accessible only to subclasses of the class in which it is defined (*protected*) or accessible from everywhere (*public*). A similar design choice was made for the ObjectMath language. The first object oriented language, (Simula), also allows non-local access to instance variables.

Object-oriented techniques make it practical to organize repositories of reusable software components. All classes have a well defined interface which makes it possible to use them as black boxes. Inheritance allows us to specialize existing classes and thereby reuse them, even if they do not exactly fit our needs as they are.

1.2 ObjectMath as a Mathematica Frontend

The Mathematica system already provides several frontends, such as the text-based frontend which works together with the text editor of your choice, or the notebook frontend which integrates text, graphics and mathematical typography into structured documents.

ObjectMath is a new Mathematica frontend that provides the following facilities:

- Language extensions for object oriented structuring of mathematical models in Mathematica
- Language extensions for type annotations and array creation
- Automatic translation and transparent transfer of ObjectMath models into Mathematica, augmented with support in Mathematica for viewing ObjectMath variables and objects
- Graphical browsing and editing of the class inheritance structure of object oriented mathematical models
- Text editing of ObjectMath models through an Objectmath mode in the programmable Emacs text editor, which interacts with the graphical browser

Several of these facilities are visible in the screen image of Figure 1.1. The frontend also includes facilities for generating efficient code in C++ or Fortran from ObjectMath or Mathematica code. This is further described in Section 1.3 on ObjectMath as a programming environment.

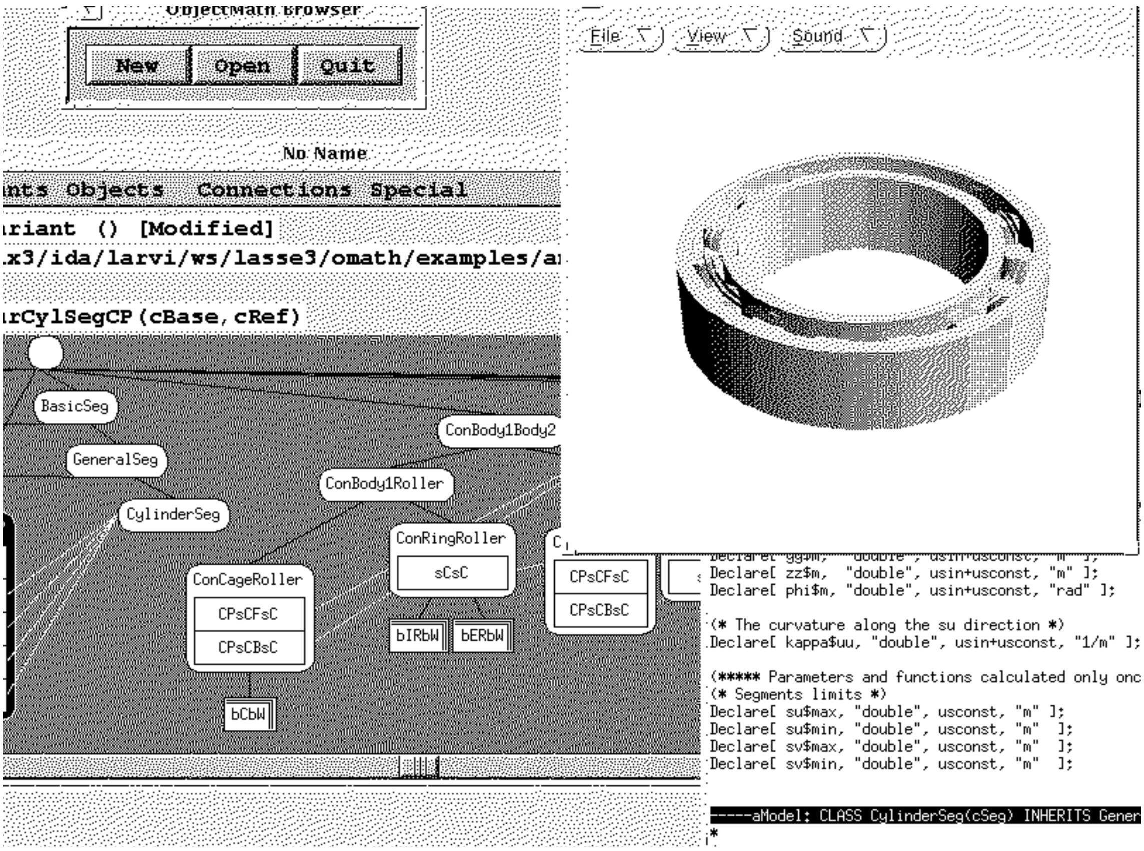


Figure 1.1: *A screen image of the ObjectMath high-level modeling environment applied to a 200-equation example model of a rolling bearing. (** Note: This figure should be updated to higher resolution)*

To create an object oriented mathematical model using the ObjectMath frontend, you first create classes in the browser, shown on the right of the interface in Figure 1.1, and select classes from an existing library. Each of the classes contains several equations and functions that together make up a system of equations describing the problem. A class can also contain equation solving instructions, transformations and substitution of symbolic expressions, input/output, graphics, etc. All operations available within the Mathematica language can be used within an ObjectMath class. The essential criterion is the operations, equations and data structures declared within the class should be related to the object or abstraction described by the class.

Then you select the *compile* command from the *model* pull-down menu, which will compile the model into Mathematica code which is loaded into the Mathematica process. One of the two text windows to the left in Figure 1.1 will then be the interaction window with Mathematica, where commands can be given, transformations and simplifications of the model can be tested, and where the executable parts of a model can be run interpretively within Mathematica. For classes that describe objects with geometric properties, it is common to have an graphic visualization operation that can show a picture of the object generated from its mathematical definition. This is the case e.g. for the SimpleRing and Roller classes, and their instances, which are nodes in the inheritance hierarchy in the right window of Figure 1.1. The graphic image of a rolling bearing to the left of Figure 1.1, containing rings and rollers, has been automatically visualized from its mathematical parametric surface definition, using such a graphic operation. In this way, the correctness of the equations and functions defining the geometry of an object can be checked visually.

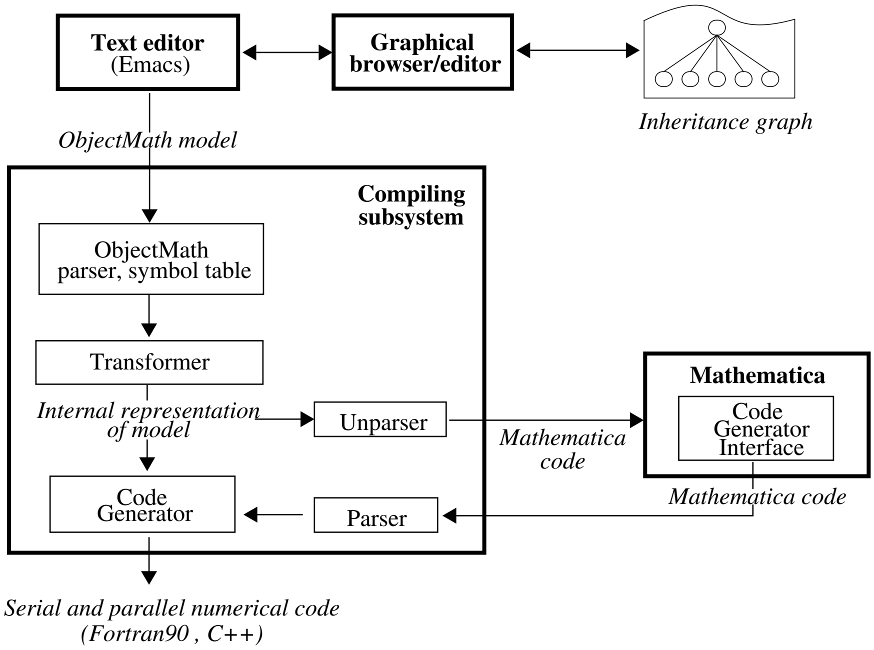


Figure 1.2: *The architecture of the interactive ObjectMath frontend and its relation to Mathematica. The frontend is composed of three processes, communicating through Unix pipes, sockets, pseudoteletypes. It communicates with Mathematica through the Mathlink interface.*

Finally, a whole ObjectMath model, or selected Objectmath/Mathematica functions or expressions can be compiled into efficient C++ or Fortran90 code, by invoking the ObjectMath code generator from within the Mathematica process. It is possible to compile expressions and Mathematica function bodies written in the compilable subset, i.e. operations that do not perform symbolic computer algebra operations, or expand into expressions that do not refer to such operations. The efficiency gain of executing compiled code instead of executing interpretively within Mathematica is highly application dependent, and typically varies from a factor of around one hundred to several thousands.

1.3 ObjectMath as a Programming Environment

As mentioned in the introduction, the main motivation behind the ObjectMath system is to provide a high-level programming environment that enhances the program development process in scientific computing. Thus it is useful to examine the traditional way of software development in scientific computing, and in what ways ObjectMath can enhance this process. We also examine the process of using ObjectMath to develop a simulation application starting from a mathematical model.

1.3.1 Background

The three major activities in scientific computing are theory development, numerical computation and design of numerical algorithms. During theory development, the model designer creates a mathematical model of an object that describes its relevant properties, such as part structures, components and substructures, material properties and interactions between components. The designer typically develops this model manually, using pen and paper. Then, during numerical computation, the designer translates the model into either a computer program or an input specification for a numerical modeling program (such as a finite-element analysis program)

Although their goal is to accurately model and solve problems, designers typically spend at least half their time writing and debugging Fortran or C programs. These programs evaluate and simulate the mathematical models. However, because simulation problems are usually nonlinear, numerical convergence problems often arise. Designers must then solve these problems, use the feed back to revise the mathematical model, and reimplemented the Fortran/C program. Figure 1.3 depicts this laborious, error-prone process.

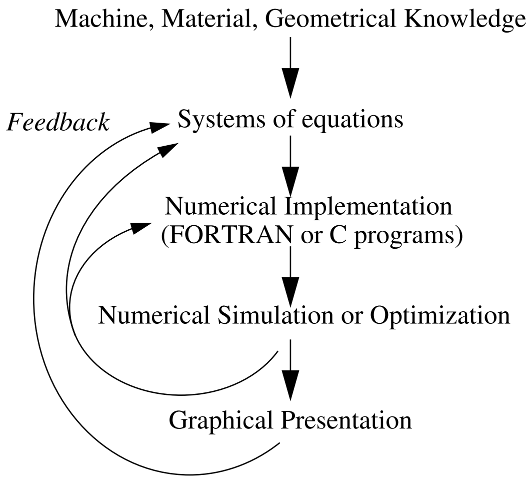


Figure 1.3: *The traditional software development cycle in scientific computing.*

Unfortunately, today's high-level tools often are of limited use here because the computational problems are too complex. Clearly, a powerful high-level programming environment is needed. The perfect programming environment would automatically transform systems of equations into efficient symbolic and numerical programs. It would select solution routines that have good convergence properties for the given problem. It would also formulate equations (given possible geometric or other constraints) and transform equations between different coordinate systems. Although it is easy to imagine such an automatic environment, it is more realistic to assume that the user will interact with the system, supplying information to help it choose the right algorithms and transformations.

The ObjectMath high-level programming environment was designed with these ideals in mind.

1.3.2 Generating a simulation program using ObjectMath

To illustrate the use of ObjectMath in software development, we use an example from machine-element analysis. Loosely defined, a machine element is some important substructure of a machine. Here we use a rolling bearing, as shown to the left of Figure 1.1. Components of a rolling bearing include inner ring, external ring, and rollers.

Objects such as rings will have attributes and associated operations. For example, attributes for a ring are the radius, a formula for a parametric definition of the ring's surface, ad formula for elasticity properties, and so on. Associated operations might be to compute the normal vector to the ring's surface, to compute the ring's volume, or to graphically visualize it on the screen.

Figure 1.4 shows, from problem input through visualization - how ObjectMath generates a simulation program, and Figure 1.2 shows the architecture of the ObjectMath interactive environment.

To use ObjectMath to simulate the behavior of a rolling-bearing, you first create classes in the browser, shown on the right of the user interface picture in Figure 1.1, and select usable already existing classes from a class library. Each of the classes contains several equations that together make up a system of equations describing the problem.

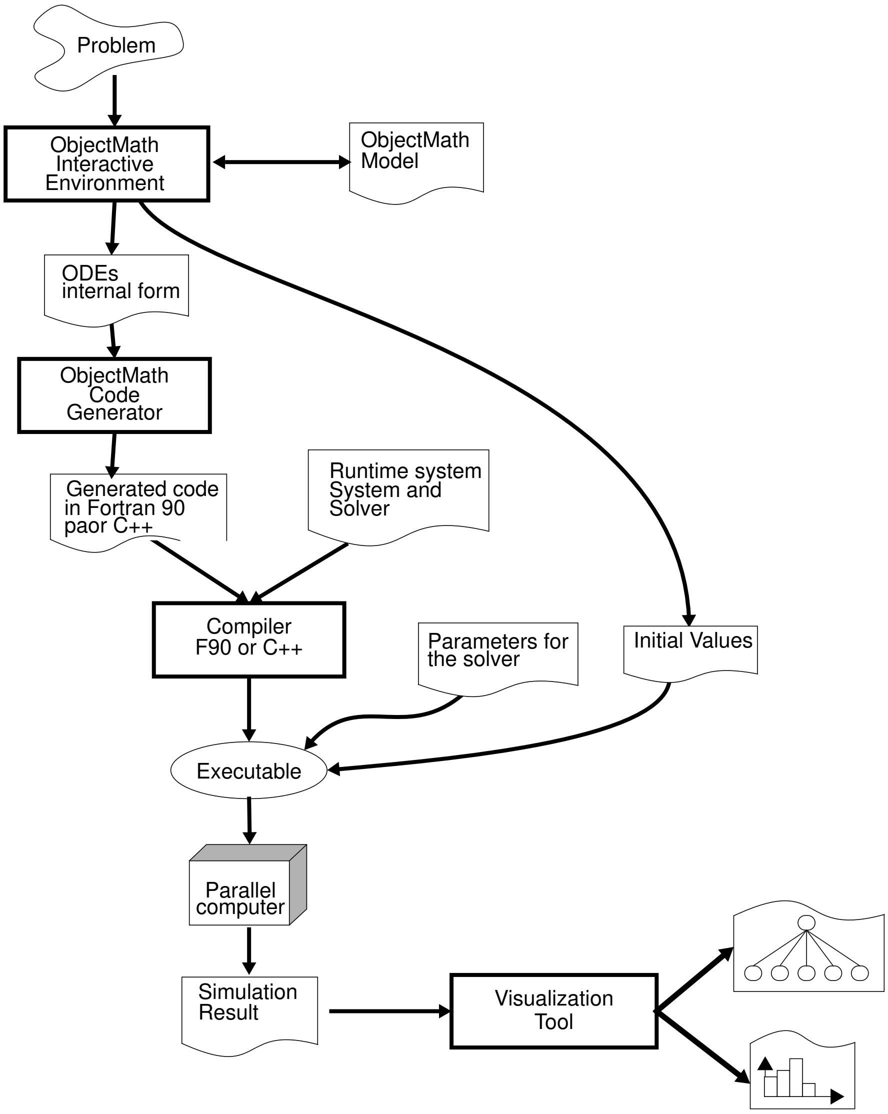


Figure 1.4: *The process of generating simulation programs using ObjectMath. Mathematica is embedded within the ObjectMath interactive environment, also depicted in Figure 1.2*

Once the class structure has been designed, you test, edit and compile the model repeatedly until the equations are of a satisfactory form. During this process, you repeatedly download and compiler the ObjectMath model to Mathematica code, and work with it interactively in the interaction window to see which sequence of transformations and simplifications gives the best results in terms of the resulting equations. At the same time, you can test small executable parts of the model.

At this point, you can also call up a visual image of physical bodies in the model, such as the bearing object to the left in Figure 1.1, and examine it for structural errors. When the equations are simple and in a suitable form that can give rise to efficient executable code, you initiate the C++/ Fortran90 code generation from ObjectMath by calling specific built-in ObjectMath functions to invoke the code generator, either interactively or from the model. ObjectMath then generates code for a numerical simulation program.

To solve linear on nonlinear equations expressed in the ObjectMath language, numerical solvers are linked to the C++/Fortran90 code. The ObjectMath-to-C++/Fortran90 code generator supplies the solver with a routine that calculates the values of functions referenced by the equations in the model.

Generated programs can either be transparently called from within the ObjectMath environment or used independently of the environment as a computation kernel (with a graphical front end, for example). To achieve high performance, the simulation program may be parallelized and executed on a high performance parallel computer. Finally, a visualization tool allows the user to browse computed numerical data by displaying curves of selected computed quantities.

Chapter 2

Getting started with ObjectMath

The purpose of this chapter is to walk through some aspects of the ObjectMath interactive environment, to help the user to become acquainted with the system. A more complete description of facilities and commands in the environment is provided in Chapter 3. The ObjectMath model *SphereCyl* which is used here as an example, is further explained in Chapter 6, and listed in Appendix B. Some Mathematica constructs will be used here without detailed explanations, which however are available in the book [Wol91] on Mathematica by Stephen Wolfram.

This description is valid for execution on Sun Sparc workstations, which currently is the only supported platform for ObjectMath.

To be able to run the system, there must be a valid Mathematica license on your workstation. Also, you need to set a few environment variables.

Setting `OMATH_HOME` to the root of the ObjectMath distribution directory is mandatory. This directory contains subdirectories bin, src, doc, examples, etc.

```
setenv OMATH_HOME /home/pelab/ws/omath4/omath
```

Also, either Emacs (named emacs) and Mathematica (named math) are in your path, or you set the `OMATH_EMACS` environment variable to the full path of the Emacs binary and the `OMATH_MMA` to the Mathematica home directory.

```
setenv OMATH_MMA  /sw/math
setenv OMATH_EMACS /usr/local/bin/emacs
```

First copy the examples to you own work directory `myomath`, by:

```
mkdir myomath
cd myomath
cp -r $OMATH_HOME/examples .
```

To start ObjectMath, type the following:

```
omath &
```

Then a window with three buttons appear, as shown in Figure 2.1. An Emacs window with two subwindows also appears.

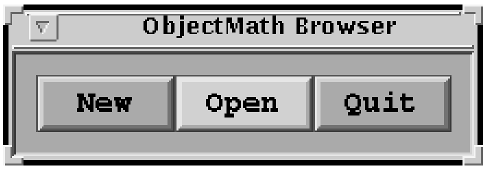


Figure 2.1: *Objectmath master window, for opening or creating models, or leaving ObjectMath.*

Click on the open button, and a file selection popup will appear, as in Figure 2.2. Select your new work directory, e.g. ~/myomath, and select the *spherecyl.omt* model.

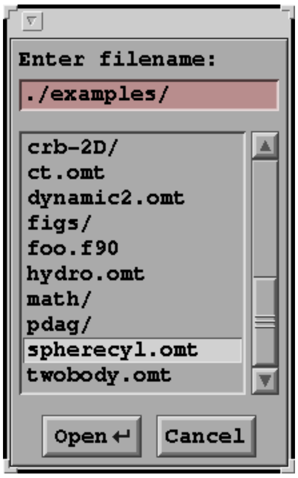


Figure 2.2: *File selection popup window*

The browser window then appears, showing the inheritance structure of classes in model *Sphere-Cyl*, see Figure 2.3.

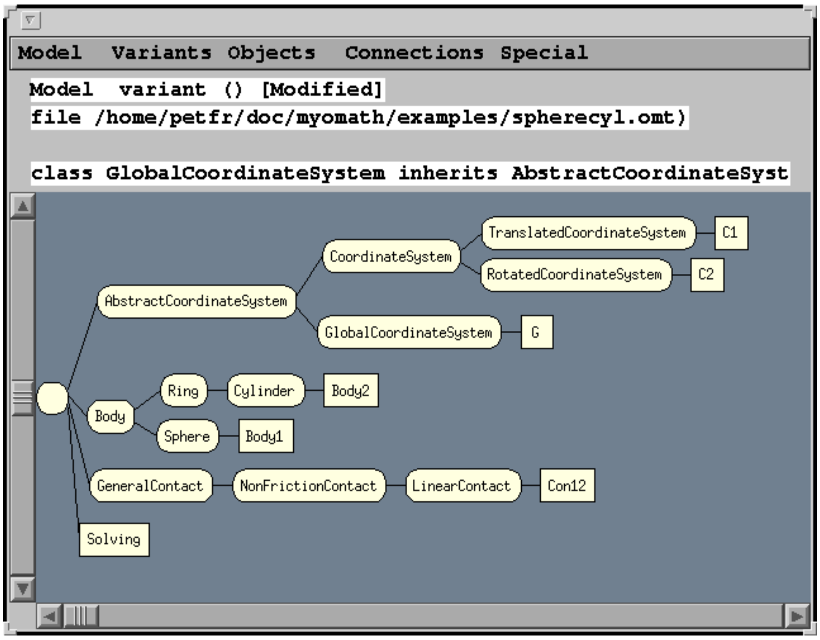


Figure 2.3: *Browser window with horisontal display of the inheritance hierarchy for the SphereCyl model.*

Click once on a symbol for some class, for example `Body`. The code belonging to class `Body` appears in the upper Emacs window, see Figure 2.4.

Figure 2.4: *Emacs windows, showing the part of the class Body in the upper window, and the Mathematica interaction window below.*

If you (left) click on some other icon, the code associated with that class will be shown in the next Emacs window, etc. You can create more than two Emacs windows by the standard Emacs command `C-X 2`.

If you would like to get back to the Mathematica interaction window, click *Model* at the top left in the ObjectMath master window (Figure 2.1), and select *Interaction* in the pull-down menu.

If you want to see the class parameters of `Body`, edit the parameters, or edit the inheritance graph, click the right mouse button over the `Body` icon in Figure 2.3 to make a popup menu appear:

Popup menu alternatives:

- Add Class — Add a new child class below the current node.
- Add Instance — Add a new child instance.
- Add Part — Add a new part within the current class or instance.
- Rename — Change name of class or instance.
- Delete — Delete the class.
- Edit — Edit class attributes.
- Move — Move the class. *(***??- to where?)*
- Show — Make a previously collapsed subgraph appear.

- *Collapse* — Make the appearance of the current subgraph be collapsed.
- *Connect* — Connect two nodes. *(** Is this really true? Check!????**)*
- *Cancel* — Cancel this popup menu.

Figure 2.5 shows the window when edit is chosen. Notice that *Body* is a class with class parameters S and B.

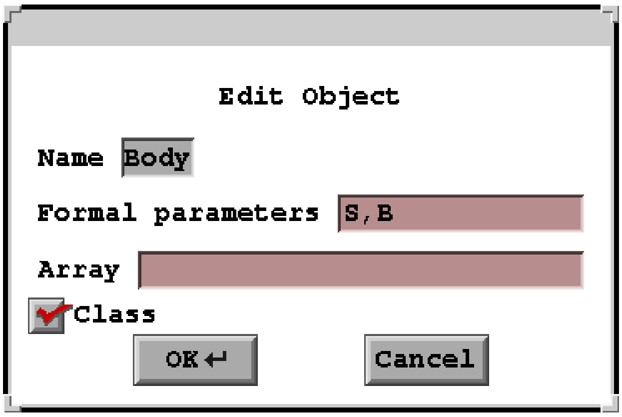


Figure 2.5: *Class parameter editing popup*

The *Model* pull-down menu (Figure 2.6) in the Browser window provides the following alternatives:

- *Interaction* — Pop up a Mathematica interaction window as one of the Emacs windows. This is used for symbolic manipulation.
- *Compile* — Translate and send a model to the Mathematica process, which becomes interaction window
- *Save* — Save the model under the cujrrent name (text format, with extension `.omt`).
- *Save as* — Save the model under specified name (text format, with extension `.omt`).
- *Print* — Print the graphic class diagram on the postscript printer.
- *Close* — Close the current model

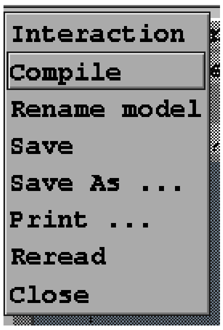


Figure 2.6: *The Model pulldown menu in the Browser window.*

Select `Compile` to translate the ObjectMath model to Mathematica, and load it into the Mathematica process accessible in the interaction window, where the text "`Receiving input from ObjectMath`" will be visible, and a new prompt will appear.

