Accepted 1994 for IEEE Software. A thoroughly revised version will appear in IEEE Software, July 1995.

ObjectMath – An Environment for High-Level Mathematical Modeling and Programming in Scientific Computing

Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber*
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Email: petfr@ida.liu.se, Phone: +46 13 281484

**SKF Engineering and Research Centre B.V.*
Postbus 2350, 3430 DT Nieuwegein, The Netherlands

Abstract: The current state of the art of programming in scientific computing is still rather low-level. A substantial part of a typical project effort is often spent on implementation and debugging.

To improve this situation, we have designed ObjectMath, which is a high-level programming environment and modeling language for scientific computing. The language integrates object-oriented constructs such as classes, single and multiple inheritance, with computer algebra features. Objects are used to structure models that might consist of a large number of equations. Using ObjectMath, it is possible to model classes of equation objects, to support inheritance of equations, to model part-of relations between objects, and to solve systems of equations. The class specialization hierarchy is also used to generate specialized efficient numeric code from general symbolic equations and expressions. The ObjectMath environment is being used for industrial applications and is designed to handle realistic problems. This is achieved by allowing the user to specify transformations and simplifications of formulae in the model, to arrive at a representation which is efficiently solvable by numerical methods. Model equations can be transformed to C++ or FORTRAN code for efficient numerical solution.

Some experience is reported from successful use of ObjectMath, e.g. in representing and solving several realistic problems. The re-use of equations through inheritance in general reduces model size by a factor of two to three, compared to a direct representation in the Mathematica computer algebra language. Another observation is that multiple inheritance from orthogonal classes sometimes facilitates re-use and maintenance of application models.

Keywords: Object oriented, Scientific computing, Computer algebra, Compilation, Symbolic transformation, Mechanical Analysis, Programming environment

1 Introduction

The goal of the ObjectMath project is to develop a high-level programming environment that enhances the program development process in scientific computing, initially applying the environment to advanced machine element analysis (a machine element can loosely be defined as “some important sub-structure of a machine”). There is a clear need for such tools as the current state of the art in the area is still very low-level. Most scientific software is still developed in FORTRAN the traditional way, manually translating mathematical models into procedural code and spending much time on debugging and fixing convergence problems. See [FF91b] for a detailed discussion. The ObjectMath programming environment is centered around the ObjectMath modeling language which combines object-oriented constructs with computer algebra. In this paper we describe the ObjectMath environment and report experience from using it for modeling and analyzing realistic machine element analysis problems, in addition to generating efficient numerical code for solving such problems.

The structure of the rest of the paper is as follows: First we review the current state of the art in scientific computing, and propose an alternative improved development strategy. Then, the structure of typical models in the machine elements application domain are analyzed. This is illustrated in terms of a rolling bearing example in addition to some smaller examples. The requirements of a very high level programming and modeling environment are reviewed. Then the design, use, implementation and applications of the ObjectMath environment are discussed. The appendix contains some information on the topic of parametric surfaces.

2 Motivation

The current state of the art in modeling and programming in scientific computing and advanced mechanical analysis is still rather low-level. A designer often spends more than half the time and effort of a typical project in writing and debugging FORTRAN programs. These programs are written in order to perform numerical experiments to evaluate and optimize a mathematical model, e.g. of a machine element. Numerical convergence problems often arise, since the optimization problems usually are non-linear. A substantial amount of time is spent on modifying the program to achieve convergence.

Feedback from results of numerical experiments usually lead to revisions in the mathematical model, which subsequently require re-implementing the FORTRAN program. This whole process, depicted in Figure 1, is rather laborious and error prone.

There is a clear need for a higher level programming environment that would eliminate most of these low-level problems and allow the designer to concentrate on the modeling aspects.