Note that you may have several models simultaneously open, even if they have the same name.

Each model will get its own browser window. However, the same Emacs and Mathematica processes are used for all models. Several models with different names may be loaded into Mathematica at the same time.

Let us continue with the `demo` example. Left click on the `Solving` instance in the browser. Call the method `SetPlotdata` by typing into the Mathematica window:

```
Solving‘SetPlotdata
```

Push *meta-return* to send this expression to Mathematica. Normal return is used to type in multi-line expressions. Meta-return is pushed by holding down the key *left* to space bar and return simultaneously, at least for most standard sun4 keyboards.

Push the `Body` icon to see the code again. Type into the interaction window:

```
Body1`Graphic[G]
```

which will construct a graphics object, bound to the current name for returned values, e.g. `Out[4]`. Type for example:

```
Show[Out[4]]    or    Show[%4]
```

to have a look at the object on the screen. Figure 2.7 shows your commands within the Mathematica *interaction* window (the lower window) in Emacs.

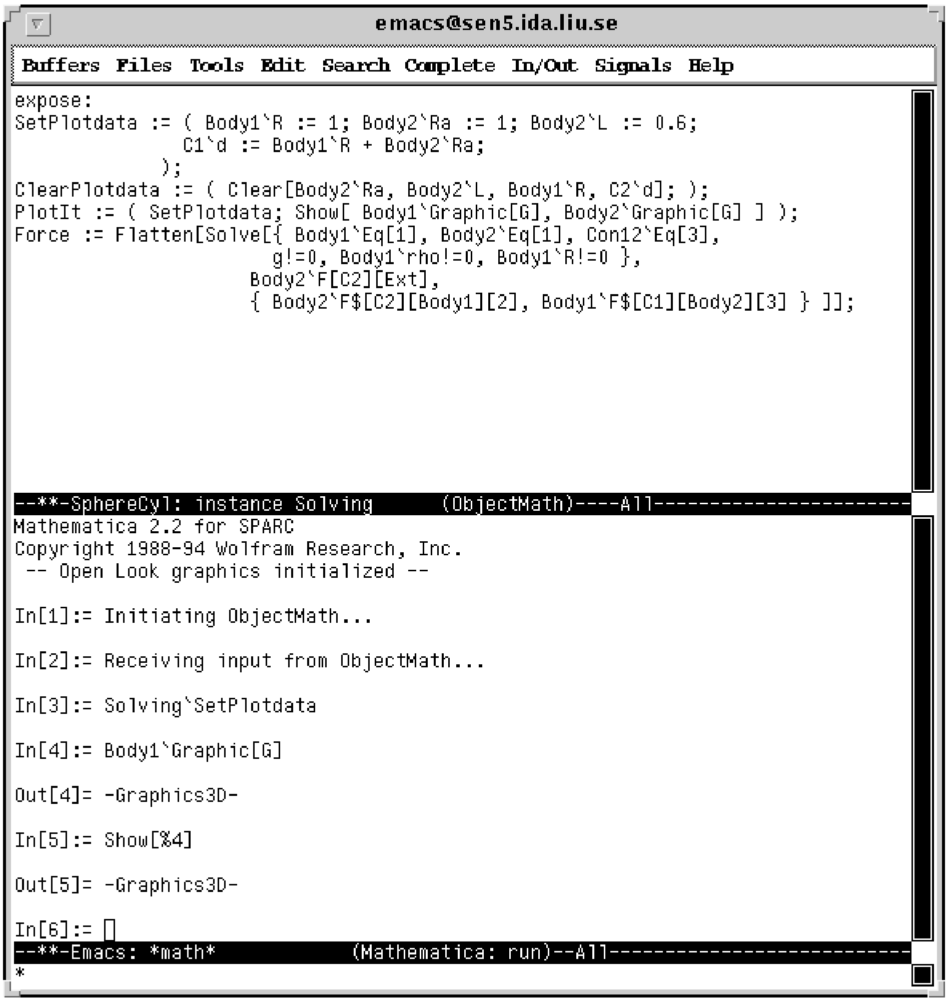


Figure 2.7: *Emacs windows. The interaction window shows the steps to create the graphics.*

Figure 2.8 shows the object when it appears.

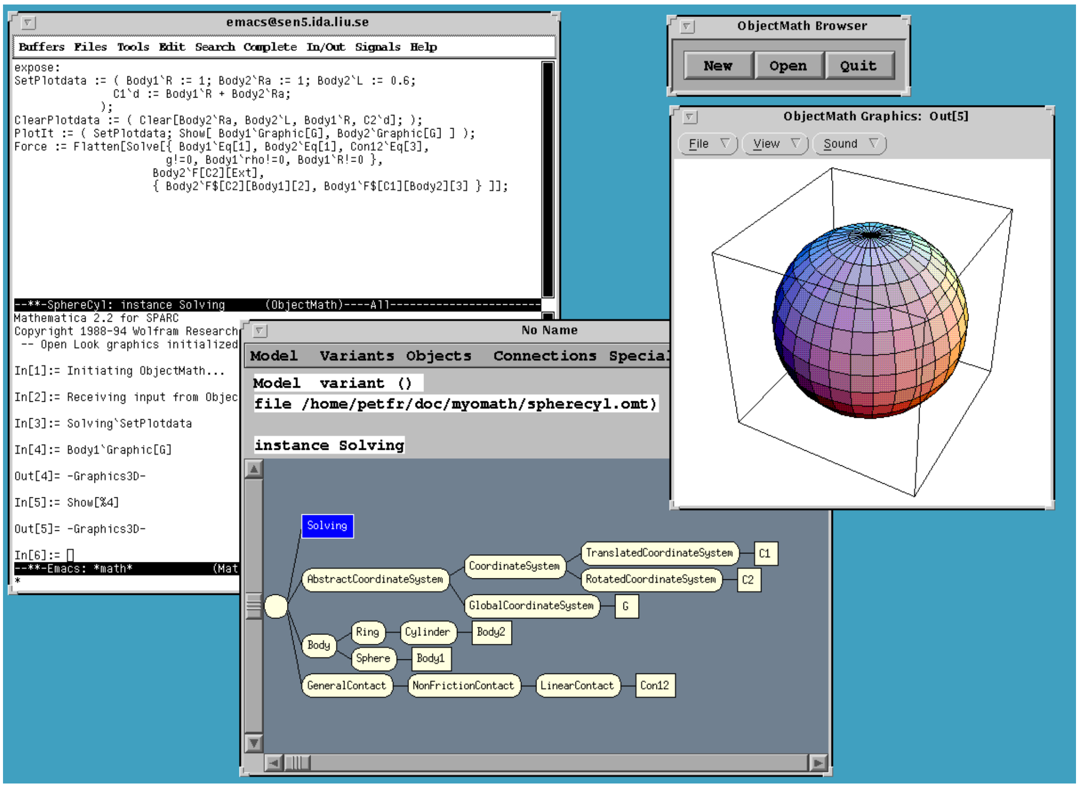


Figure 2.8: *Screen image with the sphere.*

In order to also take a look at the cylinder and sphere together, first type the following to compute a graphic object of the cylinder instance (Body2 in the model):

`Body2`Graphic[G]` (Meta-return)

Then, assuming that the cylinder was bound to `Out[6]`, type a Show command

```
Show[{%4,%6}]
```

which shows both objects (Figure 2.9).

The coordinate system `C1` is attached to `Body1` (the sphere). The coordinate system `C2` is attached to `Body2` (the cylinder). `C1` is translated relative to `G` in the z direction, which makes the sphere appear above the cylinder. `C2` is rotated 90 degrees relative to `G`, which makes the cylinder appear standing instead of lying down. The general method of using parametric surfaces makes it possible to generate pictures of a number of objects together.

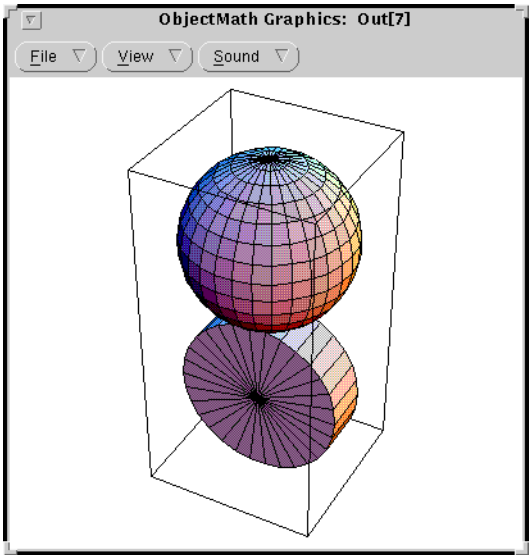


Figure 2.9: *The sphere resting on the cylinder.*

Now we would like to solve a system of equations symbolically. Regard two of the definitions in the class Solving:

```
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] } ]];
```

First we need to call `ClearPlotdata` to remove the numeric values set earlier by `SetPlotdata` (needed for plotting). Then we must call `Force`, in order to do perform the solution (see below).

The input equations to Solve are the static equilibrium equations for `Body1` and `Body2`, the contact equation 3 in `Con12`, and some additional conditions to avoid non-interesting solutions (the gravitation constant `g>0`, the density `rho>0`, and the radius `R>0`):

```
{ Body1`Eq[1], Body2`Eq[1], Con12`Eq[3],
  g!=0, Body1`rho!=0, Body1`R!=0 }
```

The desired solution is wanted in terms of `Body2`F[C2][Ext]` – the external force acting on the cylinder (which is `Body2`). In this solution we are not interested in expressing the solution in the intermediate contact forces between the sphere and the cylinder. Thus these names are passed as the third parameter to `Solve`:

```
{ Body2`F$[C2][Body1][2],
  Body1`F$[C1][Body2][3] }
```

Type to the interaction window:

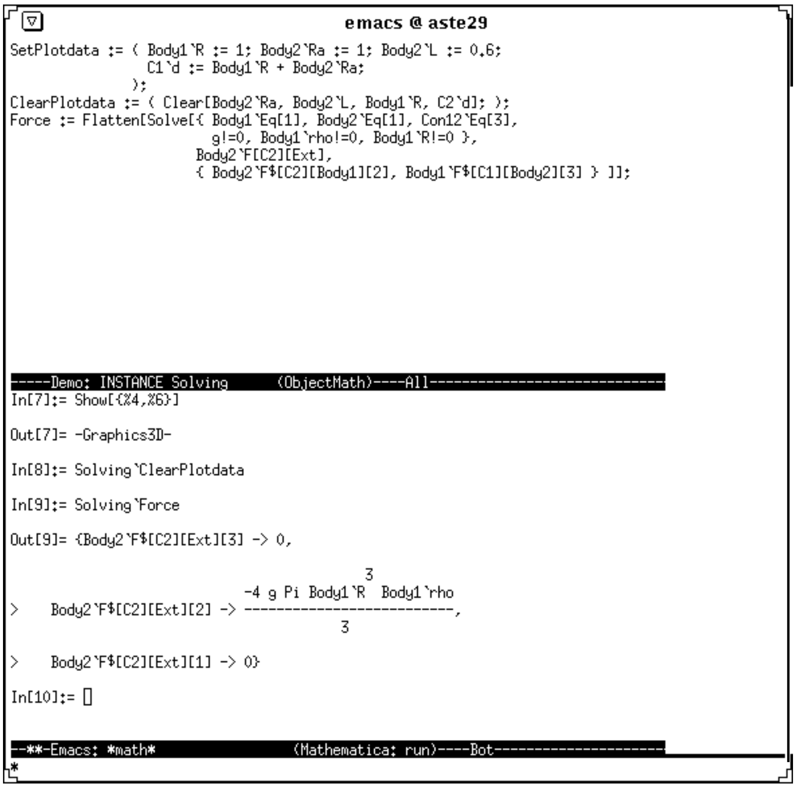
```
Solving`ClearPlotdata
Solving`Force
```

in order to perform the solving.

The solution process consists of:

- Symbolic coordinate transformations.
- Symbolic integration.
- Symbolic solution of the system of equations.

The obtained solution is g times the volume of the sphere times the density (`rho`), see Figure 2.10.


```
emacs @ aste29
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] } ]];

-----Demo: INSTANCE Solving      (ObjectMath)----All----------------------------
In[7]:= Show[{%4,%6}]

Out[7]= -Graphics3D-

In[8]:= Solving`ClearPlotdata

In[9]:= Solving`Force

Out[9]= {Body2`F$[C2][Ext][3] -> 0,

                                               3
                                -4 g Pi Body1`R  Body1`rho
>    Body2`F$[C2][Ext][2] -> --------------------------,
                                          3

>    Body2`F$[C2][Ext][1] -> 0}

In[10]:= []

--**-Emacs: *math*             (Mathematica: run)----Bot--------------------
*
```

Figure 2.10: *The final solution in symbolic form, showing the formula for the force of the sphere resting on the cylinder.*

Chapter 3

ObjectMath menus and commands

There are a number of commands available in the ObjectMath programming environment. Commands for opening an existing model (*Open*), creating a new model (*New*), as well as leaving the environment (*Quit*) are provided through the main browser window, see Figure 3.1 below.

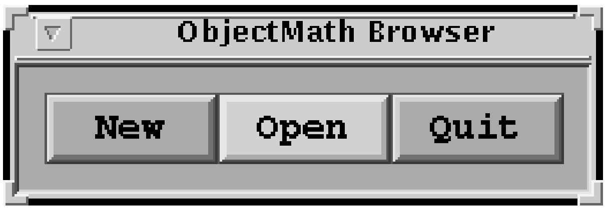


Figure 3.1: *New, Open and Quit commands available in the main window of the Browser.*

Most other commands are provided through pulldown menus in the ObjectMath class browser. General utility commands for compiling, loading, saving, printing, etc. of whole models are available in the Model pulldown menu. The variants menu controls variant handling in classes. The Objects and Connections menus have commands for editing the structure of the class inheritance hierachy within the graphic class browser.

3.1 Model pulldown menu

The *Model* pulldown menu contains the following eight utility commands that are relevant for whole models, such as interacting with models, and compiling, saving, printing, reloading or closing a model.

We describe each command briefly below.

Interaction This alternative makes the Mathematica interaction window appear as one of the Emacs windows. This is the window where the user interacts with Mathematica when performing symbolic transformations, calling the code generator for generating C++ or Fortran90, etc.

Compile This command will translate the currently selected ObjectMath model to Mathematica code, and download the generated code into the Mathematica process.

Rename model This command just renames the current model.

Save This command saves the current model as a text file, using its current name. File extension is `.omt`.

Save as This will save the model into a file with specified name.

Print The *Print* command will print the graphic class diagram on the postscript printer.

Reread *Reread* will read the model once more.

Close The *Close* command closes the currently open model.

The Model pulldown menu appears in Figure 3.2 below.

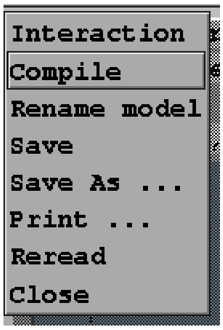


Figure 3.2: *The Model pulldown menu in the Browser window.*

3.2 Variants pulldown menu

ObjectMath contains a variant mechanism that allows several variants of the same classes to be represented within a single model. This mechanism is explained in more detail in Section 6.6. There is a notion of *current variant*, which is the currently active variant. Equations that are not in the current variant will not be translated to Mathematica code by the abovementioned *Compile* command.

The *variants* pulldown menu contains three commands, create, delete and choose:

Create The *Create* command will create a new variant with a specified name.

Delete The *Delete* command will delete a variant.

Choose *Choose* will choose the current variant to become active.

3.3 Objects pulldown menu

The Objects pulldown menu provides operations on the graphically represented class inheritance structure in the class browser. All commands are relative to a current position in the inheritance structure, which is selected by left-clicking on some node in the graph. This is always possible, since even an empty model always contains a top node. The following commands are provided:

Add class This command adds a new class node below the current node.

Add instance This command adds a new instance node below the current (class) node.

Add part The *Add part* command adds a new part within the current class or instance node.

Rename *Rename* renames the currently selected node (class, instance or part).

Delete The *Delete* command deletes the currently selected node.

Edit The *Edit* command allows editing of the class/instance parameter list, whether the node is a class or instance, and whether it represents an array of nodes.

Move The *Move* command moves a node (including its child nodes) to some other position. For example moving a class node left or right relative to sibling nodes, or moving up or down in the inheritance hierarchy. *(** check this).*

Show *Show* makes a previously collapsed subgraph be visible again.

Collapse The *Collapse* command makes the appearance of a whole subgraph be collapsed into a single node, in order to save space in the graphic display. A collapsed subgraph can be made visible again through the *show* command.

3.4 Connections pulldown menu

The *Connection* menu has commands for manipulating *inheritance* relations between classes, instances and parts.

Add The *Add* command adds an inheritance arc between the current node and some other class.

Delete The *Delete* command deletes the selected inheritance arc.

Edit *Edit* provides editing of properties of an inheritance arc. (** which properties? check this)

3.5 Special pulldown menu

The *special* pulldown menu provides a few miscellaneous commands such as choosing between horisontal (Left-Right) vertical (Top-Down) layout of the inheritance graph, or whether to display part inheritance arcs.

LR/TD Layout This command allows switching between horizontal (LR – Left-Right) or vertical (TD – Top-Down) layout of the class hierarchy. Horisontal layout sometimes gives better use of screen area when class names are long; vertical layout may be slightly easier to read and understand.

Yes/No) Part inheritance This command switches on or off the graphical display of archs that show *part inheritance*. It is useful for complex models, where the many arcs that indicate part inheritance may clutter the graphical display. By turning off part inheritance, the ordinary inheritance structure of classes and instances becomes more clear.

3.6 Popup menu

The popup menu appears if you click the right mouse button when the cursor is above a class or instance icon in the browser.

Add Class Add a new child class below the current node.

Add Instance Add a new child instance.

Add Part Add a new part within the current class or instance.

Rename Change name of class or instance.

Delete Delete the class.

Edit Edit class attributes.

Move Move the class to another position in the inheritance tree. (** check this*??)

Show Make a previously collapsed subgraph appear.

Collapse Make the appearance of the current subgraph be collapsed.

Connect Connect two nodes in the inheritance tree. (** check this???**)

Cancel Cancel this popup menu.

Figure 3.3: *Popup menu for class or instance editing.*

3.7 Mouse commands in the browser

Either of the *left*, *middle* or *right* mouse buttons can be used to activate a pulldown menu and select an alternative. However, within the graphical browser window they are used for separate functions:

Left button Click *left* on a class, instance or part node to select it and make its contents appear in an Emacs window.

Middle button Click the *middle* mouse button on a class or instance node to select it, and keep the button down while moving the mouse in order to change the position of the node. This can be used to *edit* the default *appearance of the inheritance graph* to obtain a nicer looking and more readable graph. Possibly changed node positions will be preserved when you save the model.

Right button Click the *right* button on any class, instance or part node, in order to make a popup menu (see Figure 3.3) of node operations visible.

3.8 Special ObjectMath mode commands in Emacs

The following two commands may be useful when editing ObjectMath class and instance definitions within Emacs windows.

Ctrl-x ` Find the next ObjectMath compilation error.

Ctrl-c % Query replace over all Emacs buffers, containing all classes and instances instances of the current model.

Chapter 4

Solution of a simple system of equations

The purpose of this chapter is to develop a very simple simulation application from a model mainly consisting of first order ordinary differential equations (ODEs), of the form:

$$\dot{x} = f(x, u, t)$$

Here x is a state vector and $\dot{x}$ the derivative of the state vector. Such as system can be solved numerically and/or symbolically, yielding the state vector as a function of time, x*(t)*.

Inspired by a similar example in [Cel91], we present an example model that describes the vertical motion of a rocket that is just about to perform a soft landing on the surface of the moon. The state vector contains three state variables, namely the altitude *h* over the lunar surface, the vertical component of the velocity *v*, and the mass *m* of the lunar lander. Derivatives are marked with an apostrophe ’, as for *h’(t)* which is the derivative of the altitude *h(t)*, and equal to *v(t)* – the vertical component of the velocity – as specified in equation `Eq[1]`. There is also the acceleration *a(t)* which is dependent on mass, thrust (`thrustFun`), and gravity (`gFun`).

The general structure of the *lunar* model appears below, together with a few equations. Object oriented concepts such as *class* and *instance* are explained in more detail in Chapter 6. Here we focus on the lunar example and its dynamic behavior, and rely on the readers’ intuition regarding object-oriented constructs.

```
model lunar;
  ...
class rocket(thrustFun, masslossrate, gFun)
  ...
  a[t_] :=  (1/m[t]) * (thrustFun - m[t]*gFun);
  Eq[1] :=  h'[t]  ==  v[t];
  Eq[2] :=  v'[t]  ==  a[t];
  Eq[3] :=  m'[t]  ==  -masslossrate * Abs[thrustFun];
end;

instance rocket1 inherits rocket(landing1`thrust[t], landing1`masslossrate,
                                 landing1`g[t])
end;

class planet(radius,mass)
end;

instance moon inherits planet(1.738*10^6,7.382*10^22)
end;
```

```
class landing(planet,rocket)
  ...
  g[t_] := G*planet`mass/(rocket`h[t]+planet`radius)^2;
end;

instance landing1 inherits landing(moon,rocket1)
  ...
end;
```

The model includes a general class `rocket` which contains the equations that determine the dynamic behavior of a rocket, which is inherited by the specific rocket instance called rocket1 that actually performs the landing.

There is also a general class `planet`, and a specialized instance called `moon` that has a radius of $1.738*10^6$ meters and a mass of $7.382*10^{22}$ kg.

There is a class `landing` which represents general properties of landings such as gravity depending on altitude *h(t)*, together with facilities for computing a solution (performing a simulation), i.e. computing the state vector component functions x1(t), x2(t) during a specified time interval for which the simulation is desired.

The landing instance `landing1` contains numeric parameters for a specific landing

4.1 Three methods for solution

There are basically three possibilities for solution of a system of ordinary differential equations in ObjectMath:

- Symbolic solution using the Mathematica function DSolve, giving a list of functions in symbolic form as the result.
- Numeric solution using the Mathematica function `NDSolve`, performing a numeric solution and giving a list of interpolate functions as the result.
- Numeric solution by generating Fortran90 or C++ code, which is compiled and linked together with an efficient external solver, and then executed on an appropriate machine.

The builtin symbolic solver DSolve is limited to rather small problems. The numerical solver NDSolve is appropriate for somewhat more complex examples such as the lunar landing model presented in this chapter. The third possibility, of generating code to be linked together with an external solver, can handle very large and computationally heavy problems which may need to execute on parallel computers to achieve acceptable computing time.