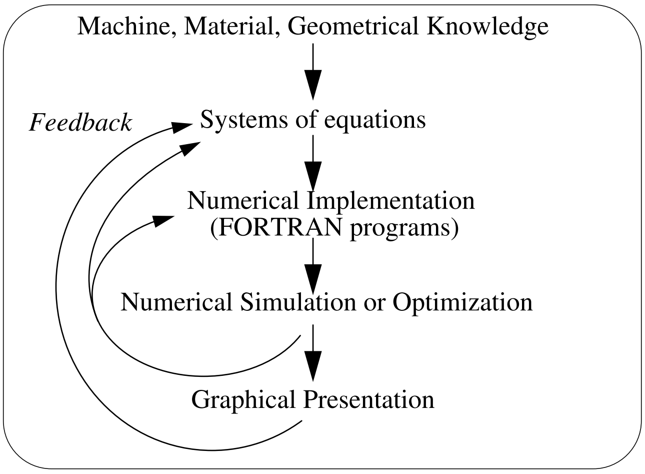

Figure 1: The iterative process of modeling in traditional mechanical analysis. Often 50-75% of the time is spent on FORTRAN programming

2.1 Application domain: modeling in mechanical analysis

At the current state of the art, there are two main classes of activities within research projects or advanced development projects in mechanical analysis of machine elements:

- Theory development
- Numerical computation

During theory development, a mathematical model is developed that describes the relevant properties of the investigated machine element. This is normally done manually, using only pen and paper.

In order to perform numerical computations, the model must be translated to some programming language, or to input specifications for some existing numerical modeling program, such as for example a finite element analysis program. Normally, existing high level tools can at best be used for limited parts of the total computational problem, because it usually is too complex.

2.2 The ideal programming environment: programming in high-level equations

The ideal high level programming environment would automatically transform systems of equations into efficient symbolic and numerical programs. It would select optimization routines with good convergence properties for the given problem. The environment would also aid in formulating equations given geometrical constraints and transforming equations between different coordinate systems. However, fully automatic versions of some of these capabilities will be hard to achieve. It is more realistic to assume that the user will work in dialogue with the interactive system, and that the user can supply valuable hints and information that will guide the system to choose the right algorithms and transformations. Another important advantage of a mathematical representation over a FORTRAN implementation is that more inherent parallelism of the problem is preserved since no dependencies have been introduced through low level implementation details. Thus better results can be expected when generating code for parallel machines. Some desired capabilities of the environment are listed below:

- Support in expressing systems of equations, e.g. handling geometrical constraints and coordinate transformations
- Integration of object-oriented techniques in mathematical modeling
- Algebraic transformations of equations
- Compilation/transformation of model equations to efficient numerical programs
- Transformation of equations for computation on parallel hardware
- Automatic selection of proper numerical algorithms to achieve convergence

- Evaluation of Numerical Experiments
- Graphical presentation

3 Related work

In this section we mention some related work and currently available tools, their strengths and weaknesses. A more detailed review, including full references and discussion can be found in [FF91b]. So far, none of these tools offer exactly what we want, although some provide a subset of the desired capabilities.

One group of tools and methods includes *finite element analysis* (FEM), *Multibody systems analysis* programs, *CAD (Computer Aided Design)* and *CAE (Computer Aided Engineering)*. Some of these tools could theoretically solve the above problems. However, they usually generate masses of low-level numerical data, which is not what we want. Also, they tend to be very inefficient, especially on non-linear problems. They provide no mathematical model development support.

Computer algebra systems such as *Mathematica* [Wol91], *Maple* [CGGMW88], *Axiom* [JS92], *Macsyma* and *Reduce* are good at representing and manipulating symbolic formulae and equations but are lacking model development support and efficient numerical computation, although *Axiom* has a type system which in some sense is object oriented. However, their symbolic capabilities are needed in our desired environment. Matrix languages such as *MATLAB*, provide convenient programming of matrix, vector and linear algebra related problems. However, non-linear problems are not handled and symbolic formula capabilities are missing. FINGER [Wan88] is a system specialized to the automatic generation of finite element analysis programs from symbolic descriptions. However, this system is applicable only to problems which can be solved using certain FEM methods.

A system that has some similarities to the ObjectMath programming environment is ALPAL [Coo90]. This system is integrated with Macsyma. However, ALPAL is not object-oriented and seems to be specialized towards development of certain kinds of physics simulation programs. Another system, the Sinapse program synthesis system [Kan92]supports design, implementation and evaluation of mathematical models. Compared to ObjectMath it supports a rather narrow application area of certain partial differential equations, but on the other hand provides more automatic program synthesis support in this special domain.

A related system featuring a modeling language called Omola (Object-oriented MOdeling LAnguage) has been developed for control and process engineering applications [And90]. A similar system, called ASCEND, is described in [PEWW91]. Dymola is an earlier system without inheritance, which has been used for larger practical applications than Omola. Several examples using Dymola are described in [Cel91]. In comparison to ObjectMath, the Omola and ASCEND environments are *not* integrated with a computer algebra system, and do *not* permit interactive manipulation of equations, which is a key property for achieving efficient implementations in complex application domains such as mechanical analysis. By contrast equations represented in Omola are relatively simple. The emphasis is rather on automatic transformation and solution of simpler sets of equations in differential algebraic form.

4 The Structure of Models in the Machine Element Application Domain

In order to make our discussion more concrete, we will focus on one application domain in scientific computing, namely mechanical analysis with machine elements. However, most of our results are also applicable to other areas of scientific computing. In the machine element application domain, we need to describe the structure of parts, components and their substructure, together with material data and interaction between components. These together represent the total machine element. As an example machine element model, we will use a rolling bearing model throughout part of this paper. The picture of the rolling bearing in Figure 2 has been automatically generated from model equations expressed in the ObjectMath modeling language.

4.1 Introducing objects

Now we introduce the term *"object"*. An object can represent many different things and we can have different classes of objects. Some examples of what an object can be are:

- A real three dimensional body.
- A set of equations.
- A coordinate system.
- A set of material data.

Regarding the example rolling bearing, depicted in Figure 2, one can conclude that there are primarily two types of information that can be grouped into objects:

- *Self-contained* objects
  These are objects which are more or less self-contained, e.g. physical objects or abstract objects such as coordinate systems. They describe a certain object and not the relations between objects.
- *Connection* objects
  The main purpose of these objects is to describe relations and connections between other objects. A typical example is a contact between two physical bodies.

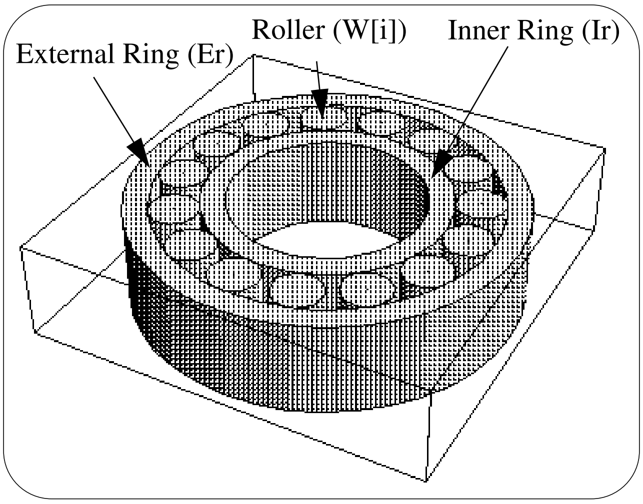

Figure 2: A three-dimensional view of the simplified rolling bearing, automatically generated from the model example equations.

Another observation is that it is quite useful to use existing objects when defining new objects. New objects can be defined through *composition* and combination of existing objects which then become *parts of* (see also section 6.5) the composite object, or as *specialization* of existing objects using *inheritance*. For example, a cylindrical roller element can inherit many properties from an existing general cylinder object, and add special information such as size and material properties.

4.2 Self-contained objects

In this class of objects there are several natural different object classes that describe various aspects of the studied problem. Some examples of basic objects:

- A geometry description of a physical body.
- A coordinate system.
- A theory that describes the structural deformation of a body given loads or displacements on the body.
- A material model for a material.
- Material data for a certain solid or fluid.

When defining the *external ring object*, as depicted in Figure 2, it is natural to use and combine several of the basic objects defined above in order to give an appropriate description of a general *simple ring object*. We then create the *external ring* machine element object, by inheriting from the general ring object, and adding properties that are special for the external ring. This inheritance hierarchy is shown in Figure 3.

When later defining an object such as the *inner ring*, we can create a new object that inherits information from the general ring object, and subsequently add specialized information.