For the lunar landing example we will initially use `NDSolve`, for which the parameter list is shown below together with the parameter list of the symbolic solver `DSolve`. The system of equations represented by the list `{eqn1,eqn2,...}` will be solved for the variables `{x1,x2,...}` which are functions of t and will be computed in the time interval [`tmin..tmax`].

```
DSolve({eqn1,eqn2,...}, {x1,x2,...}, t]

NDSolve({eqn1,eqn2,...}, {x1,x2,...}, {t, tmin, tmax}]
```

4.2 Specifying a simulation model

To create the lunar model within the ObjectMath environment you first push the *New* button in the master window (Figure 3.1) to open a new model, specifying the name *lunar*. Then select the parent node and use the *Add class* command in the *Objects* pull-down menu or the *Add class* command in the popup menu (push the right button), to create the classes `rocket`, `planet` and `landing`. Then select `rocket` and use *Add instance* to add the instance `rocket1`. Do similarly with `planet` and `landing` to add the instances `moon` and `landing1`. Do not forget to specify class parameters when creating the class nodes, or alternatively add this later through the *Edit* command. Finally define the contents of each class/instance by first selecting the appropriate node and then typing in the contents.

Alternatively load the existing model (if you have it) from the file `lunar.omt`. The created inheritance tree for the *lunar* model appears in the browser as shown in Figure 4.1.

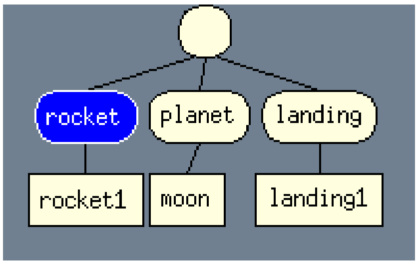


Figure 4.1: *The inheritance tree for the lunar model. Astract classes have round corners, whereas instantiated classes are rectangular.*

4.2.1 Detailed overview of the lunar model

Here we present a detailed overview of the *lunar* landing model. The model is structured in such a way that it easily can be generalized to more than a lunar landing. For example, landings of different versions of rockets on different planets such as Mars, the moons of Jupiter, etc. can conveniently be represented by creating additional instances of the classes `planet`, `rocket` and `landing`.

Model header

All ObjectMath models start with a model header section that specifies the model name and possibly specify some global quantities that are relevant for the whole model. Here the general gravitational constant *G*, and the time parameter *t* are given. It is necessary to declare *t* globally, otherwise the time parameters *t* used in the different classes will be regarded as disjoint time parameters, which will create problems for the solver.

```
model lunar;
  G := 6.672*10^-11;    (* General gravity constant *)
  t;                    (* time *)
```

Class *rocket* and instance *rocket1*

The class `rocket` embodies the equations of vertical motion for a rocket which is influenced by an external gravitational force `gFun`, and a force `thrustFun` from the rocket motor, acting in the opposite direction to the gravitational force, as in the expression for acceleration below.

$$a(t) = \frac{thrust(t) - m(t)\,g(t)}{m(t)}$$

Equations Eq[1] to E[3] are first-order differential equations stating well-known laws of motion between altitude *h(t)*, vertical velocity *v(t)* and acceleration *a(t)*:

$$\begin{aligned} h'(t) &= v(t) \\ v'(t) &= a(t) \\ m'(t) &= -\text{masslossrate}\ |\text{thrust(t)}| \end{aligned}$$

The derivative of the rocket mass, *m'(t)*, is negative since the rocket looses fuel mass proportional to the amount of thrust from the rocket motor.

The three differential equations are grouped in a list called `Equations`, which is exported from the class `rocket` (keyword `expose`) for later use by the solver `NDSolve`, whereas the rest of the definitions are hidden within the class (keyword `hide`).

```
class rocket(thrustFun, masslossrate, gFun)
 expose:
  Equations := {Eq[1], Eq[2], Eq[3]};
 hide:
  a[t_]  :=  (1/m[t]) * (thrustFun - m[t]*gFun);
  Eq[1] :=  h'[t]  ==  v[t];
  Eq[2] :=  v'[t]  ==  a[t];
  Eq[3] :=  m'[t]  ==  -masslossrate * Abs[thrustFun];
end;
```

The particular rocket instance called `rocket1` below, has been specialized to the level of `thrust`, `masslossrate` and gravitational force `g` specific to `landing1`, by passing these quantities as class parameters when instantiating `rocket`.

```
instance rocket1 inherits rocket(landing1`thrust[t], landing1`masslossrate,
                                  landing1`g[t])
end;
```

Class *planet* and instance *moon*

We have defined a general class `planet` with attributes radius and mass represented as class parameters, and a specific instance of `planet` called `moon` with mass and radius as the earth's `moon`.

```
class planet(radius,mass)
end;

instance moon inherits planet(1.738*10^6, 7.382*10^22)
end;
```

Class *landing* and instance *landing1*

The class landing contains definitions which are common to all landing instances. This includes the gravitation *g(t)* between the planet and the rocket according to the formula below for the gravitational force between two masses M_1 and M_2, separated by a distance d

$$G_{\text{Force}} = \frac{GM_1M_2}{d^2}$$

which gives the gravitational field *g(t)* as follows

$$g(t) = \frac{GM_{\text{Planet}}}{(h(t) + radius)}$$

The thrust function is a step function with an initial thrust force level `f1` during the time interval $[0..t_{\text{decrease}}]$ and a second thrust level `f2` during the time interval $[t_{\text{decrease}}..t_{\text{end}}]$. The step functions can conveniently be expressed using Mathematica conditional expressions If[*condition*,*true-ex-*

pr,false-expr]. The `masslossrate` determines the rate of lost mass per time unit and force unit when the rocket motor is fired. Six variables are “virtual”, i.e. they will not be given values until the class is instantiated.

All landings involve a planet and a rocket, which are represented as two class parameters.

```
class landing(planet,rocket)
 expose:
  masslossrate := 0.000277;  (* change constant massloss/sec/force-unit; *)
  thrust[t_] := If[t<tdecrease, f1, If[t<tend, f2, 0] ];
  g[t_] := G*planet`mass/(rocket`h[t]+planet`radius)^2;
 hide:
  tdecrease; (* virtual; time when thrust is reduced *)
  tend;      (* virtual; time when thrust ends *)
  tmax;      (* virtual; time when simulation ends *)
  f1;        (* virtual; initial thrust *)
  f2;        (* virtual; reduced thrust *)
  initialconditions; (* virtual *)
```

The statement below invokes the numerical ordinary differential equation solver `NDSolve`(*equations*,*variables*,*time-interval*) passing the list of differential equations and initial condition equations (concatenated by `Join`), the list of state variables to be solved for, and a list of the independent variable t and values for the solution interval [0,tmax].

The numerical result from `NDSolve` is represented as interpolate functions for the solved state variables during the specified solution interval.

```
  results := NDSolve[Join[rocket`Equations,initialconditions],
                     {rocket`m[t],rocket`h[t],rocket`v[t]},
                     {t,0,tmax} ];
```

The parameterless function `plot` will plot five diagrams of interesting quantities from the simulation. For each diagram, it invokes the function `auxPlot` passing the quantity to be plotted, and an appropriate title string.

```
  plot := (
    auxPlot[thrust[t], "Thrust"];
    auxPlot[rocket`m[t], "Mass"];
    auxPlot[g[t], "Gravity"];
    auxPlot[rocket`h[t], "Altitude"];
    auxPlot[rocket`v[t], "Velocity"];
  );
```

The function `auxPlot` accepts two arguments; the quantity to be plotted (`fun_`) and a partial title string (`title_`).

The display title is set to the string "`Lunar landing: `1`", where the ``1`` is substituted by the first parameter to `StringForm`, which happens to be the `title` argument.

The expression `Evaluate[fun /. results]` means that symbolic occurrences of state variables within `fun` are substituted by corresponding computed interpolate functions in `results`. For example, the symbolic expression `v[t]` would be substituted by the interpolate function that represents the computed values of `v[t]` during the simulation interval. This is then passed to the Mathematica `Plot` function to produce a diagram in the interval `[0,tmax]`.

```
  auxPlot[fun_, title_] :=
    Block[{$DisplayTitle := StringForm["Lunar landing: `1`", title]},
          Plot[Evaluate[fun /. results], {t, 0, tmax}, PlotRange -> All]];
```

This is the end of class landing.

```
end landing;
```

The landing instance `landing1` is a specific landing of `rocket1` on the `moon`. It assigns numerical values to parameters such as `tdecrease`, `tend`, `tmax`, `f1`, and `f2`.

The initial conditions are defined by three equations in the list `initialconditions`, specifying the initial altitude, velocity and mass of the rocket.

```
instance landing1 inherits landing(moon,rocket1)
 hide:
  tdecrease := 43.2;  (* time when thrust is reduced *)
  tend := 210;        (* time when thrust ends *)
  tmax := 230;        (* time when simulation ends *)
  f1  := 36350;       (* initial thrust *)
  f2  := 1308;        (* reduced thrust *)
  initialconditions := {rocket1`h[0] == 59404,
                        rocket1`v[0] == -2003,
                        rocket1`m[0] == 1038.358} ;
end landing1;
```

The complete lunar model

The complete lunar model follows below.

```
model lunar;
  G := 6.672*10^-11;    (* General gravity constant *)
  t;                    (* time *)

class rocket(thrustFun, masslossrate, gFun)
 expose:
  Equations := {Eq[1], Eq[2], Eq[3]};
 hide:
  a[t_]  :=  (1/m[t]) * (thrustFun - m[t]*gFun);
  Eq[1] :=  h'[t]  ==  v[t];
  Eq[2] :=  v'[t]  ==  a[t];
  Eq[3] :=  m'[t]  ==  -masslossrate * Abs[thrustFun];
end;

instance rocket1 inherits rocket(landing1`thrust[t], landing1`masslossrate,
                                 landing1`g[t])
end;


class planet(radius,mass)
end;

instance moon inherits planet(1.738*10^6, 7.382*10^22)
end;


class landing(planet,rocket)
 expose:
  masslossrate := 0.000277;  (* change constant massloss/sec/force-unit; *)
  thrust[t_] := If[t<tdecrease, f1, If[t<tend, f2, 0] ];
  g[t_] := G*planet`mass/(rocket`h[t]+planet`radius)^2;
 hide:
  tdecrease; (* virtual; time when thrust is reduced *)
  tend;      (* virtual; time when thrust ends *)
  tmax;      (* virtual; time when simulation ends *)
  f1;        (* virtual; initial thrust *)
  f2;        (* virtual; reduced thrust *)
  initialconditions; (* virtual *)
```

```
  results := NDSolve[Join[rocket`Equations,initialconditions],
                     {rocket`m[t],rocket`h[t],rocket`v[t]},
                     {t,0,tmax} ];

  (* plot will compute results and display in five graphs *)
  plot := (
    auxPlot[thrust[t], "Thrust"];
    auxPlot[rocket`m[t], "Mass"];
    auxPlot[g[t], "Gravity"];
    auxPlot[rocket`h[t], "Altitude"];
    auxPlot[rocket`v[t], "Velocity"];
  );

  auxPlot[fun_, title_] :=
    Block[{$DisplayTitle := StringForm["Lunar landing: `1`", title]},
           Plot[Evaluate[fun /. results], {t, 0, tmax}, PlotRange -> All]];
end landing;


instance landing1 inherits landing(moon,rocket1)
 hide:
  tdecrease := 43.2;  (* time when thrust is reduced *)
  tend := 210;        (* time when thrust ends *)
  tmax := 230;        (* time when simulation ends *)
  f1  := 36350;       (* initial thrust *)
  f2  := 1308;        (* reduced thrust *)
  initialconditions := {rocket1`h[0] == 59404,
                        rocket1`v[0] == -2003,
                        rocket1`m[0] == 1038.358} ;
end landing1;
```

4.3 Running a simulation

To run this simulation, first compile the lunar ObjectMath model into Mathematica code to be down-loaded into the Mathematica process. This is performed by choosing the *Compile* command in the *Model* pull-down menu (see Section 3.1).

Next make the *Interaction* Emacs window appear by selecting *Interaction* in the *Model* pull-down menu, if it is not already visible.

In the Interaction window, invoke the `plot` function of `landing1` by typing:

```
landing1'plot
```

followed by *Meta-return* or *Esc return*. The `plot` function will invoke the `auxPlot` function for each plot, which in turn calls the parameterless function `results` that calls `NDSolve` to perform the actual solution of the system of equations in the lunar model.

Remember that the `plot` function is inherited from `landing` into `landing1`. Only code in instances is actually turned into Mathematica code. Therefore it is necessary to call `landing1'plot` and not `landing'plot`.

The reader might also wonder about how `plot` can be accessed from the *Interaction* window even though it is specified in the `hide` section of local definitions in class `landing`. Such definitions are not normally visible outside their own class, and thus are not accessible from other classes or instances. However, in the ObjectMath environment all class/instance attributes are accessible from the Interaction window, irrespective whether they are hidden or exposed. This feature simplifies debugging of class and instance definitions.

4.4 Graphic display of lunar landing simulation

The result of the lunar landing simulation is displayed in five diagrams, Figure 4.2 through Figure 4.6.

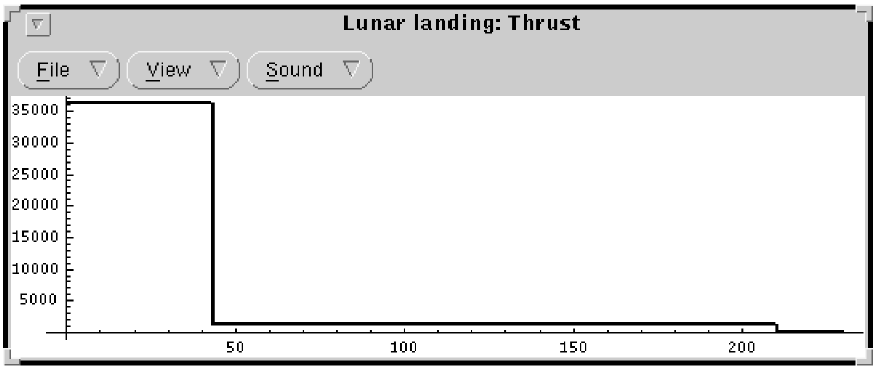


Figure 4.2: *Thrust from the rocket motor, with an initial high thrust f1 followed by a lower thrust f2.*

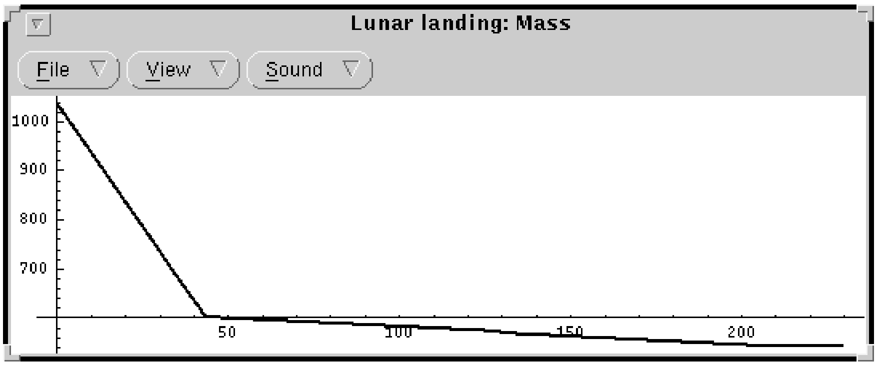


Figure 4.3: *Rocket mass decreases when the fuel is consumed.*

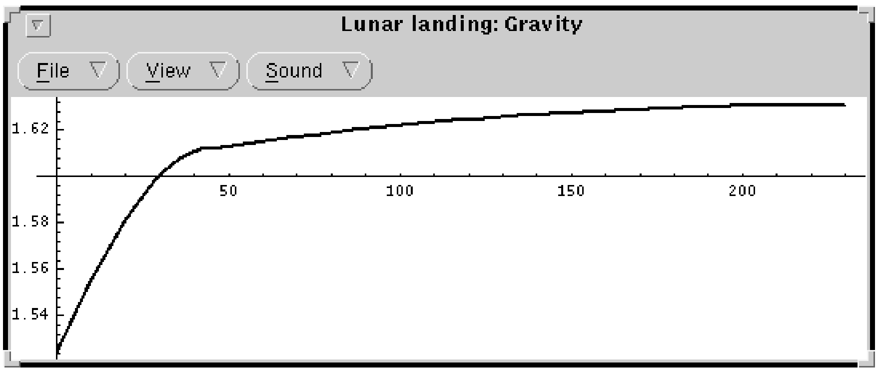


Figure 4.4: *Gradually increasing gravity when the rocket approaches the lunar surface.*

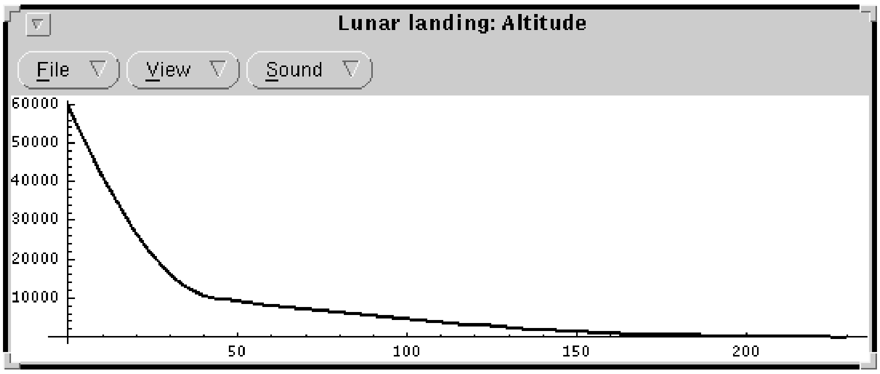


Figure 4.5: *Altitude over the lunar surface.*

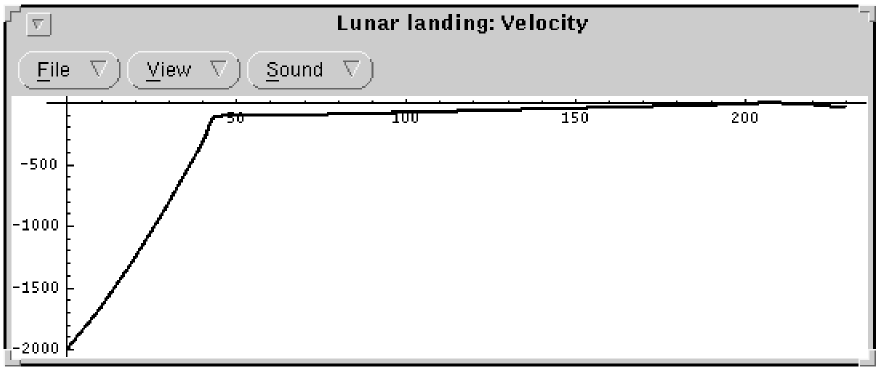


Figure 4.6: *Vertical velocity relative to the lunar surface.*

An interesting observation can be made in Figure 4.5 and Figure 4.6. At the very end of the simulation, the altitude becomes slightly negative and the vertical velocity downwards increases slightly from a zero value.

This is due to a deficiency in the simulation model. At touchdown, when the altitude and the vertical velocity have been reduced to zero, the rocket motor was turned off. However, the gravity force is still acting on the rocket and increasing the negative vertical velocity.

In real life, however, the gravity force is balanced out by a vertical force from the surface of the moon touching the landed rocket. This force we forgot to put into our simulation model, which causes the rocket to continue to accelerate into the interior of the moon in our model.

Chapter 5

ObjectMath language extensions

ObjectMath is an object oriented extension to the Mathematica computer algebra language, as has already been mentioned several times. Since the Mathematica part of the ObjectMath language is described in detail in [Wol91] we focus on aspects which are specific to Objectmath in this manual. First we give a short overview of language extensions mostly related to object orientation. In a later chapter we describe the compilable subset of ObjectMath, which can be translated into efficient C++ or Fortran90 code.

This chapter briefly describes extensions and restrictions of the ObjectMath language as compared to Mathematica 2.0. More detailed instructions on how to use the language extensions provided by ObjectMath are provided in Chapter 6 of this manual. The syntax of ObjectMath language extensions is described informally below, and defined formally in Appendix G using BNF notation.

5.1 Structure

Models expressed in ObjectMath conform to certain rules regarding their structure. A model starts with a model header, followed by some global declarations, followed by class/instance declarations which contain equations, variables and functions, etc. Such class declarations are the essence of a model.