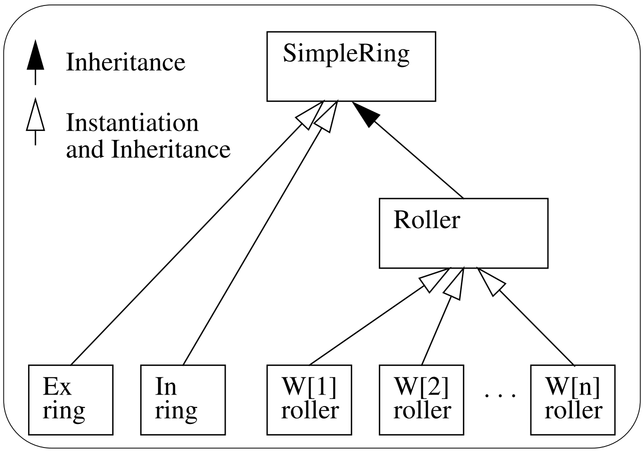

Figure 3: Inheritance and instantiation of self-contained equation objects for the rolling bearing in Figure 2.

4.3 Connection objects

The purpose of *connection objects* is to describe relations and connections between self-contained objects. This corresponds to forming systems of equations in traditional analysis.

Examples of connection objects are:

- Fixed contacts or connections between bodies.
- An elastohydrodynamic contact between two bodies.
- Equilibrium conditions for a body or a set of bodies.
- A constraint which reduces the number of degrees of freedom.
- An application of a certain displacement, rotation, or force field to a body.

It is possible to classify connection objects into two loose groups:

- *Binary connection objects* that involve two self-contained objects. Examples are contact problems.
- *One to many connection objects* that involve a relation between one self-contained object and many other self-contained objects. An example is a force field which is influencing many bodies.

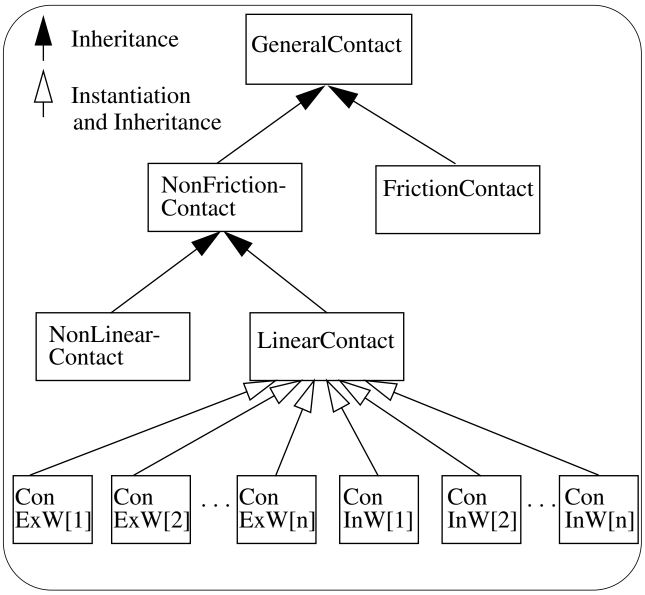

Figure 4: Inheritance and instantiation of connection objects for the rolling bearing in Figure 2.

When defining new *connection objects* it is natural to combine existing basic connection objects, or to use inheritance to re-use most definitions from general connection objects when defining classes of more specialized objects. For example, a general contact can be specialized into the two classes non-friction and friction contacts, where all general contact and equilibrium equations can be re-used. The class of non-friction contact objects can then be further specialized into linear or non-linear contacts, etc., as in Figure 4.

5 Requirements on the Ideal Programming Environment

Below, we make a quick overview of some desired properties of an ideal programming environment. Our current implementation of this environment provides some support within each category, with an emphasis on model specification and model transformation.

1. Model specification

A machine element object, such as a ring, will have attributes and associated operations. As an example, attributes for a ring are the radius, a formula for parametric definition of the surface of the ring, a formula for elasticity properties etc. Associated operations can for example be to compute the normal vector to the surface of the ring, to compute the volume, or to graphically visualize it on the screen.

The programming environment should support the user in defining such objects. In addition, the creation of composite objects, and inheritance of attributes and equations from general objects to more specialized objects should be supported. It should be possible to model constraints and connections between physical objects within *connection* objects.