5.1.1 The structure of models

An Objectmath model has the following general structure:

```
<model header>
...
<package import declarations>
...
<global variable or function declarations>
...
<sequence of class or instance declarations>
```

As an example, the SphereCyl model in Appendix B has the following structure:

```
model SphereCyl;
  packages "Vectors`", "Subst`", "Misc`";
```

```
  g; (* Gravity constant *)

<sequence of class or instance declarations>
...
```

5.1.2 The structure of class/instance declarations

A class or instance declaration in ObjectMath has the following general structure:

```
<class header>
...
<optional part declarations>
...
<optional sequence of local blocks>
...
<optional initialization section>
...
<end keyword>
```

A *class header* starts with one of the keywords `class` or `instance`. A *local block* starts with one of the keywords `expose`, `reveal` or `hide` followed by a colon. The optional *initialization section* starts with the keyword `begin`. Finally the class/instance declaration is ended by the keyword `end`. Two example class declarations follow below:

```
  class Bicycle(C,P)
    expose:
    part Frontwheel inherits Wheel(P);
    part Rearwheel inherits Wheel(P);
    part Frame inherits Body;
    ...
  end Bicycle;

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

5.2 Restrictions compared to Mathematica

Before describing the ObjectMath language extensions compared to Mathematica, it is useful to be aware of a few restrictions.

5.2.1 Implied multiplication

Implied multiplication is not allowed in ObjectMath. This Mathematica feature can be convenient for small expressions, but in large models it sometimes give rise to bugs which are hard to find. A conversion facility is provided for importing Mathematica code into ObjectMath models, which inserts a multiplication operator in all places where such an operator is implied, for a syntactically correct Mathematica program.

Implied multiplication of a and b, such as the following, is not allowed in ObjectMath:

```
  a b
```

Correct ObjectMath:

```
a * b
```

5.2.2 Placement of declarations

Within a class or instance, declarations come in the following order:

First comes the `part` declarations, if there are any; then zero or more local declaration blocks, where each local block starts with a visibility keyword (`hide`, `reveal` or `expose`) followed by an optional access list, followed by a number of declarations/definitions of variables, symbols, functions and equations, where functions and equations may only be defined using `SetDelayed` (`:=`).

Finally there may or may not be an optional `begin` followed by an optional sequence of arbitrary Mathematica expressions. The class/instance always ends with the `end` keyword.

5.2.3 Name structure

As ObjectMath is based on Mathematica, most of the syntactic properties are the same. However, the lexical syntax of names differ in the following ways:

- The character $ is not allowed in identifiers or symbols.
- A declared name may not start with a ` (back tick).
- A most one set index written within square brackets ([ ]) may be added to each element in the "context" part of a name. The set index is a list of expressions. They are typically used as indices in mathematical notation. For instance, the following are legal identifiers in ObjectMath:

```
a[i]`b`c
a[i*j + k[x]]`b[i]`c
a`b[k]`c
```

There is an additional restriction for typed ObjectMath variables declared in local blocks. Such variables are often intended to be translated to variables in generated Fortran90 or C++. Names that contain a set index within square brackets may only have "simple names" as indices, i.e. names that do not contain backticks or most other special Mathematica characters. Thus, these variable names are allowed in local blocks:

```
a[i]`b`c, a[3,5,j], b[g,h[r]]
```

but the following are not allowed because of the special characters ` and +:

```
a[hh`[3]], b[a+5]
```

Untyped ObjectMath variables are however not affected by this restriction since they are primarily intended for use in symbolic processing, and not intended to be translated to corresponding variables in other languages.

5.3 Object-orientation

class

A `class` is a collection of related equations, functions, variables, etc. that describe some (abstract) object. See Chapter 6 for more information.

instance

An `instance` is a class declaration which also gives rise to an instance of this class. Only instances give rise to Mathematica code when a model has been translated and downloaded to Mathematica. See Chapter 6 for more information.

end

The `end` keyword ends a class or instance declaration. The name of the class or instance may follow the end keyword. Example:

```
end SimpleRing;
```

inherits

The inherits keyword is used to denote single or multiple inheritance from a class or instance to another class, instance or part. See Chapter 6 for more information. Example:

```
class Ring(S, B) inherits Body(S, B)
```

part

A part is a named item in a class or instance. It typically inherits and instantiates another class or instance. See Chapter 6 for more information. An example:

```
...
part Frontwheel inherits Wheel(P)
part Rearwheel inherits Wheel(P)
...
```

this

The keyword `this` is a reserved word which is used to denote the current object (instance or class).

5.4 Visibility and local blocks

expose:

The `expose` keyword starts a local block of declarations with *read-write* access, where the definitions (variables) can be both read and written. An example including both `hide`, `reveal` and `begin` follows below:

```
instance Y[i] inherits X({1,2,3})
hide:
  e := a + b;
  array [3,3] f;
reveal:
  g[h] := 42 + d;
begin
  l := m + a;
end;
```

reveal:

This keyword starts a local block of declarations with *read-only* access, where the definitions can be read (i.e accessed) but not written/redefined. The end of the local block is at another visibility keyword, or `begin` or `end`.

hide:

This keyword starts a local block of declarations with *no* access, where the definitions are not visible outside.

begin

The optional `begin` keyword is used to denote the beginning of an "*initialization*" or "*compile-time-executable* code" section of a class or instance. The expressions in this section are evaluated each time the ObjectMath model is compiled and loaded into Mathematica. This section can contain all possible Mathematica expressions, including immediate evaluation (`=`, `Set`) expressions. Its use is generally not recommended when designing well-structured mathematical models but may sometimes be needed for specific purposes. Example:

```
instance ...
...
begin
  CreateSymbol(a);
end;
```

5.5 Modularity and scoping

model

The mandatory `model` keyword starts the definition of an ObjectMath model. Example:

```
model Example;

a := 17;
symbol b;

class X(c)
....
end;
```

packages

This keyword starts an optional section for importing Mathematica (library) modules. The definitions within such modules may be referenced and called from within the ObjectMath model. Arbitrary Mathematica code may be present in such a library module.

However, if C++ or Fortran90 code should be generated for a function from the library, either the source code of the function is incorporated in the ObjectMath model, or a typed wrapper function with symbolic parameters passed to the library function is defined in the model. Then code can be generated using CompileEvaluateFunction (see Chapter 11) on the wrapper function, since the library function will return a symbolic expression that defines its computation. Example:

```
packages "Vectors`", "Subst`", "Misc`";
```

This will load Mathematica modules from files `Vectors.m`, `Subst.m`, `Misc.m`.

extern

Declaration of external variables and functions in C, C++ or Fortran, which can be linked together with and called from the generated code of a compiled ObjectMath model. Example:

```
real Sq2[int x_] := extern "Sq2";
```

5.6 Special operators and symbols

The operator :==

This operator works as a delayed assignment which stops the automatic symbolic expansion in Mathematica that replaces left-hand-sides by right-hand-sides. This is needed, for example, if part of a model refers to an array that is defined as an array of components, and need to deal with these components, whereas other parts of the model should only refer the array as named un-divisible entity that should not be expanded into its component. The latter case is needed, for example, when generating Fortran90 or C++ expression code that uses built-in array functions in those languages. The :== operator works as an ordinary assignment operator, but limits symbolic expansion. In translated code produced in C++ or Fortran90, it works as an ordinary assignment operator.

```
avar :== b+c;
```

Thus, the symbol `avar` can be referenced without being evaluated into `b+c`. There is also a built-in function `Lookup`, that will force evaluation of `avar` if needed.

```
evaluated = Lookup(avar);
```

This will evaluate the previously delayed evaluation of `avar`.

symbol

This denotes names which are global for the whole model, and are just used as names, not as variables. Thus, they may not be assigned. A typical use is as symbolic indices.

```
symbol foo;
```

The operator =>

This is an indirection operator, similar to ‘ (backtick) in Mathematica. However, the evaluation rules are different:

```
a=>b
```

is the same as

```
a`b
```

but evaluates `a` before accessing the item `b`.

The standard Mathematica semantics is to regard `a`b` as one name. The new operator => is useful if the variable `a` can denote different objects, and a need to be evaluated to obtain the component of the object (similar to the -> dereferencing operator in C and C++).

5.7 Declarations and arrays

real

This keyword denotes double precision real:

```
real r;
```

int

This keyword denotes integer.

```
int i;
```

bool

This keyword denotes booolean.

```
bool b;
```

array

This keyword specifies the declaration and creation of an array with the given dimension and size, and initially filled with symbolic elements.

```
real array[3]          (* a one-dimensional array of 3 real numbers*)
real array[5,4,10]     (* a three-dimensional real array *)
int  array[4,4]        (* a 4x4 array of integers *)
bool array[3,3]        (* a 3x3 array of booleans *)
```

once

This operator forces the right-hand side to only be evaluated once, for each value of the in-parameter that a function (f) is called with. The result is saved and returned next time the name is referenced. Two examples follow:

```
once a := b + c;
once f(x_) := b + c;
```

Chapter 6

Classes, instances, inheritance, parts, variants

In this chapter we use a number of small examples to explain ObjectMath language constructs such as *class*, *instance*, *part* and their use to express inheritance, composition and object creation. Many examples refer to the SphereCyl model that can be found in Appendix B.

6.1 Classes and instances

A *class* declaration declares a class which can be used as a template when creating objects. ObjectMath classes can be parameterized. Classes may inherit from one or several other classes. Objects are then declared with an *instance* declaration.

The *instance* declaration is the only way to create an object, i.e. objects cannot be created dynamically at run-time. In a traditional sense, the ObjectMath *instance* declaration is both a declaration of a class and a declaration of one object (instance) of this class. This makes the declaration of classes with one instance compact. It is possible to inherit from classes implicitly declared by an ObjectMath *instance* declaration, just as from classes declared with a *class* declaration. The bodies of ObjectMath *class* and *instance* declarations contain formulae, equations and functions. Mathematica syntax is used to express these.

6.2 Single inheritance

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 6.1, there are additional classes which describe interactions between bodies and coordinate systems. The full inheritance hierarchy is displayed in the lower part of Figure 2.3 and the full class definitions are available in Appendix B. Note that the inheritance hierarchy usually is edited graphically so that the user does not have to write the class headers by hand.

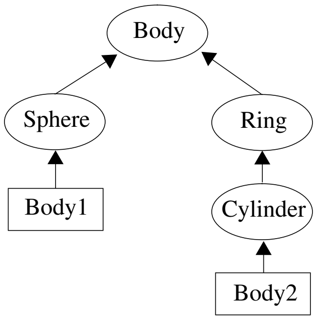


Figure 6.1: *An inheritance hierarchy of classes for modeling bodies with different geometries such as cylinders and spheres.*

At the top of this inheritance sub-hierarchy is the class `Body`, which contains definitions and functions common to all bodies. This includes the virtual 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 specialized to `r[C1][u_,v_]` since `C1` is the body-centered coordinated system of `Body1`.

```
class Body(S, B)
 expose:
(* Geometry defined through a parametric surface *)
 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 };
(* 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 6.2. 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 B.

```
class Sphere(S, B) inherits Body(S, B)
 expose:
 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})
 expose:
 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;
```

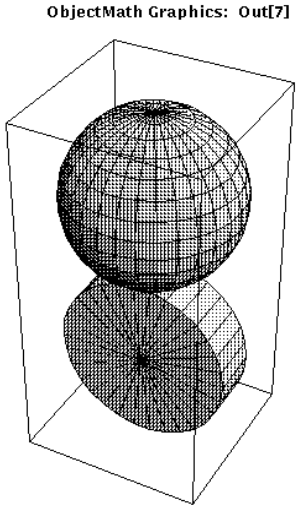


Figure 6.2: *The sphere rests on top of the cylinder.*

6.3 Multiple inheritance

The first version of the ObjectMath language supported only single inheritance. Later, it was enhanced with features for multiple inheritance and modeling part-of relations between objects. Both of these features have turned out to be important in realistic application models.

Multiple inheritance is useful when combining independent concepts, as shown in Figure 6.3. Because material properties and geometry are independent, there are no collisions between inherited definitions. If we were forced to use a single-inheritance hierarchy, as shown in Figure 6.4, we would have to repeat the equations describing material properties twice, which is bad model engineering. It also precludes the creation of pure material library classes that can be combined with other classes. Another disadvantage of a single-inheritance model structure is that it is harder to maintain when changes are introduced.

The general form of multiple-inheriting class declarations is

```
class Child inherits Parent1,Parent2,... ParentN
   . . .
end Child
```

If there are conflicts between inherited definitions (if they have the same name, for example) definitions from `Parent1` will override definitions from `Parent2`, which will override definitions from `Parent3`, and so on. Single inheritance would occur if only `Parent1` was present.

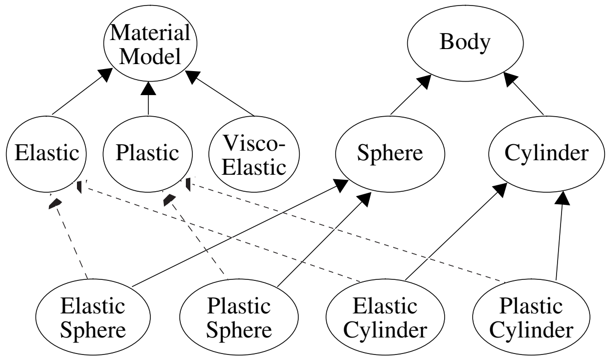


Figure 6.3: *Multiple inheritance hierarchy of bodies of different materials and geometries.*

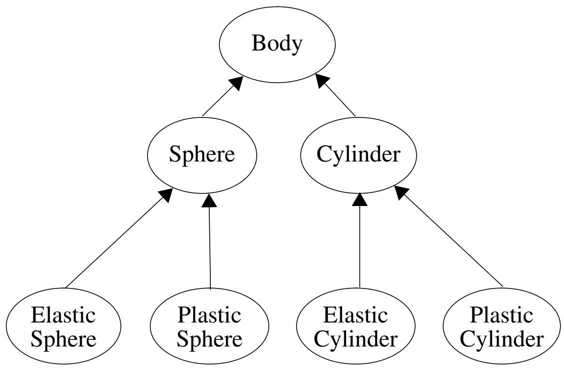


Figure 6.4: *Single inheritance version of the material-geometry model of Figure 6.3.*

The following examples are taken from the material-geometry model in Figure 6.3.

```
class Sphere(S,B) inherits Body(S,B)
  expose:
  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
  expose:
  Force := k1 * delta;
end Elastic;

class Plastic inherits Material_Model
  expose:
  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 one in which a numerical integration method is inherited into classes from two separate inheritance hierarchies, as shown in Figure 6.5. The entities inherited from class `IntegrationMethodA` will typically be a combination of entities such as procedural code, transformation rules, and so on.

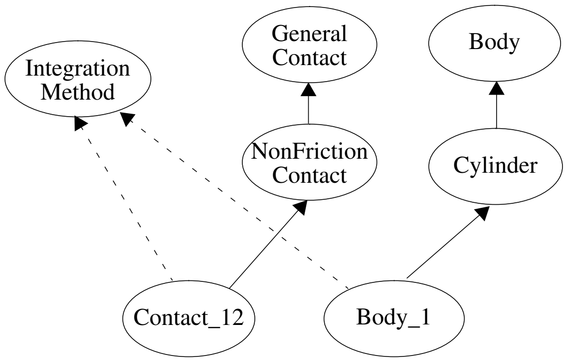


Figure 6.5: *Example of multiple inheritance of a numerical integration method into two different classes. Here to be used for integrating forces or volumes. One class contains contact equations; another contains volumes, moments and equilibrium equations.*

6.4 Sets of objects

To create a set that contains a variable number of objects from one instance declaration, you add an index variable in brackets to the instance name. This creates a variable number of similar objects (for example, the rolling elements in a rolling bearing, or the electrons around an atomic nuclei). To represent differences among such objects, ObjectMath uses functions (methods) that are dependent upon the index variable. This lets you do computations that have a symbolic number of elements in the set.

6.5 Part-of relations

The *part-of* relation is important for modeling objects which are composed of other objects. Remember that this concept is independent of 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 contains inner ring, outer ring, rolling elements, lubrication fluid, etc.

The ObjectMath syntax for expressing composition using the *part-of* relation is shown below for an example `Bicycle` class:

```
class Bicycle(C,P)
  ...
  part Frontwheel inherits Wheel(P);
  part Rearwheel inherits Wheel(P);
  part Frame inherits Body;
  ...
end Bicycle;
```

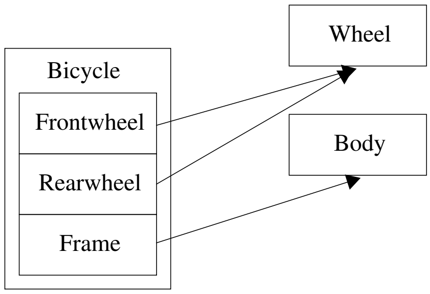


Figure 6.6: *A bicycle example class using the part-of relation and inheritance*

Another example is a small section of a class from a rather advanced model of four-body interaction developed at SKF:

```
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;
```

6.6 Variants

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.

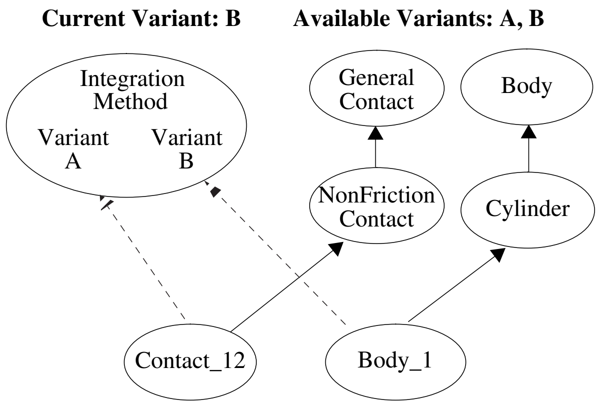


Figure 6.7: *The same example as in Figure 6.5 but with two available variants A and B. The currently active variant of the model is B, which will select Variant B of Integration Method and the default of other classes.*

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 a mechanism is available 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 6.7. 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 currently works 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.

```
class Plastic inherits Material_Model
expose:
#ifdef VariantA
  Force := k2 * LimitA - 35
#elseif VariantB
  Force := k3 * Limit +100
#elseif
  Force := k2 * Limit
#endif
end Plastic;
```

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 part-of structure.

Chapter 7

A Model of a 2D Rolling Bearing

Lars Viklund, Patrik Hägglund and Peter Fritzson

In this chapter we exemplify the ObjectMath language by describing a two-dimensional dynamic model of a cylindrical rolling bearing. This model has been developed as a test case for the ObjectMath code generator and is a reimplementation of a simulation model which have previously been implemented in C++ by SKF. The intention is that the performance of the generated code should be directly compared with the performance of the hand written code. The bearing consists of an outer ring, an inner ring and a single roller. However, the model can easily be extended to include an arbitrary number of rollers. Figure 7.1 shows the geometry of the bearing. The *z*-axis is pointing inwards in the picture and the coordinate system is right-oriented. Angles are measured clockwise from the *x*-axis.

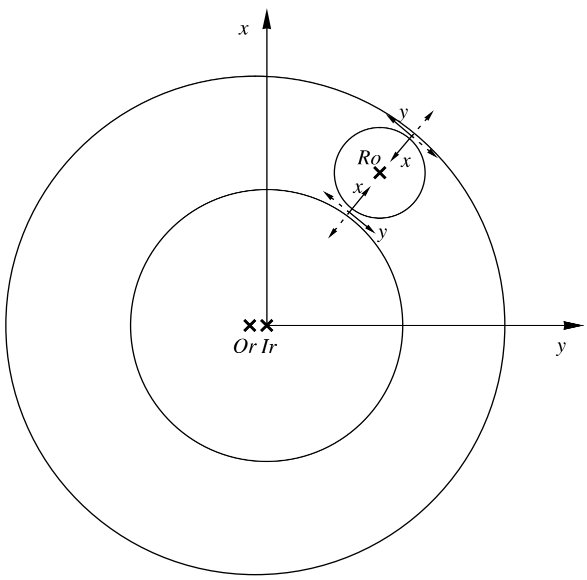


Figure 7.1: *Geometry of the bearing*

All the bodies (the two rings and the roller) are defined in a global coordinate system. In this particular model the inner ring is fixed: its origin coincides with the origin of the global coordinate system (GCS). The position and rotation of the outer ring are inputs to the problem to be solved. We are mainly interested in the motion of the roller which can be described by a system of ordinary differential equations. The class hierarchy of the model is shown in Figure 7.2.

In this figure we use the same graphical representation as in the graphical editor in the ObjectMath programming environment. The arrows represents inheritance and the *part* relation is shown by drawing the boxes representing the parts inside the oval representing the class they belong to. For comparison, in Appendix F the same structure is shown using Coad and Yourdon's OOA notation.

As can be seen in the figure, the object which models the contacts between the rings and the roller is a part of the object representing the roller. An alternative would be to view the contact objects as parts of the bearing object, together with the rings and the roller. One advantage of modeling the way shown in the figure is that it makes it somewhat easier to extend the model to include an arbitrary number of rollers.

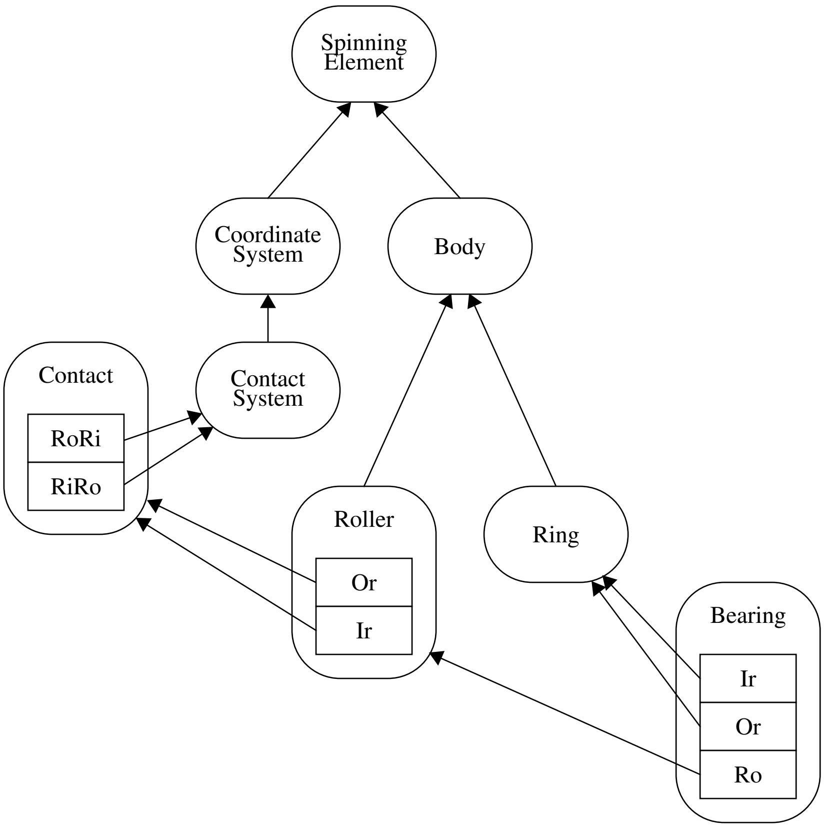


Figure 7.2: *Classes in the bearing model*

The abstract class `SpinningElement` models an object (not necessarily a physical body) which rotates. This class introduces two instance variables: `Origin` and `Turn` which denote the origin and rotation of the object relative to the global coordinate system. The actual definitions of these are given in subclasses inheriting from `SpinningElement` (c.f. virtual member functions in C++ or deferred routines in Eiffel.)

```
class SpinningElement
  expose:
  Origin;
  Turn;
end SpinningElement;
```

A local coordinate system is modeled by the class `CoordinateSystem` which inherits from `Spin-`

ningElement. The most important attribute of this class is a function for transforming a vector, point or velocity between the local and the global coordinate system and vice versa. This function is called Transform. For instance, to transform a velocity v, given in the local coordinate system represented by an object C, to the global system one would write: C`Transform[v, from, velocity].

Actually, only the vector transformation is used in this particular model, but the class is designed to be general so that it can be reused in other models. Pattern matching is used when defining the Transform function as can be seen below. This is just a matter of personal taste, one could equally well use If expressions instead.

```
class CoordinateSystem(t) inherits SpinningElement
  expose:
  (* Transformation matrix from local to global system *)
  X := {  Cos[Turn[[3]]], Sin[Turn[[3]]], 0 };
  Y := { -Sin[Turn[[3]]], Cos[Turn[[3]]], 0 };
  Z := {              0,              0, 1 };
  TransformationMatrix := Transpose[{ X, Y, Z }];

  vOrigin := Dt[Origin, t];   (* velocity of origin *)

  Transform[a_, from, type_] :=
    TranslateAfter[TransformationMatrix . a, from, type];
  Transform[a_, to, type_] :=
    TranslateAfter[Transpose[TransformationMatrix] . a, to, type];
  SetAttributes[Transform, HoldAll];

  TranslateAfter[a_, dir_, vector]   := TranslateAfter1[a, dir, {0, 0, 0}];
  TranslateAfter[a_, dir_, point]    := TranslateAfter1[a, dir, Origin];
  TranslateAfter[a_, dir_, velocity] := TranslateAfter1[a, dir, vOrigin];

  TranslateAfter1[a_, from, trans_] := a + trans;
  TranslateAfter1[a_, to,   trans_] :=
    a - Transpose[TransformationMatrix] . trans;
end CoordinateSystem;
```

The parameter t of the CoordinateSystem class denotes the time and the instance variable vOrigin denotes the velocity of the origin of the coordinate system.

CoordinateSystem is inherited from by the class ContactSystem which defines Origin and Turn (which were introduced but not defined in the class SpinningElement) for a local coordinate system. Two such coordinate systems are used for modeling each contact between two bodies. Thus, this class also defines the force and torque in the local coordinate system as well as in the global coordinate system (by calling the function Transform defined in the superclass). Parameters to this class are body1, which denotes the body the calculations are done for, as well as body2 which is the other body involved in the contact. Another parameter is contact which denotes the object modeling the whole contact, an instance of the class Contact described below.