The use of such object-oriented programming techniques for modeling using equations eliminates unnecessary repetition of similar formulae, imposes a better structure on the mathematical model, and reduces the overall number of equations.

2. Model transformation

The user should interactively be able to transform equations and formulae associated with an object, with the help of a computer algebra system. Transformed formulae should be saved within new versions of the objects from which they were derived. It should be possible to specify which variables should be solved for, and which variables are input or output.

3. Choice of numerical solution routines

The user should be able to choose between a few built-in numerical solution and optimization routines, obtained from some appropriate numerical library. Specification of, e.g. iteration start values and stop conditions should be supported. All this information should be associated with the appropriate equation object.

4. Generation of numerical code and computation.

Formulae and equations produced during the interactive symbolic transformation phase should be used to generate efficient numerical code, e.g. in efficiently compiled languages such as for example FORTRAN or C++. Code for automatic call of selected numerical routines should be automatically generated.

5. Visualization of results

Resulting data should be amenable to visualization in two or three dimensions. By using parametric equations for defined objects, this can be obtained, e.g. by calling general 3D graphics routines which accept parametric surface definitions.

6 The ObjectMath Language and Environment

ObjectMath is both a language and a programming environment. The programming environment is designed to be easy to use e.g. in mechanical analysis by application engineers, who are not computer scientists. It is interactive and includes a graphical browser. Another part of the environment contains support for the generation of numerical code and for visualization. Initial experience from using an early version of ObjectMath is reported in [FVHF92].

The ObjectMath language is an hybrid modeling language, combining object-oriented constructs with a language for symbolic computation. This makes ObjectMath a suitable language for implementing complex mathematical models, such as those used in machine element analysis. Formulae and equations can be written with a notation that closely resembles conventional mathematics, while the use of object-oriented modeling makes it possible to structure the model in a natural way.

Similar to the computer algebra language Mathematica ObjectMath is dynamically typed – type checking is performed at run-time during evaluation. Function lookup is performed through a pattern matching mechanism at run-time. Note, however, that dynamic object creation is not needed for mathematical models since equations and model structure are statically described. Therefore, ObjectMath function lookup in inheritance hierarchies is static, which is also more efficient. The implementation of the inheritance mechanism supports static binding, similar to languages such as Simula or C++. The dynamic typing allows redefinition of methods and other attributes in subclasses.

However, it is possible to specify groups of similar objects using a set-like feature, where the number of objects in such a group can be left unspecified during symbolic transformations or specified either at compile-time or at run-time. We have chosen to use an existing computer algebra language, Mathematica, as a basis for ObjectMath. Building an object-oriented layer on top of an existing language is not ideal, but this decision was made to make it possible to implement a stable system that could be used to test the feasibility of our approach. The current ObjectMath language is a preliminary version and we plan to redesign the language in the future to include type declarations and other extensions, and also to make it more independent of Mathematica. On the other hand, keeping a reasonable compatibility with Mathematica may facilitate a more wide-spread use of ObjectMath because of the installed base of Mathematica with approximately 200 000 users.

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

6.1 ObjectMath Classes and Instances

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

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

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

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

On the other hand, access to equations defined in different classes is necessary when specifying computer-algebra related transformations, substitutions and simplifications. Such symbolic operations on equations need to access the source representation of those equations. Since transformations are specified manually and executed interactively, equations which should be symbolically combined with equations from other classes should not be hidden from the user. Enforcing access functions for accessible attributes, similar to Smalltalk, would be very clumsy in this respect. The extreme of enforcing encapsulation of all attributes is only possible in systems such as Dymola [Cel91], which do not allow manual manipulation of equations.

We are currently working on a revision of the ObjectMath language, where the level of encapsulation can be specified in a flexible way, similar to what is possible in many other object-oriented languages. Thus, attributes such as equations and functions can either be locally or globally visible, depending on what is needed.