```
class ContactSystem(body1, body2, contact, t) inherits CoordinateSystem(t)
  expose:
  (* In GCS *)
  RO := body2`Origin - body1`Origin;
  RO1 := body2`rsign * RO;  (* radius direction *)
  RO2 := - body1`rsign * body2`rsign * RO; (* coordinate system direction *)
  Origin := contact`CP;
  (* First axis pointing towards the body  *)
  Turn := ArcTan[ RO2[[1]], RO2[[2]] ] * {0, 0, 1};
  vBodyG := Dt[body1`Origin, t];          (* velocity of body1 in GCS *)
  vRotG := Transform[vRot, from, vector]; (* velocity of rotation *)
  vContG := vBodyG + vRotG;               (* velocity of the contact *)
```

```
  vRel := vContG - contact`vCP; (* velocity of body relative contact point *)
  ForceG := Transform[Force, from, vector];
  TorqueG := Transform[Torque, from, vector];

  (* Scalars *)
  fi := ArcTan[ RO1[[1]], RO1[[2]] ]; (* rotation angle *)
  (* radius with sign, depends on whether the surface is concave or convex *)
  sradius := body1`rsign * body1`Radius[ fi - body1`Turn[[3]] ];
  lever := sradius + contact`Delta / 2;
  TorqueRoll := Roll[ contact`ForceCollision, vSum[[2]] ];

  (* In local coordinate system, but fixed *)
  vRot := CrossProduct[ Dt[body1`Turn, t ], { -lever, 0, 0 } ];
  vBody := Transform[ vBodyG, to, vector ];
  vCont := vBody + vRot;  (* velocity at contact point on the body1 *)

  (* In local coordinate system *)
  vSum := Transform[ contact`vSumG, to, vector ] // Simplify;
  Force := { contact`ForceCollision, contact`ForceSlip, 0 } // Simplify;
  TorqueSlip := CrossProduct[ {-lever, 0, 0}, Force ];
  Torque := TorqueSlip + { 0, 0, TorqueRoll };

  (* Internal *)
  (* F vertical force, v horisontal speed, returns torque *)
  Roll[F_, v_] := F * 21*10^-7 * SmoothSign[v];
end ContactSystem;
```

As mentioned above, a contact between two bodies is modeled by the class `Contact`. An object of this class consists of two parts of the class `ContactSystem`, one for each of the two bodies involved in the contact. Most of the mathematics for the contact is contained in this class. For instance, the instance variables `ForceCollision` and `ForceSlip` represent the force resulting from the collision between the bodies and friction between the bodies respectively. In this particular model, the force `ForceCollision` is approximated with zero when the gap between the bodies is larger than zero.

```
class Contact(body1, body2, t)
  expose:
  part RoRi inherits ContactSystem(body1, body2, this, t);
  part RiRo inherits ContactSystem(body2, body1, this, t);

  (* Scalars *)
  (* distance between the origins of the bodies, with sign *)
  dist := body1`rsign * body2`rsign * Norm[`RoRi`RO] // Simplify;
  Delta := dist - (`RoRi`sradius + `RiRo`sradius);
  ForceElastic := Elastic[ Delta, body1`rsign * body1`radius, Infinity,
                           body1`rsign * body2`radius, Infinity ];
  ForceCollision := SqueezeFactor[ vDelta[[1]] ] * ForceElastic;
  ForceSlip := Slip[ ForceCollision, vDelta[[2]] ];

  (* In GCS *)
  (* contact point *)
  CP := (`RiRo`lever * body1`Origin + `RoRi`lever * body2`Origin) / dist;
  vCP := Dt[CP, t]; (* velocity of the contact point *)
  (* sum of the relative velocity of the two bodies *)
  vSumG := `RoRi`vRel + `RiRo`vRel;

  (* In GCS, but moving as RoRi and RiRo *)
  (* In the RoRi and RiRo systems *)
  vDelta := `RoRi`vCont + `RiRo`vCont // Simplify; (* symmetry *)
```

```
  (* Internal *)
  (* constants *)
  Emod = 226*10^9;
  CDry = 8*10^-1;
  Mu0 = 1*10^-1;
  gamma = 8*10^3;
  (* d gap, r1, r2 radii, returns vertical force *)
  Elastic[d_, rx1_, ry1_, rx2_, ry2_] := 0 /; d > 0;
  Elastic[d_, rx1_, ry1_, rx2_, ry2_] :=
  2^(3/2)/3 * Emod / ( 1/rx1 + 1/ry1 + 1/rx2 + 1/ry2 ) * (-d)^(3/2) /; d <= 0;
  (* v vertical speed, returns approx 1 *)
  SqueezeFactor[v_] := 1 - CDry * SmoothSign[v];
  (* F vertical force (>= 0), v horisontal speed, returns horisontal force *)
  Slip[F_, v_] := - Mu0 * SmoothSign[v, gamma] * F;
end Contact;
```

Properties which are common for all bodies in the model are collected in the class `Body`. The function `Radius` defines the radius of the body as a function of the angle from the *x*-axis. In the class `Body` it is a constant, but as we will see, it might be redefined in subclasses of `Body`. The instance variable `rhs` denotes the right hand sides of the ordinary differential equations for the body. As usual, the class parameter `t` denotes the time.

```
class Body(t) inherits SpinningElement
  expose:
  Origin = { R[t] * Cos[Fi[t]], R[t] * Sin[Fi[t]], 0 };
  Turn = { 0, 0, T3[t] };
  Radius[ang_] := radius;
  radius;
  Mass;
  Inertia = Mass * radius^2 / 2;
  Force;
  Torque;
  g = 981*10^-2;
  ForceExternal := { -g * Mass, 0, 0};

  h1 := { Cos[Fi[t]], Sin[Fi[t]], 0 } . Force / Mass + R[t] * Fi'[t] ^ 2;
  h2 := ({ Cos[Fi[t]], -Sin[Fi[t]], 0 } . Force / Mass
          - 2 * Fi'[t] * R'[t]) / R[t];
  h3 := Torque[[3]] / Inertia;
  rhs := { h1, h2, h3 };
end Body;
```

The class `Ring` is a specialization of the class `Body`. Here `Radius` is redefined; a sine wave is added to the base radius to model imperfections in a real ring.

```
class Ring(contact, t) inherits Body(t)
  expose:
  Radius[ang_] := radius + Awaves * Sin[waves * ang];
  Awaves;
  waves;
  Force := contact`RiRo`ForceG + ForceExternal;
  Torque := contact`RiRo`TorqueG;
end Ring;
```

The class `Roller` is also a specialization of the class `Body`. It contains the two objects modeling the contacts with the outer and inner ring.

```
class Roller(or, ir, t) inherits Body(t)
  expose:
  part Or inherits Contact(this, or, t);
  part Ir inherits Contact(this, ir, t);
```

```
  Force := Ir`RoRi`ForceG + Or`RoRi`ForceG + ForceExternal;
  Torque := Ir`RoRi`TorqueG + Or`RoRi`TorqueG;
end Roller;
```

Finally, the class Bearing models a complete bearing consisting of an outer ring (`Or`), an inner ring (`Ir`) and a roller (`Ro`).

```
class Bearing
  expose:
  part Or inherits Ring(Ro`Or, this);
  part Ir inherits Ring(Ro`Ir, this);
  part Ro inherits Roller(Or, Ir, t);
  t; (* Time *)
end Bearing;
```

The `Bearing` class can then be used when defining a particular bearing by assigning values to the instance variables:

```
instance Bearing1 inherits Bearing
  begin
  clearance = -10*10^-6;
  Ro`radius = 5*10^-3;
  Ro`rsign = 1; (* surface out from center *)
  Ro`Mass = 2*10^-3;
  Ir`Fi[t_] = 0;
  Ir`R[t_] = 0;
  Ir`T3[t_] = 0;
  Ir`radius = 30*10^-3;
  Ir`rsign = 1; (* surface out from  center *)
  Ir`Awaves = 5*10^-6;
  Ir`waves = 9;
  Ir`Mass = Infinity;
  Or`Fi[t_] = 0;
  Or`R[t_] = 2*10^-6; (* the outer ring is displaced *)
  Or`T3[t_] = 10*t;
  Or`radius = `Ir`radius + 2 * `Ro`radius + clearance;
  Or`rsign = -1; (* surface towards center *)
  Or`Awaves = 10*10^-6;
  Or`waves = 5;
  Or`Mass = Infinity;
end Bearing1;
```

The model can now be used for simulating the bearing, i.e. solving the system of ordinary differential equations given by `B1`Ro`rhs`. For simple cases it is possible to solve the equations in Mathematica, but in general it is necessary to use the code generation facilities of ObjectMath to generate numerical code which can be used for simulations outside Mathematica. This is further discussed in the next section.

Typically one or several special classes in an ObjectMath model are used for collecting methods which perform calculations using the model. One such method might for instance trigger the symbolic simplification of the equations in the bearing model and then use a built-in Mathematica function to simulate rotating the bearing for a certain period of time.

Chapter 8

Modeling complex networks in ObjectMath

Rickard Westman

In this chapter we examine the current ObjectMath mechanisms for modeling structurally complex circuits and their component, as well as discussing conceivable language extensions in this area

Until recently, the applications of ObjectMath have predominantly been systems where a few complex components are connected in a structurally simple way. In many applications, however, the structure of the interconnection network itself is quite complex, whereas the components are relatively simple and standardized.

Here we will discuss various ways of structuring ObjectMath models with respect to object interconnections, especially with an eye towards modularity and re-use. Wherever extensions to the ObjectMath language and environment are considered worthwhile, this will also be discussed.

8.1 Applications

Complex networks of interacting components occur in many types of physical systems, especially when dealing with man-made systems such as electrical circuits, control systems, manufacturing lines, hydraulic systems etc. In such networks, a connection (direct or indirect) exists between all components that can possibly interact in some situation. The absence of such a connection means that interaction is impossible, or produce effects small enough to be negligible.

Such networks (possibly along with the designated component types) can serve as useful abstractions in themselves, and be used as re-usable submodels in yet more complex networks. At the same time, the models of components used in such networks should ideally be usable also in other networks, so it would be beneficial if they were loosely coupled to the model of the network.

8.2 An example model

As a prototypical application, which we will use throughout this report, consider the simple electrical circuit in Figure 8.1. In this network, all components are electrically connected, directly or indirectly. As components, we have a voltage source, two ideal resistors, a capacitor, an inductor and a grounding point.

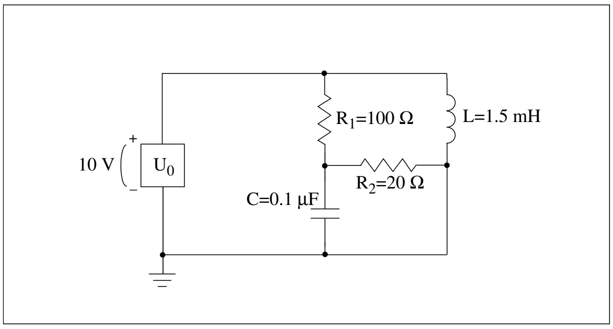


Figure 8.1: *A simple electrical circuit*

It is natural to represent these components as object instances in our ObjectMath model. The mathematical model for them would be represented by reusable classes, containing equations describing the properties of resistors, capacitors etc. To promote re-use, we want to separate the equations which depend on the structure of the surrounding interconnection network from the equations describing the component itself. This separation, and the requisite interface, is done by introducing so-called cuts.

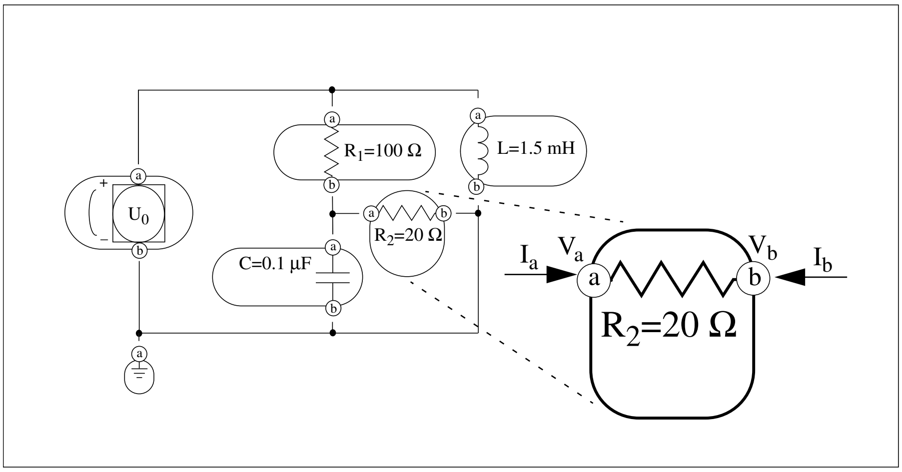


Figure 8.2: *Introducing cuts*

Figure 8.2 illustrates these cuts. The grounding point has one cut point, *a*, whereas all the other components, being two-poles, have cut points *a* and *b*. Each cut point has two associated variables, representing the voltage potential, $V_{a/b}$ and the current going into the cut point, $I_{a/b}$.

8.3 Modeling the components in ObjectMath

In this section we will go through how the components of the example circuit would be represented in ObjectMath to enable us to use the different approaches to modeling of networks covered later. This will also serve as an illustration to the basic object-oriented features of ObjectMath.

Since most of the components are two-poles, it seems sensible to create a class for two-poles. These have the same number of cut points, and consequently the same set of cut variables. Furthermore, the currents going into the two poles are related in the same way, i.e. their respective magnitude is equal, but with opposite signs. In the current version of ObjectMath, this definition could be expressed as follows.

```
class TwoPole
  expose:
```

```
    Ia;    (* Current that goes into terminal a *)
    Ib;    (* Current that goes into terminal a *)
    Va;    (* Potential at terminal a *)
    Vb;    (* Potential at terminal b *)
  reveal:
    Eq[1] :=  Ia == -Ib;
  hide:
    U     :=  Va-Vb;     (* Voltage drop from a to b *)
    I     :=  Ia;        (* Current from a through b *)
end;
```

The `expose` keyword indicates that the cut variables, `Ia`, `Ib`, `Va` and `Vb` can be set from outside of the object, something we will use in some (but not all) of the approaches to network modeling discussed later[1].

The syntax and semantics for the "code bodies" of classes is that of the computer algebra language Mathematica. Here, we define an equation (named `Eq[1]`) relating `Ia` and `Ib`. Since it occurs within a `reveal`-block, it will be visible from outside of the objects of this class (presumably for use in some procedural object collecting equations from instances in the model and applying a solver to them.)

For convenience, we also define two variables `U` and `I` for the voltage and current associated with the component. These can only be used internally within the object, and are replaced with their definition whenever they are used.

Now we can inherit from `TwoPole` in defining the concrete components of our model, for instance the `Capacitor` class:

```
class Capacitor(epsilon, TC1, TC2, area) inherits TwoPole
  (* uses globals DTemp, DTempSq *)
  reveal:
    Eq[2]  :=  C*U''[t] == I;
    C      :=  (epsilon+TC1*DTemp+TC2*DTemp^2)*area;
end;
```

For the ideal capacitor, we define the relationship between voltage, current and capacitance in `Eq[2]`. The capacitance is a function of the temperature of the environment, the area of the capacitor and the capacitivity. The temperature is accessed through a global variable `DTemp` (representing the difference relative to the normalized room temperature, 20 C), while the capacitivity, area, and coefficients for the calculation are parameters of the class. When instances of the class are created, or the class is inherited from by another class, these parameters must be supplied (numerically or symbolically).

The other components are defined similarly (see Appendix A for details), with a small twist for the common terminal described by class instance `Common`:

```
instance Common inherits OnePole
  expose:
    Eq[3] := Va == 0;
end;
```

Since `Common` is not a two-pole, it does not inherit from the two-pole class, but rather from a `One-Pole` class:

```
class OnePole inherits TwoPole
  expose:
    Eq[2] :=  Va == Vb;
```

1. Of course, if an approach is used where cut variables need not be set from outside, the access qualifier should be adjusted appropriately.

```
end;
```

A bit surprisingly, perhaps, the `OnePole` class inherits from the `TwoPole` class. As can be seen in figure 8.3, the one-pole can be seen as a special case of the two-pole, namely the case where the poles are short-circuited. Defining it this way means that we do not have to treat it as a separate case in interconnections – one-poles will have two cut points, too, although they have an extra restriction embodied in `Eq[2]` of `OnePole`.

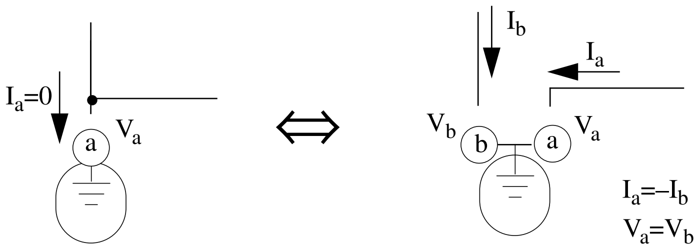


Figure 8.3: *The one-pole as a special case of the two-pole*

8.4 Representing networks in ObjectMath

In this section, three different approaches to representing networks in ObjectMath will be covered. Hierarchical circuit objects, wire objects, and a third approach which may provide more flexibility.

8.4.1 Hierarchy of circuit objects

The idea here is to build circuit classes, which connects "cut-compatible" objects in simple ways. (Objects are "cut-compatible" if they have the same set of cut variables – e.g. if they belong to the `TwoPole` class defined earlier.) By making instances of these circuit classes cut-compatible, they can be used recursively to create arbitrarily complex circuits. Figure 8.4 shows how the example circuit can be represented by two simple circuit classes – the binary serial and parallel circuits.

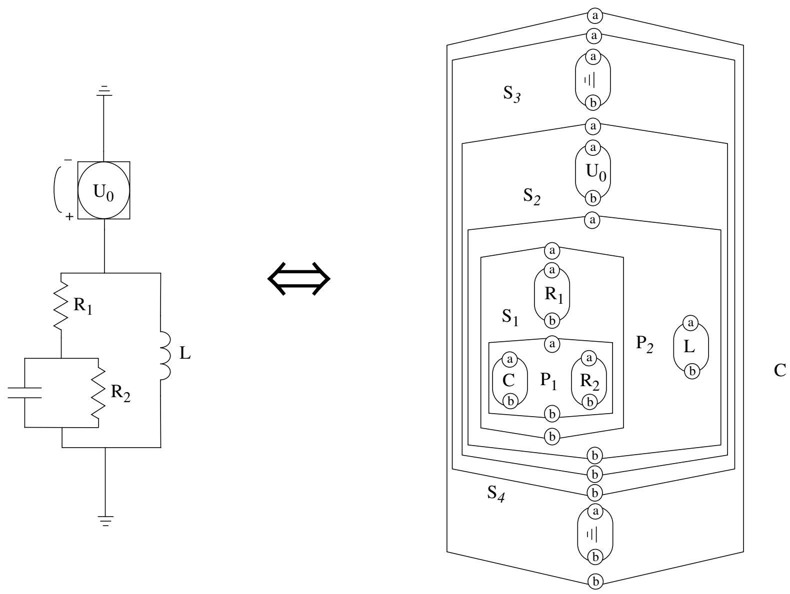


Figure 8.4: *Network using hierarchical circuit objects*

To the left we see a slight redrawing of the original circuit, so that the correspondence between it and the hierarchy of sub-circuits becomes more clear. Note that the number of circuit objects could be reduced from six to four, if the serial circuit class could connect more than two objects at a time.

Implementation of circuit classes

The implementation of these two classes is straightforward. They take two objects as parameters and generate equations for the connection points corresponding to Kirchoff's first law, and the equality of electrical potential in connection points, respectively:

```
class ParCircuit(component1, component2) inherits TwoPole
  expose:
    (* connection from A to component1/2 *)
    Eq[2] :=  component1`Va == Va;
    Eq[3] :=  component2`Vb == Va;
    Eq[4] :=  component1`Ia+component2`Ia == Ia;

    (* connection from component1/2 to B *)
    Eq[5] :=  component1`Vb == Vb;
    Eq[6] :=  component2`Vb == Vb;
    Eq[7] :=  component1`Ib+component2`Ib == Ib;
end;

class SerCircuit(component1, component2) inherits TwoPole   expose:
    (* connection from A to component1 *)
    Eq[2] :=  component1`Va == Va;
    Eq[3] :=  component1`Ia == -Ia;

    (* connection between components *)
    Eq[4] :=  component2`Va == component1`Vb;
    Eq[5] :=  component1`Ib == -component2`Ia;

    (* connection from component2 to B *)
    Eq[6] :=  component2`Vb == Vb;
    Eq[7] :=  component2`Ib == -Ib;
end;
```

Language extensions that could be useful

If ObjectMath was a strongly typed language, `component1` and `component2` would be declared as `TwoPole`'s, so that cut-compatibility could be checked at compile time. Even though the addition of strong type checking to the entire language would be a major undertaking, doing it on the object level would be quite simple and worth considering.

As mentioned above, it would be useful with versions of these circuit classes which took more than two sub-components. Of course it would be straightforward to implement special versions which took three, four, five or more parameters, but a general mechanism for having a variable number of class parameters would be better.

Other kinds of circuit classes

Most electrical circuits consisting of two-pole components can be represented by composition of serial and parallel circuits, but some need to be transformed into an equivalent (but topologically different) circuit before this is possible. Such transformations are not desirable from a high-level modeling point of view. Thus, introducing additional circuit types such as delta- and Y-circuits may be useful. It is also necessary to handle components with more poles than two. One way of doing that is by using wire objects (described in the next section), which explicitly connects two cut points with each other.

Drawbacks

Although the concept of circuit objects is appealing in some ways, it has drawbacks. Using this for describing networks will lead to many small objects which cannot easily be manipulated in the current environment. Small changes in the network topology (such as removing a connection) may necessitate major restructuring of the circuit objects. Also, the translation of an existing circuit diagram into nested circuit objects can be non-trivial and error prone if tool support is lacking.

8.4.2 Wire objects

Wire objects are similar to the circuit objects described earlier, in that they contain simple equations that connect cut points to each other. Unlike circuit objects, they are not themselves components. Their class parameters are not components, but cut points of components which should be connected. Their use results in a flat object structure which is easy to construct from an existing circuit diagram. Every conceivable circuit can be built using only one class of wire objects (provided that variable number of class parameters is introduced in ObjectMath – otherwise there will be one class for each supported connection degree).

8.4.3 Representing and generating connections

Hitherto, connections between objects have mainly been represented in ObjectMath by class parameters given at instantiation time, indicating which other objects the object interacts with. For instance, when instantiating the ring instance of a roller bearing, one of the parameters would be the name of the rollers which could have mechanical contact with the ring. The ring class would have equations which referred to certain variables inside these rollers, relying on knowledge of the names of these variables. Although inflexible, this scheme works quite well for rolling bearing applications, since the structure of the modeled system is rather static. Thus, it does not really matter if assumptions about the structure of the system are embedded into the individual objects. Also, the ObjectMath feature for handling sets of instances makes it possible to leave the number of rollers in a bearing unspecified in the model. The restriction that set members must be of the same type is not a problem in rolling bearing applications.

For electrical circuits, however, this scheme would be unsuitable. A component could be used in an arbitrarily complex network of components of unknown types, so a more flexible approach is required.

A flexible manual approach

One way to allow for arbitrarily complicated connections is to add the connecting equations in conjunction with instantiating the object. In ObjectMath, it is not necessary to create a class for this purpose, since instances can have their own unique code bodies.

While this approach is workable, it is a bit awkward. Consider the connection equations that would be necessary in the definition of R_2 in our example circuit in Figure 8.1. What exactly would these equations specify? We know that the voltage potential of connected cuts should be equal, so we could have equations like:

```
Eq[2] :=  R2.Va == R1.Vb;
Eq[3] :=  R2.Va == C.Va;
Eq[4] :=  R2.Vb == L.Vb;
Eq[5] :=  R2.Vb == C.Vb
Eq[6] :=  R2.Vb == Ground.Va
Eq[7] :=  R2.Vb == U0.Vb
```

and similar equations using Kirchoff's first law for specifying the currents.

However, it is not obvious that `Eq[2]` should reside in the R_2 object – after all it is just as natural to put it in R_1. This could be put in both of the connected components, but would be bulky, be more work to maintain, and increase the risk for inconsistencies. If a single equation per connection is desired, some convention could be used for selecting in which object to put the equation, e.g. the first in lexical order, but that is inelegant and would be a hassle when renaming objects.

Generating connection equations

One way to make this approach more attractive is by constructing a more appealing interface for generating connection equations. This could be done by letting the ObjectMath system analyze the model and generate the needed connection equations from obtained information. For instance, one could define an application-specific `Connect` function which knew how to generate these connection equations provided that neighbor lists specifying the connections associated with cut point `a` and `b` was stored in each instance. Instead of specifying nine equations by hand in R_2, something like this would suffice:

```
instance R2 inherits Resistor(20, 0, 0, 1)
begin
  $a$ := {R1.Vb, C.Va};
  $b$ := {L.Vb, C.Vb, Ground.Va, U0.Vb};
end
```

Here, a naming convention is used to help `Connect` find the neighbor lists for cut points `a` and `b`, respectively. To simplify writing the `Connect` function, utility functions for iterating over instances, collecting neighbor lists, constructing equations etc. could be provided in a standard package.

Chapter 9

ObjectMath^{c}, the compilable subset

The ObjectMath environment provides facilities to translate a subset of the ObjectMath and Mathematica languages to compiled programs in C++ or Fortran. This subset includes all functions and operators that compute numeric values, but excludes symbolic and computer algebra related functions that return symbolic expressions. However, it is possible to evaluate a symbolic expression (which may contain operations such as simplification, symbolic differentiation, substitution, etc.) and generate executable numeric code from the symbolic expression resulting from this evaluation, provided that the resulting expression(s) only contains operators and functions in the compilable subset described in this chapter. The compilable subset of ObjectMath is called ObjectMath^{c}.

The arithmetic model used in ObjectMath^{c} is specified by the IEEE Standard for Binary Floating Point Arithmetic, IEEE Standard 754. Operations on complex numbers are currently not supported.

9.1 Predefined functions and operators

Expressions listed in this section are predefined by the compiler and will be translated correctly from ObjectMath into the target language (Fortran90 or C++) without any additional type declarations.

The operators marked *ce* belong to the compilable expression subset, i.e. all value-returning operators and predefined or user-defined functions without side effects (functions that do not change global variables or perform input/output).

The reason to impose the condition of side-effect free functions in ObjectMath^{ce} is that expressions can be re-ordered and common subexpressions removed in the generated code, in order to make execution more efficient. Another order in assigning and referencing global variables or performing input/output usually results in different, often unintended, program behavior. However, some restricted cases of side-effects can be re-ordered without changing the meaning of the program. One such case is when the elements of an array are assigned once, and independently of each other, and not used in the same expression. Such restricted side-effects are allowed for functions in ObjectMath^{ce}. ObjectMath does not check the condition of side-effect freeness; this is the user's responsibility.

All operators and functions in *ce* also belong to *c*, the compilable subset, which also contains control expressions (If, While, etc.), assignment statements and functions with side effects. All real and integer constants naturally belong to *ce*.

Operators marked *c* below belong to the compilable subset, but not to the compilable expression subset. Operators marked *ce* belong to the compilable expression subset. Some operators and functions can be applied to arrays (arr) or return arrays as values. This works in the same way as in Mathematica.

The current version of the compilable subset is oriented toward operations on real numbers and integers, and arrays containing such numbers. The basic mathematical functions usually found in C/C++ or Fortran are provided. In Mathematica there are also a number of special mathematical functions such as BesselJ[], Gamma[], etc. If the user has access to an implementation of such a function in C/C++ or Fortran, or a linkable object code library containing this function, it can be declared as an external function and thus automatically included in the compilable subset.

ObjectMath^c could be extended with string operations, input/output and certain mapping operations provided in Mathematica. Since computation and mathematical modeling is the primary focus of ObjectMath, this has not yet been done.

9.1.1 Named Constants

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
	True	-	bool	ce
	False	-	bool	ce
	E	-	real	ce
	Pi	-	real	ce

9.1.2 Control expressions

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
	For[*start,test,incr,body*]	-	-	c
	While[*test,body*]	-	-	c
	If[*cond,true-expr,false-expr*]	-	-	c or ce
	Which[*tests,values*]	-	-	c
	Break[]	-	-	c

9.1.3 Structure expressions

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
e_1; e_2; ..	CompoundExpression[*exprs*]	-	real,int,bool	ce
	Module[*vars,inits,body*]	-	-	c
	Block[*vars,inits,body*]	-	-	c
	With[*vars,inits,body*]	-	-	c

9.1.4 Arithmetic Expressions

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
==	Equal[*exprs*]	real, int	bool	ce
!=	Unequal[*exprs*]	real, int	bool	ce
>	Greater[*exprs*]	real, int	bool	ce
<	Less[*exprs*]	real, int	bool	ce
>=	GreaterEqual[*exprs*]	real, int	bool	ce
<=	LessEqual[*exprs*]	real, int	bool	ce
!	Not[*e*]	bool	bool	ce
||	Or[*exprs*]	bool	bool	ce
&&	And[*exprs*]	bool	bool	ce
+	Plus[*exprs*]	real, int, arr	real, int, arr	ce
-	Subtract[e_1,e_2]	real, int, arr	real, int, arr	ce

-	Minus[*e*]	real, int, arr	real, int, arr	ce
*	Times[*exprs*]	real, int, arr	real, int, arr	ce
/	Divide[e_1,e_2]	real, int	real	ce
	Rational[e_1,e_2]	real, int	real	ce
^	Power[e_1,e_2]	real, int, arr	real, int, arr	ce
	Abs[*e*]	real, int, arr	real, int, arr	ce
	If[*cond,true-expr,false-expr*]	-	real, int, arr	ce
	Sign[*e*]	real, int	int	ce
	Floor[*e*]	real	int	ce
	Ceiling[*e*]	real	int	ce
	Round[*e*]	real	int	ce
	NaturalDivide[e_1,e_2]	int	int	ce
	Sqrt[*e*]	real, int	real	ce
	Exp[e_1,e_2]	real, int	real	ce
	Log[e_1,e_2]	real, int	real	ce
	Sin[*e*]	real, int	real	ce
	Cos[*e*]	real, int	real	ce
	Tan[*e*]	real, int	real	ce
	Cot[*e*]	real, int	real	ce
	Sec[*e*]	real, int	real	ce
	Csc[*e*]	real, int	real	ce
	ArcSin[*e*]	real, int	real	ce
	ArcCos[*e*]	real, int	real	ce
	ArcTan[*e*]	real, int	real	ce
	ArcTan2[e_1,e_2]	real, int	real	ce
	Sinh[*e*]	real, int	real	ce
	Cosh[*e*]	real, int	real	ce
	Coth[*e*]	real, int	real	ce
	Sech[*e*]	real, int	real	ce
	Csch[*e*]	real, int	real	ce
	Tanh[*e*]	real, int	real	ce
	ArcSinh[*e*]	real, int	real	ce
	ArcCosh[*e*]	real, int	real	ce
	ArcTanh[*e*]	real, int	real	ce
	ArcCoth[*e*]	real, int	real	ce
	Cross[e_1,e_2]	arr	arr	ce
	Transpose[*e*]	arr	arr	ce
$e_1 . e_2 . ...$	Dot[e_1,e_2,...]	arr	real, int	ce
	IReduce[*func,arrexpr,itr*]	-	real, int, arr	ce]

The functions which return an integer value converted from real, Sign, Floor Ceiling, Round, give an undefined value or an exception (depending on the underlying target language, C++ or Fortran90) when trying to fit a too big number into a (32-bit) integer.

`NaturalDivide` is a function which performs integer division, taking two natural numbers as arguments. If any of the arguments are negative, the behavior is undefined. In Mathematica, a possible implementation would be `Floor[Divide[e,e]]`. In the target programming language, it should be possible to use integer division, regardless of how it is defined for negative arguments. NaturalDivide only works for positive integers, and will give error messages for negative arguments in Mathematica (whereas the corresponding original Mathematical counterparts are defined for negative arguments and will not give an error message).

The `Rational` function is part of the compilable subset. It is treated exactly like `Divide`, and converted to `Divide` during code generation.

The special purpose `Cross` function computes the cross product of two vectors for the special case of vectors of length 3, producing a result vector of length 3. It is substantially more efficient than the Mathematica function `CrossProduct` for this case.

The `IReduce` construction forms an expression by "reducing" an array (i.e. list in Mathematica) of expressions without using any side-effect causing constructs such as `For` or `While`. The first argument is a function or operator which is associative and commutative, e.g. `Plus` or `Times`, and which when applied repeatedly will reduce the whole array or array expression to a single number. The second argument is an indexed array element (or expression). The third argument is an iterator. For example:

```
IReduce[Plus, a[i,j], {{i,1,5},{j,1,5}}]
```

will calculate the sum of every element in the array a[5,5]. Another example, the ObjectMath expression:

```
IReduce[Plus, Times[ a[[i,j}}, b[[j,k]] ], {j,1,5}]
```

can be automatically translated to the following Fortran90 iterative loop:

```
om_T1 = a(i,1) * b(1,k)
do j=2,5
  om_T1 = om_T1 + a(i,j) * b(j,k)
end do
```

9.1.5 Assignment

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
`var:= e`	SetDelayed[*var,e*]	arr,real,int,bool	-	c
`var :== e`	SetDelayedDelayed[*var,e*]	arr,real,int,bool	-	c
var = expr	Set[*var,expr*]	arr, real, int	real, int, arr	c
{vars}=funcall	Set[List[*vars*],*funcall*]	-	arr	c

The supported main assignment function, SetDelayed, does not have a return type itself, but requires its arguments (left-hand and right-hand side of the assignment) to be of compatible types. They are compatible if they can be made the same type by performing standard type promotion (e.g. promoting integer to real, or a scalar or lower-dimensional array to a higher-dimensional array), provided that this promotion does not change the type of the left-hand side. If it does, then the assignment is illegal. This means that an expression of a real type cannot be assigned to a variable of integer type without using explicit conversion of the right-hand side (e.g. using `Floor[]`).

In the last case, *funcall* must be a call to a function returning a list of the same length as the list in the left hand side of the assignment. Also, the *vars* list in the left hand side must contain only variables.

9.1.6 Other expressions

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
$\{e_1, e_2, ...\}$	List[(*exprs*]	arr,real,int,bool	arr	ce
expr[[i]]	Part[*expr,i*]	arr, int	arr,real,int,bool	ce

(** What is the semantics of List in C++ or Fortran90? To combine lower-dimensional arrays or scalars into a higher-dimensional array?)

9.1.7 User defined and external functions

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
	funcname[*exprs*]	real,int,arr,bool	real,int,arr,bool	ce or c

All user-defined functions which have been type declared according the ObjectMath typing rules belong to the compilable subset. The same is true for functions declared as external, and exist in a library (or a compiled file) that can be linked together with the generated code in C++ or Fortran90.

Functions with multiple return arguments can be compiled if they are type declared. Such a function can only be used in the right hand side of an assignment, and the left hand side has to be a list of variables. Thus, a call that returns multiple values can for example look like this:

```
{a, b, c}  =  F[x+y, 3.4];
```

9.1.8 Operators that do not occur in standard Mathematica

Spec syntax	Operator	Arg type(s)	Result type(s)	Subset
:==	SetDelayedDelayed[*var,e*]	arr,real,int,bool	-	c
	NaturalDivide[e_1,e_2]	int	int	ce
	Cross[e_1,e_2]	arr	arr	ce

The above operators occur in ObjectMath but not in standard Mathematica. They are listed here once more for clarity.

Chapter 10

Type declarations

ObjectMath provides language extensions for declaring the types of Mathematica variables, arrays and function, and augments the facilities for type annotation of function parameters already present in Mathematica. This static typing scheme is suitable for ObjectMath^c, the compilable subset of ObjectMath.

The main reason to introduce static typing in ObjectMath is to be able to generate efficient code in C++ and Fortran90. This is not always possible without static type information, since there are a number of cases where the interpretation of parts of ObjectMath/Mathematica models are ambiguous when used as input for code generation. A related reason is to provide a consistent handling of arrays between Mathematica, C++ and Fortran90 (and perhaps additional languages in the future).

The type declarations are mainly treated as annotations within Mathematica, i.e. they usually have no effect on execution or algebraic transformations within Mathematica. The exception is the *array* declaration which actually creates an array with symbolic components in Mathematica, represented as a (nested) list structure according to the usual conventions for Mathematica arrays. The type information is however necessary for generating executable code in Fortran90 or C++.

10.1 Scalar variables

The scalar types that can be declared for variables within an ObjectMath model are integer (32 bit), IEEE double precision real numbers or boolean values. Boolean values have values True or False. Examples are:

```
int i;
real r;
bool b;
```

10.2 Arrays

Arrays in ObjectMath^c are homogenous, ordered collections of elements, all belonging to a common base type (real, int or bool). Arrays can be declared as 1-dimensional, 2-dimensional or multi-dimensional. If the ObjectMath model is going to be translated to Fortran90, there is a limitation of at most 7 dimensions. For translation to C++ there is currently a limitation of 6 dimensions. Arrays can be passed as arguments to ObjectMath functions, and be returned as function values. Only arrays with constant dimensions or symbolic dimensions that evaluate to constant values at declaration time can be declared. The reason is that an array with the specified dimensions containing symbolic components is actually created at declaration evaluation time in Mathematica. This makes is possible to access array components symbolically and to perform symbolic transformations involving array elements.

```
real array[3]          (* a one-dimensional array of 3 real numbers*)
real array[5,4,10]     (* a three-dimensional real array *)
int  array[4,4]        (* a 4x4 array of integers *)
bool array[3,3]        (* a 3x3 array of booleans *)
```

10.3 Functions

Functions can be declared with argument types and a return type. First we show examples for scalar types:

A function with one integer parameter returning a real:

```
real Square[int x_] := x*x;
```

A function with one real parameter, returning two reals is shown below. (When translating to C++ or Fortran90, multiple return values are handled by adding additional return parameters).

```
{real,real} SinCos[real x_] := { Sin[x], Cos[x] };
```

10.3.1 Functions returning arrays

Arrays can be passed as parameters to functions:

```
real AddThree[real array[3] a_] := a[0]+[1]+a[2];
```

and arrays can be returned as function values:

```
real array[3] OneTwoTree[] := { 1, 2, 3 };
```

10.3.2 Functions with no return value

This is an imperative function, i.e. a function (really a procedure) that does not return anything (the empty tuple) but has a side effect on a declared integer variable ii:

```
{} Impfunc[int foo_] := { ii = foo };
```

Chapter 11

Code generation

The ObjectMath environment provides basically two kinds of code generation facilities:

- Generation of Mathematica code from ObjectMath models
- Final code generation to generate Fortran90 or C++ code

There is also an experimental facility for generating parallel code, described in Chapter 12.

The generation of Mathematica code is needed for modeling and prototyping, and to integrate the ObjectMath frontend with the Mathematica system to provide computer algebra transformations. This is not really what is usually meant when speaking about code generation, and in the following we focus exclusively on final code generation.

Considering final code generation, there are two cases: System level code generation accepts a list of equations from an ObjectMath model (compare with the simple example in Chapter 4), and generates code to solve this system of equations using a specified numeric solver. Function level code generation allows the user to guide the code generation process in more detail, by specifying which variables, functions and expressions should be translated to Fortran90 or C++.

11.1 An example of generated code

The function level interface of the ObjectMath code generator provides routines for declaration and for code generation. By calling the declaration routines the user supplies the code generator with information about the types of variables and functions. Then the generation routines can be called to generate code for functions.

The code generator takes advantage of the fact that pure functions like *sin*, *cos*, and *tan* are devoid of side-effects in order to eliminate common subexpressions that the C++ compiler cannot optimize. Note that it is necessary to eliminate all common subexpressions (even if the compiler can handle the ones involving only arithmetic operators) so that we do not miss any opportunities for further optimizations. Temporary variables which hold the results of subexpressions are introduced. Thus the code generator must derive the type of each subexpression. Even without the common subexpression elimination this would be necessary since the expressions otherwise may become so large that the compiler cannot handle them. (Many compilers seem to have a built-in hard limit on the size of expressions.)

As an example, consider the following ObjectMath expression, a vector of length three. The variable `` B`t `` is a scalar (denoting time) while both `` B`Ro`R `` and `` B`Ro`Fi `` are functions which return a scalar.

```
{ Cos[B`Ro`Fi[B`t]] * B`Ro`R[B`t],
  B`Ro`R[B`t] * Sin[B`Ro`Fi[B`t]],
  0 }
```

The code generator generates the following C++ code for computing the expression above:

```
// Declarations
double Tmp_721;
double Tmp_722;
double Tmp_723;
double Tmp_724;
double Tmp_725;
double Tmp_726;
doubleVec3 Tmp_727;
// Compute expression
Tmp_721 = B_Ro_Fi(B_t);
Tmp_722 = cos(Tmp_721);
Tmp_723 = B_Ro_R(B_t);
Tmp_724 = Tmp_722*Tmp_723;
Tmp_725 = sin(Tmp_721);
Tmp_726 = Tmp_723*Tmp_725;
SetArray3(Tmp_727,Tmp_724,Tmp_726,0);
// Return result
return(Tmp_727);
```

In this small example the only common subexpressions are the calls to the functions `B`Ro`R` and `B`Ro`Fi`. The results of these calls are stored in the temporary variables `Tmp_721` and `Tmp_723` respectively. For complicated expressions in realistic applications, the common subexpression elimination often reduces the size of the generated code by a magnitude or more.

Different numerical programs can be generated from the same ObjectMath model depending on how much information we supply before generating and compiling the code. If numerical values are assigned to ObjectMath variables before code generation, symbolic simplification usually results in a more efficient, but less general, program. This can be viewed as a form of *partial evaluation*.

11.2 System level code generation

This function, which currently exist only in a preliminary version, accepts a list of equations from an ObjectMath model, a specification of the numeric solver to be used, and generates code to solve this system of equations. (** Update this when a full facility has been implemented)

Currently, only solution of ordinary differential equations (ODEs) in explicit form is supported, but work is ongoing on extensions to cover partial differential equations.

A preliminary version also supports generation of message-passing parallel code for numeric solution of the system of equations (ODEs), see Chapter 12.

11.3 Function level code generation

The function level code generation provides facilities for generating code for ObjectMath functions and variables. Functions can either be more or less directly translated into the target programming language (e.g. C++, Fortran90), or the function body can be symbolically evaluated before generating code.

Functions and variables for which code should be generated must be type declared as described in Chapter 10.

The code generation facilities are invoked by calling a number of ObjectMath functions, defined in the Mathematica package `CodeGeneration`. These routines are actually just stubs which calls corresponding routines in the code generator via MathLink as depicted in Figure 1.2. Below we describe the available code generation functions.

OpenFile["*file*", "*language*"]

This function must be called before calling any of the functions which actually emits code. Emitted code will be placed on a file named "file" with a suffix appended. The suffix depends on the target programming language. If the target language is Fortran90, the extension will be `.f90`. If the target language is C++ two files will be used. One with the suffix `.h` for declarations and one with suffix `.C` for definitions. The first file will be included in the latter.

The second argument selects the desired target language. Legal values are currently "Fortran" for Fortran90 and "C++" for C++. If an output file already is open (by a previous call to OpenFile), it will be closed before opening the new file. An example:

```
OpenFile["foo", "Fortran"];
```

CloseFile

A call to this function closes the currently open output file. An example call is:

```
CloseFile[];
```

GenerateVariable[*name*]

This function generates declarations of one or more variables on the currently open output file. There are several different cases depending on the argument:

- If *name* is an ObjectMath variable, a declaration of this type annotated variable is generated.
- If *name* is an ObjectMath *instance*, declarations of all type annotated variables in this instance are generated.
- If the argument is omitted, declarations of all type annotated variables in the ObjectMath model are generated.

Examples, given the integer variable `i`, real variable `r3` and boolean variable `b2` are declared within the instance `Funs` belonging to the currently loaded ObjectMath model `cgen`:

```
GenerateVariable[Funs`i];     (* Declare int i *)
GenerateVariable[Funs`r3];    (* Declare real r3 *)
GenerateVariable[Funs`b2];    (* Declare bool b2 *)
GenerateVariable[Funs];       (* Declare all variables in Funs *)
GenerateVariable[];           (* Declare all variables in model cgen *)
```

GenerateFunction[*name*]

This function generates a function in the target programming language which corresponds to the ObjectMath function *name*. The body of the function must only contain ObjectMath or Mathematica constructs that belong to the compilable language subset ObjectMath^c, see Chapter 9. The function must be type declared. Some examples follow below, assuming that the functions `Square` and `SinCos` (declaration examples in Chapter 10) have been declared within the instance `cgen`:

```
GenerateFunction[Funs`Square];
GenerateFunction[Funs`SinCos];
```

GenerateEvaluateFunction[*name*]

This function evaluates the body of the ObjectMath function *name* symbolically in Mathematica. Such a symbolic evaluation often gives a (huge) expression as a results. Then, a function definition in the target programming language is generated, that will compute and return the value of this expression. The code generator checks that the expression resulting from the symbolic evaluation is limited to the expression subset of ObjectMath^c, the compilable ObjectMath subset. It also checks that all referenced variables are type declared as well as the function *name* itself.

An example follows, assuming the declaration of functions `Silly` and `Large`, which are also shown. During code generation, common subexpressions in the symbolic expression will automatically be factored out, to produce more efficient code. Huge symbolic expressions often contain a number of (sometimes quite large) common subexpressions. The function `Silly` below contains the common subexpression `(x+y)`. The function `Large` must be compiled with `GenerateEvaluateFunction`, since it contains primitives outside the compilable subset, which however are expanded and substituted during symbolic evaluation.

```
int Silly[real x_, real y_] := (x+y)^(x+y);

real Large[real y_] := Normal[N[Series[Sin[x],{x,0,300}],3] /. x->y];

GenerateEvaluateFunction[Funs`Silly];
GenerateEvaluateFunction[Funs`Large];
```

11.4 A complete code generation example

The calls to code generation functions are usually embedded within a special instance within the ObjectMath model, and evaluated in a certain order. This is the case for the instance `Code` in the example below.