6.2 Naming in ObjectMath

In addition to equations and variables, ordinary functions and procedures (=methods) can be associated with an object. Since the Mathematica context facility is used in our implementation of objects, the normal dot-notation for referring to intra-class definitions has been replaced by Mathematica's apostrophe notation for referring to definitions within a class. This can be seen in the example:

```
Coord`TransPoint[r[Ex][u,v],Ex,WNo[i]]
```

where the `TransPoint` *method* within the `Coord` object is called.

One convention used in models that we have designed so far is that each equation within an object is named, e.g. `Eq[1]`, `Eq[2]`, in order to allow reference or application of such equations within other objects. See for example equations within the `GeneralContact` class in section 6.3. Another naming convention could be `Eq[SimpleRing1]`, `Eq[SimpleRing2]` for two equations within the `SimpleRing` class mentioned in Figure 3, which might be advantageous for avoiding name collisions when inheriting equations.

Another naming convention in this model is to let the symbol within the first pair of brackets always denote the coordinate system within which the relevant quantity is defined. For example, the quantity `r` in: `r[WNo[i]][u,v]` denotes the parametric surface function `r` of parameters `u` and `v`, defined within the `WNo[i]` coordinate system. Note that these two conventions are not part of the actual ObjectMath language – they just serve to make models more consistent and easier to understand.

6.3 Single Inheritance Examples

In this section we use some of the classes from the rolling bearing model and some other applications to exemplify the ObjectMath language. Note that the inheritance hierarchy usually is edited graphically so that the user does not have to write the class headers by hand.

A CLASS example

The following is an example of a class which contains four equations that hold for all contacts between bodies. These equations will be inherited by more specialized classes and connection objects. The parameter `B` denotes the relevant coordinate system; `Body1` and `Body2` denote the two objects involved in the contact.

```
CLASS GeneralContact(B, Body1, Body2)
  ....
  (* Equations *)
  Eq[1] := Body1`n[B][u[Body1], v[Body1]] +
    Body2`n[B][u[Body2], v[Body2]] =={ 0, 0, 0 };
  Eq[2] := delta Body1`n[B][u[Body1], v[Body1]]
          == Body2`r[B][u[Body2], v[Body2]] -
             Body1`r[B][u[Body1], v[Body1]];
  Eq[3] := Body1`F[B][Body2] + Body2`F[B][Body1]
          == { 0, 0, 0 };
  Eq[4] := Body1`M[B][Body2] + Body2`M[B][Body1]
          == { 0, 0, 0 };

  (* The contact point *)
  p[B] := Body1`r[B][u[Body1], v[Body1]] +
       1/2 delta Body1`n[B][u[Body1], v[Body1]];
 (* The contact point in other coordinate systems
 *)
  p[c_] := B`TransformPoint[p[B], c];
END GeneralContact;
```

Below, the first four equations are shown in standard mathematical notation as a comparison. The normal vector of `Body1`, which in object-oriented notation is denoted `Body1`n[B]` is shown as $\bar{n}_1^{(B)}$ using standard subscript notation. The parameter variable values at the contact $u_1$, $v_1$ and $u_2$, $v_2$ respectively, are local to the `GeneralContact` class. For forces and moments, the first subscript denotes which body it belongs to, and the second which body it interacts with.

$$\bar{n}_1^{(B)}(u_1, v_1) + \bar{n}_2^{(B)}(u_2, v_2) = \bar{0}$$

$$\Delta \cdot \bar{n}_1^{(B)}(u_1, v_1) = \bar{r}_2^{(B)}(u_2, v_2) - \bar{r}_1^{(B)}(u_1, v_1)$$