```
instance Code
  expose;

  gen := ( Openfile["foo", "Fortran"];
           GenerateVariable[Funs`i];
           GenerateVariable[Funs`r3];
           GenerateVariable[Funs`b2];
           GenerateFunction[Funs`Square];
           GenerateFunction[Funs`SinCos];
           GenerateEvaluateFunction[Funs`Silly];
           GenerateEvaluateFunction[Funs`Large];
           CloseFile[];  );
end;
```

11.5 Code generation for the 2D bearing example

(** to be inserted)

Chapter 12

Parallel code generation

Niclas Andersson and Peter Fritzson

The parallel code generation facilities provided by the ObjectMath environment are still in a preliminary stage. Therefore the description in this chapter is incomplete and sketchy, and no precise instructions on how to invoke the parallel code generation facilities will be given.

Two variants of parallel code generation are provided:

- Generation of parallel code by automatic partitioning and parallelizing huge expressions derived from systems of equations in an ObjectMath model.
- Generation of parallel code by explicitly expressing parallelism in an ObjectMath through a `ForAll` construct.

Some results from the first approach are described in [And95] and [And96]. In the rest of this chapter we will exclusively discuss the second approach.

12.1 Motivation for explicit parallelism

There are situations where the user explicitly wants to instruct the runtime system where and when the computation is to be performed in parallel. If the user expresses the problem using arbitrary imperative constructs in ObjectMath, the compiler has no means to parallelize the problem. Since the automatic expression parallelization approach only deals with *pure* expressions, the user must encapsulate used imperative constructs in side-effect free, pure functions.

In the rest this chapter we leave the functional approach and instead describe an explicit interface to the runtime system which enables the user to gain more control of how the parallel computation should proceed. In this chapter, generated code examples are shown in Fortran90 and HPF.

12.2 Language Extensions

To explicitly express parallelism, the ObjectMath language is extended with a `ForAll` construct:

`ForAll[`*expr, iterator(s)*`]`

The expression *expr* must be an assignment whereas *iterator(s)* represents a single iterator or a list of iterators. A simple example of an ObjectMath `ForAll` follows below:

```
ForAll[ y[[i]] := rhs[x[[i]], t], {i, 1, 20}]
```

Here, 20 instances of the `rhs` function are invoked in parallel, each with its own element of the vector `x`. The return values are assigned to the vector `y`. The `rhs` function contains the right-hand sides of a system of first-order ordinary differential equations (ODEs) to be solved.

To simplify the code generation, there must be no dependencies among the different instances of the right-hand side of the assignment. The `ForAll` does not affect the data distribution used in computations on a parallel computer. Currently, this decision is left to the runtime system. In the ODE case, the data is preferably kept centralized, while it may be distributed in other cases.

12.3 Compiling ObjectMath ForAll

There are three alternatives when generating code from the ObjectMath `Forall` construct:

- Serial Fortran90 code for one processor.
- Parallel Fortran90 code which interface to a message passing runtime system.
- Data parallel HPF code.

12.3.1 Serial Fortran90 code

In this case, code is generated that iterates over the iterator variables in a number of for-loops that uses temporary variables. Afterwards, the real variables are assigned in a new set of for-loops. A generated serial code fragment for the previous ObjectMath `Forall` example appears as follows:

```
do om_i=1,20
  om_T1(om_i) = rhs(x(om_i), t)
end do
do om_i=1,20
  y(om_i) = om_T1(om_i)
end do
```

12.3.2 Parallel Fortran90 code

In this case, a runtime system is supplied that takes care of the message passing and also schedules the tasks, which here are the instances of the `ForAll` loop. The following code is generated from the Mathematica `ForAll` expression above. The generated code is produced in four sections: types for communicated data, extra variables, the parallel loop, and the right-hand side expression(s) `rhs`:

Types for communicated data:

```
common /<model>/ om_P1_id
integer om_P1_id

type om_P1_in
  sequence
  integer(om_int) p1
  real(om_real) p2
  real(om_real) p3
end type om_P1_in

type om_P1_out
  sequence
  real(om_real) p1
```

```
end type om_P1_out
```

Extra variables:

```
external om_parallel
type(om_P1_in) om_T1(n)
type(om_P1_out) om_T2(n)
```

The parallel loop:

```
do i=1,20
  om_T1%p1 = i
  om_T1%p2 = x(i)
  om_T1%p3 = t
end do

call om_parallel(om_P1_id, 20, om_T2, om_T1)
do i=1,20
  y = om_T2%p1
end do
```

And finally, the right-hand side expression(s):

```
subroutine om_P1(om_res, om_arg)
  use om_<model>
  implicit none
  type(om_P1_in) om_arg
  type(om_P1_out) om_res

  !   Unpack args
  i = om_arg%p1
  x(i) = om_arg%p2
  t = om_arg%p3

  ! Evaluate the body
  y(i) = rhs(x(i), t)

  !   Pack result
  om_res%p1 = y
end subroutine om_P1
```

The communicated variables are unpacked to the same variables in the receiver as they were packed from in the sender. That is, the environment for the expression will be the same irrespective of the used processor. Additionally, there must be a subroutine that initializes the runtime system with information about message sizes and available functions. In the example, the expression that was evaluated in parallel was a single function. The scheme could easily be expanded to include arbitrary expressions by encapsulating these in an automatically generated function.

12.3.3 Generating High Performance Fortran (HPF) code

In High Performance Fortran the runtime system is already supplied and there are data parallel constructs in the language [HPFF93]. Nevertheless, there are several constraints in HPF which make these difficult to use when generating code from ObjectMath.

An example of such a constraint is that all procedures that are invoked inside an HPF `ForAll` statement have to be *pure*. Inside such a procedure global variables must not be changed since this violates the *pureness* of the procedure. In the ObjectMath compiler, variables that are not local to a function, but may be local within an object, are transformed into global variables. Therefore, a model that is modular in the object oriented sense may not be possible to parallelize using HPF.

Moreover, several HPF compilers today only implement a subset of HPF and/or do not utilize the parallelism that exists in the source code. For example, the `!HPF$` independent directive is often ignored.

There are two main possibilities to generate parallel HPF code:

- Use the HPF `ForAll` statement.
- Use the `!HPF$` independent directive.

When using the `ForAll` alternative the code will look as follows:

```
forall (om_i=1:20)
  y(om_i) = rhs(y(om_i), a(om_i), a)
end forall
```

The function `rhs` (and all functions and subroutines that `rhs` invokes) must be declared pure.

Note, however [Koe94], that a pure subroutine strangely enough is not permitted at the top level within an HPF ForAll construct. However, a pure function is allowed that may invoke a pure subroutine.

When using `!HPF$` independent alternative, the generated code is identical to serial code except for the independent declaration:

```
! HPF$ independent
  do om_i=1,20
    om_T1(om_i) = rhs(x(om_i), t)
  end do
  do om_i=1,20
    y(om_i) = om_T1(om_i)
  end do
```

This version eases the constraints on the body of the loop. To conform to HPF, the independent directive must be true since different instances of the loop must not interfere.

Appendix A

Model of an electrical circuit

This is the ObjectMath model for the electrical circuit in Figure 8.1, using hierarchical circuit objects. Code for solving/simulation is not included.

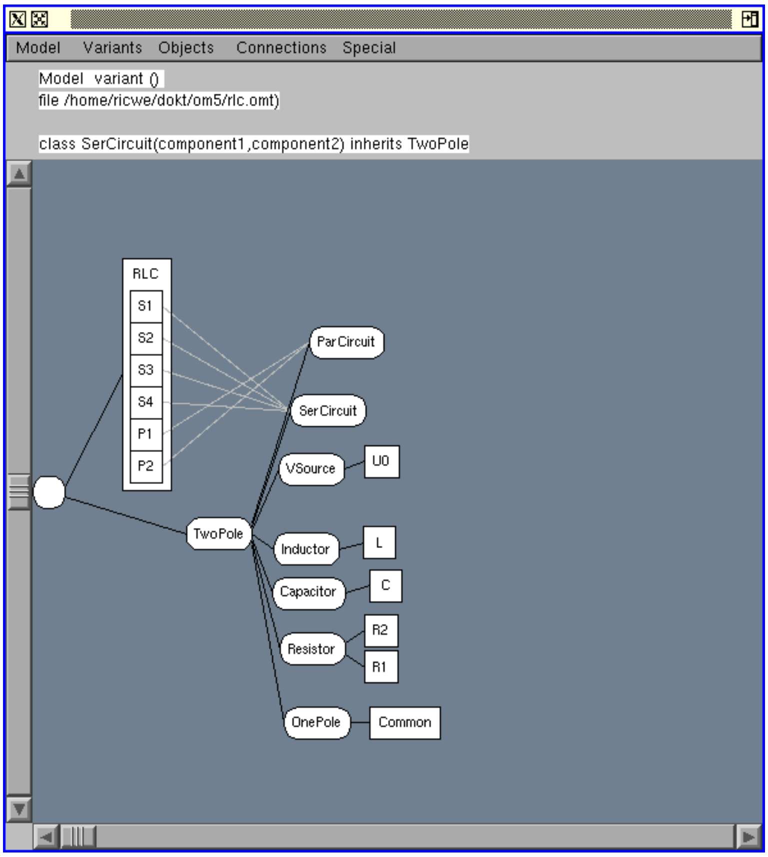


Figure A.1: *The ObjectMath class browser showing the rlc model structure.*

```
model rlc;
  t;  (* independent variable *)

class TwoPole (* Coordinates: 76, 57 *)
    expose:
      Ia;    (* Current that goes into terminal a *)
      Ib;    (* Current that goes into terminal a *)
      Va;    (* Potential at terminal a *)
      Vb;    (* Potential at terminal b *)
    reveal:
      Eq[1] :=  Ia == -Ib;
    hide:
```

```
      U     :=  Vb-Va;     (* Voltage across the two-pole *)
      I     :=  Ia;        (* Current from a through b *)
end;

class Resistor(R,TR1,TR2,area) inherits TwoPole (* Coordinates: 6, -5 *)
  expose:
    (* uses globals DTemp *)
    Eq[2]  :=  Rval*Ia == Va-Vb;
    Rval   := (R+TR1*DTemp+TR2*DTemp^2)/area;
end;

class Capacitor(epsilon,TC1,TC2, area) inherits TwoPole (* Coordinates: 0, -4 *)
  expose:
    (* uses global DTemp *)
    Eq[2]  :=  Cval*U''[t] == I;
    C      :=  (epsilon+TC1*DTemp+TC2*DTemp^2)*area;
end;

class Inductor(L,TL1,TL2) inherits TwoPole (* Coordinates: 1, 2 *)
  expose:
    (* uses global DTemp *)
    Eq[2]  :=  Lval*I''[t]   == U;
    Lval   :=  L+TL1*DTemp+TL2*DTemp^2;
end;

class OnePole inherits TwoPole (* Coordinates: 9, -155 *)
  expose:
    Eq[2] :=  Va == Vb;
end;

instance Common inherits OnePole
  expose:
    Eq[3] := Va == 0;
end;

class VSource(V) inherits TwoPole (* Coordinates: 5, 8 *)
  expose:
    Eq[2] :=  U == -V;

end;

class ParCircuit(component1, component2) inherits TwoPole (* Coordinates: 28, 76
*)
  expose:
    (* connection from A to component1/2 *)
    Eq[2] :=  component1`Va == Va;
    Eq[3] :=  component2`Vb == Va;
    Eq[4] :=  component1`Ia+component2`Ia == Ia;

    (* connection from component1/2 to B *)
    Eq[5] :=  component1`Vb == Vb;
    Eq[6] :=  component2`Vb == Vb;
    Eq[7] :=  component1`Ib+component2`Ib == Ib;
end;

class SerCircuit(component1, component2) inherits TwoPole (* Coordinates: 14, -
2 *)
  expose:
    (* connection from A to component1 *)
    Eq[2] :=  component1`Va == Va;
    Eq[3] :=  component1`Ia == -Ia;

    (* connection between components *)
    Eq[4] :=  component2`Va == component1`Vb;
    Eq[5] :=  component1`Ib == -component2`Ia;

    (* connection from component2 to B *)
    Eq[6] :=  component2`Vb == Vb;
    Eq[7] :=  component2`Ib == -Ib;
end;

instance RLC (* Coordinates: 28, -20 *)
part S1 inherits SerCircuit(R1, P1);
part S2 inherits SerCircuit(U0, P2);
part S3 inherits SerCircuit(Common, S2);
```

```
part S4 inherits SerCircuit(S3, Common);
part P1 inherits ParCircuit(C, R2);
part P2 inherits ParCircuit(S1, L);
expose:
  Temperature := 21;
  RoomTemperature := 20;
  DTemp := Temperature - RoomTemperature;
end;

instance R1 inherits Resistor(100, 0, 0, 1)
end;

instance R2 inherits Resistor(20, 0, 0, 1)
end;

instance C inherits Capacitor(0.1*10^(-6),0,0,1) (* Coordinates: 3, 6 *)
end;

instance L inherits Inductor(0.0015, 0, 0) (* Coordinates: 3, 5 *)
end;

instance U0 inherits VSource(10*Sin[2*Pi*50*t]) (* Coordinates: 0, 6 *)
end;
```

Appendix B

The SphereCyl model

This appendix contains the full model SphereCyl referred to earlier in this manual.

```
model SphereCyl;
  packages "Vectors`", "Subst`", "Misc`";

  g; (* Gravity constant *)

class GeneralContact(S, Body1, Body2)
  expose:
  (* 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;

class NonFrictionContact(B, Body1, Body2) inherits GeneralContact(B, Body1,
Body2)
  expose:
  (* The contact stiffness function *)
  k[delta_];

  (* 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;

class LinearContact(B, Body1, Body2) inherits NonFrictionContact(B, Body1,
Body2)
  expose:
  k[delta_] := Cstiff[delta]*delta - 1;
  Cstiff[delta_] := Csoft /; delta > 0;
  Cstiff[delta_] := Chard /; delta <= 0;
```

```
end;

instance Con12 inherits LinearContact(C1, Body1, Body2)
  expose:
end;

instance Body1 inherits Sphere(C1, {Body2})
  expose:
  Con[Body2] := Con12;
  (* Density *)
  rho;
  (* Mass *)
  m := rho*V;
  (* 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;

class Sphere(S, B) inherits Body(S, B)
  expose:
  (* 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;

instance Body2 inherits Cylinder(C2, {Body1})
  expose:
  Con[Body1] := Con12;
end;

class Cylinder(S, B) inherits Ring(S, B)
  expose:
  Rb := 0;
  u[max] := 3;
end;

class Ring(S, B) inherits Body(S, B)
  expose:
  (* 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 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;

  u[step] := 1; v[step] := Pi/15;
  u[steps] := 4; v[steps] := 31;

  begin
  x' := Dx;
  z' := Dz;

end;

class Body(S, B)
  expose:
  (* 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[steps] := 15; v[steps] := 15;
  Graphic[s_] :=
      Block[{p1 = r[s][u, v][[1]],
             p2 = r[s][u, v][[2]],
             p3 = r[s][u, v][[3]]},
            Insert[ParametricPlot3D[{p1, p2, p3},
                                    {u, u[min], u[max]}, {v, v[min], v[max]},
                                    {Axes -> False,
                                     PlotPoints -> {u[steps], v[steps]},
                                     DisplayFunction -> Identity}],
                  {DisplayFunction -> $DisplayFunction}, 2]];

  (* 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;

instance G inherits GlobalCoordinateSystem
end;

class GlobalCoordinateSystem inherits AbstractCoordinateSystem
  expose:
  FromGlobal := {this};
end;

instance C2 inherits RotatedCoordinateSystem(G, {Pi/2,0,0})
end;

class RotatedCoordinateSystem(Reference, Phi) inherits
CoordinateSystem(Reference, RotationMatrix[Phi], {0, 0, 0})
  expose:
  InverseA := Transpose[A];
  begin
  RotationMatrix[{phi1_, phi2_, phi3_}] :=
  {{Cos[phi3], Sin[phi3], 0}, {-Sin[phi3], Cos[phi3], 0}, {0, 0, 1}} .
```

```
  {{Cos[phi2], 0, -Sin[phi2]}, {0, 1, 0}, {Sin[phi2], 0, Cos[phi2]}} .
  {{1, 0, 0}, {0, Cos[phi1], Sin[phi1]}, {0, -Sin[phi1], Cos[phi1]}};
end;

instance C1 inherits TranslatedCoordinateSystem(G, {0,0,d})
end;

class TranslatedCoordinateSystem(Reference, R) inherits
CoordinateSystem(Reference, IdentityMatrix[3], R)
  expose:
  InverseA := IdentityMatrix[3];
end;

class CoordinateSystem(Reference, A, R) inherits AbstractCoordinateSystem
  expose:
  InverseA := Inverse[A];
  FromGlobal := Append[Reference`FromGlobal, this];
end;

class AbstractCoordinateSystem
  expose:
  FromGlobal; (* virtual *)
  A;          (* virtual *)
  InverseA;  (* virtual *)
  R;          (* virtual *)

  Transform[x_, toSystem_, transformFn1_, transformFn2_] :=
  Block[{globalPath1 = FromGlobal,
        globalPath2 = AccessMember[toSystem, "FromGlobal"]},
    If[First[globalPath1] == First[globalPath2],
       Block[{paths = DropWhile2[SameQ, globalPath1, globalPath2]},
        FoldR[transformFn2,
              FoldR[transformFn1, x, paths[[1]]],
              Reverse[paths[[2]]]]],
       Message[Transform::nocommon, this, toSystem]]];

  TransformVector[v_, toSystem_] :=
  Transform[v, toSystem,
           (Simplify[AccessMember[#1, "A"] . #2] //. transsubst)&,
           (Simplify[AccessMember[#1, "InverseA"] . #2] //. transsubst)&];

  TransformPoint[p_, toSystem_] :=
  Transform[p, toSystem,
           (Simplify[
             AccessMember[#1, "R"] + AccessMember[#1, "A"] . #2] //.
              transsubst)&,
           (Simplify[
             AccessMember[#1, "InverseA"] . (#2 - AccessMember[#1, "R"])] //.
              transsubst)&];

  begin
  Transform::nocommon =
  "No common global coordinate system for coordinate systems `` and ``.";

end;

instance Solving
  expose:
  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]; );
  PlotIt := ( SetPlotdata; Show[ Body1`Graphic[G], Body2`Graphic[G] ] );
  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;
```

Appendix C

GenExample ObjectMath model for code generation

This is a rather contrived small ObjectMath “model” called `GenExample`, which has been designed to illustrate the use of the two main code generation routines, `GenerateFunction` – for generating procedural code from an ObjectMath procedure, and `GenerateEvaluateFunction` – to generate code from a function body that is evaluated into a (usually huge) symbolic expression before being generated code from.

This example computes a function over a 2-D grid. The function values are first stored in a matrix before being plotted. Executing compiled Fortran90 code for the function `sinFun2` is approximately 200-400 times faster than evaluating the same function interpretively within Mathematica.

The example contains the following functions declared within instance Test:

- The function calcPlot accepts four arguments which are coordinates that describe a square in the x-y plane. For each point on a 21x21 grid in that square, a numeric function will be called that computes a value that is stored as a matrix element in the matrix representing the grid.
- The function `sinFun1` computes sin(x+y).
- The function `sinFun2` computes essentially the same values as `sinFun1`, but in a more complicated way using a rather large expression obtained through conversion of the arguments into polar coordinates (through `arcTan`) and then using series expansion of both `Sin` and `Cos` in 10 terms. The resulting large symbolic expression (more than a page) becomes the body of `sunFun2`, and is then used as input to `GenerateEvaluateFunction` to generate efficient Fortran90 or C++ code.
- The procedure `plot` draws a 3D diagram of the function values stored in the 21x21 matrix returned by calcPlot and passed as an argument to `plot`.
- The procedure `plotfile` plots the same 3D diagram, but instead reads the matrix of function values from an external file “`plot.dat`”, assuming that the generated Fortran90 or C++ code has been compiled an executed in order to produce this data file. This can be used to verify that computing externally and reading in the results give the same result as computing within Mathematica.
- The procedure `gen`(*lang*) produces Fortran90 or C++ code for the ObjectMath functions `sinFun2` and `calcPlot`, depending on the `lang` parameter.
- The `genc` procedure generates C++ code by performing the call `gen`(“C++”).
- The `genf` procedure generates Fortran90 code by performing the call `gen`(“Fortran”).

To run this example, start the ObjectMath environment as described in Chapter 2, *open* the model file "`genex.omt`", *compile* the model and download into the Mathematica process, and call `plot` in the *interaction* window to compute and plot the function within Mathematica:

```
Test`plot
```

To generate Fortran90 code, type:

```
Test`genf
```

To generate C++ code, type:

```
Test`genc
```

Compile the generated code using the supplied makefile, and run it to produce the output data file "plot.dat". You will notice that the execution is approximately 200-400 times faster than running the same computation within the Mathematica process.

To create a 3D plot of the produced data file, type:

```
Test`plotfile
```

within the interaction window.

The complete Genexample model follows below.

C.1 The code of the GenExample model

```
(* file genex.omt *)

model GenExample;

  packages "CodeGeneration`";

instance Test
  expose:

real array [21,21] xyMatrix;

(* change the function last in the inner loop to get another z-function *)
real array [21,21] calcPlot[real xmin_, real xmax_, real ymin_, real ymax_]
   := Module[ {
    int n = 20,
    real x,
    real y,
    int i,
    int j },

    For[i=1, i<=(n+1), i=i+1,
      For[j=1, j<=(n+1), j=j+1,
       x = xmin+(xmax-xmin)*(i-1)/n;
       y = ymin+(ymax-ymin)*(j-1)/n;
       xyMatrix[[i]][[j]] = sinFun2[x,y]
      ]
    ];
    xyMatrix
  ];

  real sinFun1[ real x_, real y_] := Sin[x+y];

  (* convert point to angle *)
  real arcTan[real x_, real y_] :=  (
```

```
    If[x < 0, Pi, 0] + If[ x == 0, Sign[y]*Pi/2, ArcTan[y/x] ]
  );

  (* taylor-expanded sin and cos functions *)
  real sin[ real x_ ] := Normal[Series[Sin[z], {z,0,10}]] /. z -> x ;
  real cos[ real x_ ] := Normal[Series[Cos[z], {z,0,10}]] /. z -> x ;

  (* sunFun2 gives a more complicated way (series expansion)
   * to calculate the same as sinFun1 *)
  real sinFun2[ real x_, real y_] := Block[ {r,xx,yy},
    r = Sqrt[x^2+y^2];
    xx = r*cos[arcTan[x,y]];
    yy = r*sin[arcTan[x,y]];
    sin[xx+yy]
  ];

  (* Procedure to plot the result *)
  plot := ListPlot3D[ calcPlot[-2.0,2.0,-2.0,2.0] ];

  (* procedure to generate C++ of Fortran, determined by the lang parameter *)
  gen[lang_] := (
            CodeGeneration`OpenFile["plot", lang];
            CodeGeneration`GenerateVariable[xyMatrix];
            (* CodeGeneration`GenerateFunction[sinFun1];  just for comparison *)
            CodeGeneration`GenerateEvaluateFunction[sinFun2];
            CodeGeneration`GenerateFunction[calcPlot];
            CodeGeneration`CloseFile[];
  );

  (* Procedures genf and genc, to generate Fortran90 or C++ *)
  genf := gen["Fortran"];  (* This generates Fortran90 *)
  genc := gen["C++"];      (* This generates C++ *)

  (* Procedure to plot the matrix data read from the file `plot.dat' *)
  plotfile := (
    ListPlot3D[ReadList[ "plot.dat", Number, RecordLists -> True ]]
  );

end;
```

Appendix D

Generated C++ code from the GenExample model

Below follows the generated C++ code from the `GenExample` model. Notice that name of the instance where a function is declared becomes a prefix of the name of the generated C++ function. Thus the ObjectMath function `Test`sinFun2` becomes `Test_TsinFun2` in C++.

The generated code from `Test`sinFun2` is produced from a large expression by `CompileEvaluateFunction`. Therefore common subexpression elimination is performed by the code generator, producing many temporary variables and subexpressions which can be seen in the body of the C++ function `Test_TsinFun2`..

By contrast, the C++ code in the body of function `Test_TcalcPlot` produced by `CompileFunction` from the ObjectMath function `Test`calcPlot`, follows the structure of the original code quite closely.

```
#include <math.h>
#include "cplot.h"

doubleNN Test_TxyMatrix(21, 21);
double Test_TsinFun2 (const double &P_D_Tx, const double &P_D_Ty)
{
  double om_T1;
  double om_T2;
  double om_T3;
  double om_T4;
  double om_T5;
  int om_T6;
  int om_T7;
  double om_T8;
  double om_T9;
  double om_T10;
  double om_T11;
  double om_T12;
  int om_T13;
  double om_T14;
  double om_T15;
  double om_T16;
  double om_T17;
  double om_T18;
  double om_T19;
  double om_T20;
  double om_T21;
  double om_T22;
  double om_T23;
  double om_T24;
```

```
double om_T25;
double om_T26;
double om_T27;
double om_T28;
double om_T29;
double om_T30;
double om_T31;
double om_T32;
double om_T33;
double om_T34;
double om_T35;
double om_T36;
double om_T37;
double om_T38;
double om_T39;
double om_T40;
double om_T41;
double om_T42;
double om_T43;
double om_T44;
double om_T45;
double om_T46;
double om_T47;
double om_T48;
double om_T49;
double om_T50;
double om_T51;
double om_T52;
om_T1 = pow(P_D_Tx, 2);
om_T2 = pow(P_D_Ty, 2);
om_T3 = om_T1+om_T2;
om_T4 = double (1)/2;
om_T5 = pow(om_T3, om_T4);
om_T6 = P_D_Tx == 0;
om_T7 = sign (P_D_Ty);
om_T8 = om_T7*3.14159265358979323846;
om_T9 = om_T8/2;
om_T10 = P_D_Ty/P_D_Tx;
om_T11 = atan(om_T10);
if (om_T6)
{
  om_T12 = om_T9;
}
else
{
  om_T12 = om_T11;
}
om_T13 = P_D_Tx < 0;
if (om_T13)
{
  om_T14 = 3.14159265358979323846;
}
else
{
  om_T14 = 0;
}
om_T15 = double (-1)/6;
om_T16 = om_T12+om_T14;
om_T17 = pow(om_T16, 3);
om_T18 = om_T15*om_T17;
om_T19 = pow(om_T16, 5);
```

```
  om_T20 = om_T19/120;
  om_T21 = double (-1)/5040;
  om_T22 = pow(om_T16, 7);
  om_T23 = om_T21*om_T22;
  om_T24 = pow(om_T16, 9);
  om_T25 = om_T24/362880;
  om_T26 = om_T12+om_T14+om_T18+om_T20+om_T23+om_T25;
  om_T27 = om_T5*om_T26;
  om_T28 = double (-1)/2;
  om_T29 = pow(om_T16, 2);
  om_T30 = om_T28*om_T29;
  om_T31 = pow(om_T16, 4);
  om_T32 = om_T31/24;
  om_T33 = double (-1)/720;
  om_T34 = pow(om_T16, 6);
  om_T35 = om_T33*om_T34;
  om_T36 = pow(om_T16, 8);
  om_T37 = om_T36/40320;
  om_T38 = double (-1)/3628800;
  om_T39 = pow(om_T16, 10);
  om_T40 = om_T38*om_T39;
  om_T41 = 1+om_T30+om_T32+om_T35+om_T37+om_T40;
  om_T42 = om_T5*om_T41;
  om_T43 = om_T27+om_T42;
  om_T44 = pow(om_T43, 3);
  om_T45 = om_T15*om_T44;
  om_T46 = pow(om_T43, 5);
  om_T47 = om_T46/120;
  om_T48 = pow(om_T43, 7);
  om_T49 = om_T21*om_T48;
  om_T50 = pow(om_T43, 9);
  om_T51 = om_T50/362880;
  om_T52 = om_T27+om_T42+om_T45+om_T47+om_T49+om_T51;
  return om_T52;
}

void Test_TcalcPlot (const double &P_D_Txmin, const double &P_D_Txmax,
  const double &P_D_Tymin, const double &P_D_Tymax, doubleNN &om_O1)
{
  int j;
  int i;
  double y;
  double x;
  int n;
  n = 20;
  i = 1;
  while (i <= n+1)
  {
    j = 1;
    while (j <= n+1)
    {
      x = P_D_Txmin+(P_D_Txmax-P_D_Txmin)*(double ((i-1))/(n+1));
      y = P_D_Tymin+(P_D_Tymax-P_D_Tymin)*(double ((j-1))/(n+1));
      Test_TxyMatrix(i, j) = Test_TsinFun2 (x, y);
      j = j+1;
    }
    i = i+1;
  }
  om_O1 = Test_TxyMatrix;
}
```

Appendix E

Generated Fortran90 code from the GenExample model

Below follows the generated Fortran90 code from the `GenExample` model. Notice that name of the instance where a function is declared becomes a prefix of the name of the generated Fortran90 function. Thus the ObjectMath function `Test'sinFun2` becomes `Test_TsinFun2` in Fortran90.

The generated code from `Test'sinFun2` is produced from a large expression by `CompileEvaluateFunction`. Therefore common subexpression elimination is performed by the code generator, producing many temporary variables and subexpressions which can be seen in the body of the Fortran90 function `Test_TsinFun2`..

By contrast, the Fortran90 code in the body of function `Test_TcalcPlot` produced by `CompileFunction` from the ObjectMath function `Test'calcPlot`, follows the structure of the original code quite closely.

```
module om_precision
  integer, parameter :: om_real = kind(1.0D0)
  integer, parameter :: om_int = kind(1000000000)
end module om_precision


module om_GenExample
  use om_precision
  implicit none

! procedure declarations

  interface
    function Test_TsinFun2(P_D_Tx, P_D_Ty) result (om_O1)
      use om_precision
      implicit none
      real(om_real), intent(in) :: P_D_Tx
      real(om_real), intent(in) :: P_D_Ty
      real(om_real) om_O1
    end function  Test_TsinFun2
  end interface

  interface
    subroutine Test_TcalcPlot(om_O1, P_D_Txmin, P_D_Txmax, P_D_Tymin, P_D_Tymax)
      use om_precision
      implicit none
      real(om_real), intent(out) :: om_O1(21,21)
      real(om_real), intent(in) :: P_D_Txmin
      real(om_real), intent(in) :: P_D_Txmax
```

```
      real(om_real), intent(in) :: P_D_Tymin
      real(om_real), intent(in) :: P_D_Tymax
    end subroutine  Test_TcalcPlot
  end interface

! variable declarations
  real(om_real) Test_TxyMatrix(21,21)
  common /GenExample/ Test_TxyMatrix


end module om_GenExample


recursive function Test_TsinFun2(P_D_Tx, P_D_Ty) result (om_O1)
  use om_GenExample, om_hide_Test_TsinFun2 => Test_TsinFun2
  implicit none
  real(om_real), intent(in) :: P_D_Tx
  real(om_real), intent(in) :: P_D_Ty
  real(om_real) om_O1
  real(om_real) om_T1
  real(om_real) om_T2
  real(om_real) om_T3
  real(om_real) om_T4
  real(om_real) om_T5
  logical  om_T6
  integer(om_int) om_T7
  real(om_real) om_T8
  real(om_real) om_T9
  real(om_real) om_T10
  real(om_real) om_T11
  real(om_real) om_T12
  logical  om_T13
  real(om_real) om_T14
  real(om_real) om_T15
  real(om_real) om_T16
  real(om_real) om_T17
  real(om_real) om_T18
  real(om_real) om_T19
  real(om_real) om_T20
  real(om_real) om_T21
  real(om_real) om_T22
  real(om_real) om_T23
  real(om_real) om_T24
  real(om_real) om_T25
  real(om_real) om_T26
  real(om_real) om_T27
  real(om_real) om_T28
  real(om_real) om_T29
  real(om_real) om_T30
  real(om_real) om_T31
  real(om_real) om_T32
  real(om_real) om_T33
  real(om_real) om_T34
  real(om_real) om_T35
  real(om_real) om_T36
  real(om_real) om_T37
  real(om_real) om_T38
  real(om_real) om_T39
  real(om_real) om_T40
  real(om_real) om_T41
  real(om_real) om_T42
```

```
real(om_real) om_T43
real(om_real) om_T44
real(om_real) om_T45
real(om_real) om_T46
real(om_real) om_T47
real(om_real) om_T48
real(om_real) om_T49
real(om_real) om_T50
real(om_real) om_T51
real(om_real) om_T52

om_T1 = P_D_Tx**2
om_T2 = P_D_Ty**2
om_T3 = om_T1+om_T2
om_T4 = real (1)/2
om_T5 = om_T3**om_T4
om_T6 = P_D_Tx == 0
om_T7 = sign (1.0D0, P_D_Ty)
om_T8 = om_T7*3.1415926535897932
om_T9 = om_T8/2
om_T10 = P_D_Ty/P_D_Tx
om_T11 = atan(om_T10)
if (om_T6) then
  om_T12 = om_T9
else
  om_T12 = om_T11
end if
om_T13 = P_D_Tx < 0
if (om_T13) then
  om_T14 = 3.1415926535897932
else
  om_T14 = 0
end if
om_T15 = real ((-1))/6
om_T16 = om_T12+om_T14
om_T17 = om_T16**3
om_T18 = om_T15*om_T17
om_T19 = om_T16**5
om_T20 = om_T19/120
om_T21 = real ((-1))/5040
om_T22 = om_T16**7
om_T23 = om_T21*om_T22
om_T24 = om_T16**9
om_T25 = om_T24/362880
om_T26 = om_T12+om_T14+om_T18+om_T20+om_T23+om_T25
om_T27 = om_T5*om_T26
om_T28 = real ((-1))/2
om_T29 = om_T16**2
om_T30 = om_T28*om_T29
om_T31 = om_T16**4
om_T32 = om_T31/24
om_T33 = real ((-1))/720
om_T34 = om_T16**6
om_T35 = om_T33*om_T34
om_T36 = om_T16**8
om_T37 = om_T36/40320
om_T38 = real ((-1))/3628800
om_T39 = om_T16**10
om_T40 = om_T38*om_T39
om_T41 = 1+om_T30+om_T32+om_T35+om_T37+om_T40
om_T42 = om_T5*om_T41
```

```
  om_T43 = om_T27+om_T42
  om_T44 = om_T43**3
  om_T45 = om_T15*om_T44
  om_T46 = om_T43**5
  om_T47 = om_T46/120
  om_T48 = om_T43**7
  om_T49 = om_T21*om_T48
  om_T50 = om_T43**9
  om_T51 = om_T50/362880
  om_T52 = om_T27+om_T42+om_T45+om_T47+om_T49+om_T51
  om_O1 = om_T52
end function  Test_TsinFun2


subroutine Test_TcalcPlot(om_O1, P_D_Txmin, P_D_Txmax, P_D_Tymin, P_D_Tymax)
  use om_GenExample, om_hide_Test_TcalcPlot => Test_TcalcPlot
  implicit none
  real(om_real), intent(out) :: om_O1(21,21)
  real(om_real), intent(in) :: P_D_Txmin
  real(om_real), intent(in) :: P_D_Txmax
  real(om_real), intent(in) :: P_D_Tymin
  real(om_real), intent(in) :: P_D_Tymax
  integer(om_int) j
  integer(om_int) i
  real(om_real) y
  real(om_real) x
  integer(om_int) n

  n = 20
  i = 1

  do while (i <= n+1)
    j = 1

    do while (j <= n+1)
      x = P_D_Txmin+(P_D_Txmax-P_D_Txmin)*(real ((i-1))/(n+1))
      y = P_D_Tymin+(P_D_Tymax-P_D_Tymin)*(real ((j-1))/(n+1))
      Test_TxyMatrix(i, j) =  Test_TsinFun2 (x, y)

      j = j+1

    end do

    i = i+1

  end do
  om_O1 = Test_TxyMatrix
end subroutine  Test_TcalcPlot
```

E.1 Main program calling the generated code

```
PROGRAM test

REAL*8 xyMatrix(21,21)
INTEGER i,j
CALL Test_TcalcPlot( xyMatrix, -2.0D0, 2.0D0, -2.0D0, 2.0D0)
PRINT '(21F12.8)', ((xyMatrix(i,j), j=1,21),i=1,21)
END
```

Appendix F

OOA Diagram of the 2D Bearing Model

Figure F.1 below shows the class structure of the 2D bearing model described in Chapter 7, which has the same class structure as previously shown in Figure 7.2 but here drawn using the OOA notation by Coad and Yourdon [Coa91] for comparison. The only important difference between the OOA notation and the ObjectMath notation, is that in OOA a component (part) might belong to several aggregates, whereas an ObjectMath part always belongs to exactly one aggregate. Note that some other object-oriented notations, e.g. the one used in OOSE [Jac92], take the same view as ObjectMath on this issue.

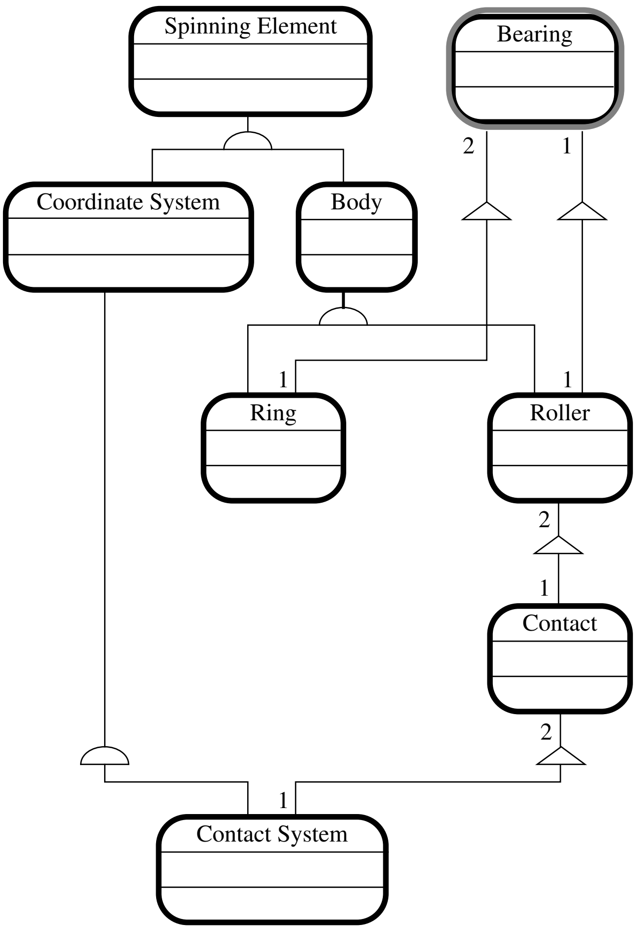


Figure F.1: *The class structure of the 2D bearing model (see also Figure 7.2), here drawn using the OOA notation by Coad and Yourdon.*

Appendix G

Formal syntax of ObjectMath extensions

Below is a BNF grammar of the ObjectMath language extensions of the Mathematica language.

```
<model>            →  <model head> [ <packages> ] [ <global> ]
                          {@ <class def> | <instance def>}
<model head>       →  "model" <id> ";"
<packages>         →  "packages" <string> {@ "," <string> } ";"
<global>           →  {@ <local def> }
<class def>        →  <class header> <object body>
<instance def>     →  <instance header> <object body>
<class header>     →  "class" <id> [ <formal params> ] [ <inherits> ]
<instance header>  →  "instance" <id>
                          [ <indices> ] [ <inherits> ]
<inherits>         →  "inherits" <parent> {@ "," <parent> }
<parent>           →  <id> [ <actual params> ]
<formal params>    →  "(" <id list> ")"
<actual params>    →  "(" <expr list> ")"
<indices>          →  "[" <id list> "]"
<object body>      →  {@ <part def> } {@ <local block> }
                          [ "begin" <expr> ] "end" ";"
<part def>         →  "part" <id> [ <indices> ] <inherits> ";"
<local block>      →  <visibility> [ <access list> ] ":"
                          {@ <local def> }
<visibility>       →  "hide" | "reveal" | "expose"
<access list>      →  "{" <id list> "}"
<local def>        →  ( <name def> | <array def> | <symbol def> )";"
<name def>         →  [ "static" | "once" ] ( <name> [ ":=" <expr> ]
                                     | <typed name def> )
<array def>        →  "array"\ "[" <integer list> "]" <name list>
<symbol def>       →  "symbol" <id list>
<name>             →  <id> {@ "[" <name index list> "]" }
<name index list>  →  <name index> {@ "," <name index> }
<name index>       →  <name> | <pattern> | <integer>
<name list>        →  <name> {@ "," <name> }
<expr list>        →  <expr> {@ "," <expr> }
<id list>          →  <id> {@ "," <id> }
<integer list>     →  <integer> {@ "," <integer> }

<typed name def>   →  <typed func def> | <typed var def>
<typed func def>   →  <typed func head> <typed func body>
<typed var def>    →  <type> <simple name> [ <attributes> ]
<typed func head>  →  <return type> <simple name> "[" [ <typed patterns> ] "]"
<return type>      →  "{" "}" | <type> | "{" <type> {@ "," <type>} "}"
<simple name>      →  <id> [{@ "[" <name-or-int list> "]" }]
<typed patterns>   →  <type> <parameter> {@ "," <type> <parameter> }
<typed func body>  →  ":=" ( <expr> | <module> ) | "extern" <string>
```

```
<type>              →  <base type> [ [ "nonexpanded" ] "array"
                                              "[" <integer list> "]" ]
<name-or-int_list> →  <name-or-int> {@ "," <name-or-int> }
<module>            →  "Module" "[" "{" [ <local var>
                             {@ "," <local var> }] "}", <expr> "]"
<local var>         →  [ <type> ] <id> [ "=" <expr> ]
<base type>         →  "real" | "int" | "bool"
<name-or-int>       →  <simple name> | <integer>
<attributes>        →  "with" <attribute> {@ <attribute>}
<attribute>         →  <id> "=" <string>
```

The syntax of ObjectMath expressions, denoted <*expr*> in the grammar, is the same as for Mathematica expressions with the following exceptions: (**Mma syntax as described in section ??)

- The token “this” is a reserved word.
- The token “:==” is an operator with the same priority and associativity as the assignment operators “=”, “:=” etc.
- The token “=>” is an operator with the same priority as function application. It is left associative. The second (rightmost) operand of the operator must be a symbol, possibly containing context marks.
- The token “\\” is an prefix operator with priority between the operator “<<” and forms containing “_”. The operand must be a symbol, possibly containing context marks.
- White-space may not be used for denoting multiplication.
- The ObjectMath parser should ignore the input control issues and special input. (** described in sections ?? and ??)

Bibliography

(** Note: Very preliminary version, to be updated. UPDATE scientific programming and IEEE Software references).

[And95] Niclas Andersson and Peter Fritzson. Generating parallel code from object oriented mathematical models. In proceedings of the *Fifth ACM SIGPLAN Symposium om Principles and Practice on Parallel Programming*, Santa Barbara, California, pp. 48-57, July 19-21, 1995.

[And96] Niclas Andersson. Generating Parallel Code in the ObjectMath Programming Environment. PREPARE Deliverable D8.8.3/1.

[Boo91] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[Cel91] Francois E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

[CelZC91] F. E. Cellier, B. P. Zeigler, and A. H. Cutler. Object-oriented modeling: Tools and techniques for capturing properties of physical systems in computer code. In *Proceedings of the IFAC Symposium Computer Aided Design in Control Systems*, pages 1–10. Pergamon Press, July 1991.

[Cha91] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1991.

[Coa91] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1991.

[Coo90] G. O. Cook, Jr. ALPAL, a program to generate physics simulation codes from natural descriptions. *International Journal of Modern Physics*, 1(1):1–51, 1990.

[Dav88] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra — Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.

[Ell90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[Elm93] H. Elmqvist. Object-oriented modeling and automatic formula manipulation in Dymola. In T. Iversen, editor, *SIMS'93, Applied Simulation in Industry — Proceedings of the 35th SIMS Simulation Conference*. Scandinavian Simulation Society, June 1993.

[Fri92] P. Fritzson and D. Fritzson. The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers & Structures*, 45(2):387–395, 1992.

[FriEV93] Peter Fritzson, Vadim Engelson, Lars Viklund. Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment. In *Proc. of DISCO'93 - Conf. on Design and Implementation of Symbolic Computation Systems, LNCS 722,* Springer Verlag, Sept. 1993.

[Fri95] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High Level Mathematical

Modeling and Programming in Scientific Computing. *IEEE Software*, pp. 77-87, July 1995.

[Gol83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[Gri93] G. Grivas and R. E. Maeder. Matching and unification for the object-oriented symbolic computation system AlgBench. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 164–176. Springer-Verlag, 1993.

[Hea87] A. C. Hearn. *REDUCE-3 User's Manual, version 3.3*. The Rand Corporation, Santa Monica, Califoria, USA, 1987. Publication CP78 (7/78).

[Hee89] J. Heering, P. R. K. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — reference manual. *ACM SIGPLAN Notices*, 14(11):43–75, November 1989.

[HPFF93] High Performance Fortran Forum (HPFF). High Performance Fortran language specification, May 1993.

[Jac92] I. Jacobsson, M. Christersson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering—A Use Case Driven Approach*. Addison-Wesley Publishing Company, 1992.

[Jen92] R. D. Jenks and R. S. Sutor. *AXIOM — The Scientific Computation System*. Springer-Verlag, 1992.

[Jen88] R. D. Jenks, R. S. Sutor, and S. M. Watt. Scratchpad II: An abstract datatype system for mathematical computation. In J. R. Rice, editor, *Mathematical Aspects of Scientific Software*, volume 14 of *The IMA Volumes in Mathematics and Its Applications*, pages 157–182. Springer-Verlag, 1988.

[Kan93] E. Kant. Synthesis of mathematical modeling software. *IEEE Software*, 10(3):30–41, May 1993.

[Kan90] E. Kant, F. Daube, W. MacGregor, and J. Wald. MathCode: A code generation package for Mathematica. draft, October 1990.

[Koe94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., Mary E. Zosel. The High Performance Fortran Handbook. Scientific and Engineering Computation Series. The MIT Press, 1994.

[Mat93] S. E. Mattsson, M. Andersson, and K. J. Åström. Object-oriented modelling and simulation. In D. A. Linkens, editor, *CAD for Control Systems*, chapter 2, pages 31–69. Marcel Dekker Inc, New York, 1993.

[Nor94] Patrik Nordling, Peter Fritzson. Solving Ordinary Differential Equations on Parallel Computers - applied to Dynamic Rolling Bearing Simulation, In Dongarra, Hansen and Wasniewski, editors, *PARA'94 Workshop on Parallel Scientific Computing*. UNI-C Danmarks EDB-Center for Forskning och Uddannelse. Copenhagen, June 1994., LNCS 879, Springer Verlag.

[Pag89] T. W. Page, Jr., S. E. Berson, W. C. Cheng, and R. R. Muntz. An object-oriented modeling environment. In *OOPSLA'89 Conference Proceedings*, pages 287–296, 1989.

[Pal92] R. S. Palmer and J. F. Cremer. SimLab: Automatically creating physical systems simulators. Computer Science Technical Report TR92-1246, Cornell University, 1992.

[Pal94] R. S. Palmer. Chain models and finite element analysis. Computer Science Technical Report TR94-1406, Cornell University, 1994.

[PieE91] 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.

[Pie87] C. Pierret-Golbreich. Object-centered knowledge representation for modelling in biology. In *International Symposium on AI, Expert Systems and Languages in Modeling and Simulation*, pages 281–286, June 1987.

[Sym85] Symbolics Inc. *MACSYMA Reference Guide*, 1985.

[Vik92] L. Viklund, J. Herber, and P. Fritzson. The implementation of ObjectMath — a high-level programming environment for scientific computing. In U. Kastens and P. Pfahler, editors, *Compiler Construction — 4th International Conference, CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 312–318. Springer-Verlag, 1992.

[Vik95] Lars Viklund, Peter Fritzson: ObjectMath – An Object-Oriented Language and Environment for Symbolic and Numerical Processing in Scientific Computing. *Scientific Programming*, Vol. 4, pp. 229-250, 1995..

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):8–87, August 1990.

[Wol91] S. Wolfram. *Mathematica — A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, second edition, 1991.