$$\bar{F}_{2,1}^{(B)} + \bar{F}_{1,2}^{(B)} = \bar{0}$$

$$\bar{M}_{2,1}^{(B)} + \bar{M}_{1,2}^{(B)} = \bar{0}$$

The `SimpleRing` class mentioned in Figure 5 and

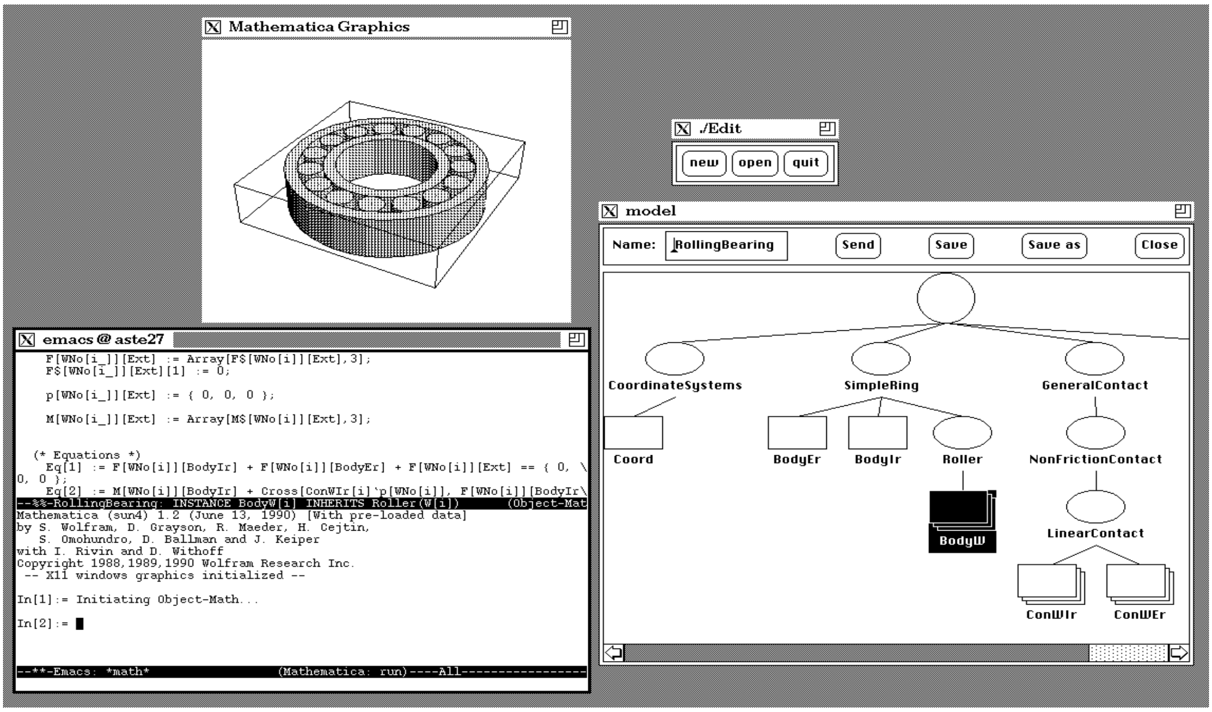

Figure 5: The ObjectMath high-level modeling environment on the 200-equation rolling bearing example. The three-dimensional view of the bearing has been automatically generated from equations expressed in ObjectMath.

Figure 9 contains general ring properties, such as parametric surface equations for a ring, which are explained in the appendix. These equations are inherited by other objects such as the external ring, the inner ring, and the rolling elements. All these can be regarded as specialized rings. The notation `/;` is for equations with guarding conditions - the equation only applies when the guard is true.

An INSTANCE Example

This instance class declaration specifies the external ring object, which inherits equations from the `SimpleRing` class. For example, the equation `Eq[1]` specifies the well known law that the sum of all forces acting on a system in equilibrium is zero. In this case, the sum of all forces on the rolling elements `BodyW[i]` where i varies from `1` to `Wn`, and the external force.

Note: Double brackets `[[]]` are used for vector indexing, e.g. `A[[1]]` for the first component of vector A. Single brackets `[]` are used for parameterized naming, e.g `A[one]` is a new variable different from `A`. Underscore is used at the end of pattern matching variable names in matching rules.

```
INSTANCE BodyEr INHERITS SimpleRing(Er)
(*Forces & pure moments from contact with rollers
*)
  F[c_][BodyW[i_]] := Array[F$[c][BodyW[i]],3];
  M[c_][BodyW[i_]] := Array[M$[c][BodyW[i]],3];

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

  (* Equations *)
  Eq[1] := Sum[F[ErG][BodyW[i]], {i, 1, Wn}] +
                F[ErG][Ext] == { 0, 0, 0 };
  Eq[2] := Sum[M[ErG][BodyW[i]] +
              Cross[ConErW[i]`p[ErG],
           F[ErG][BodyW[i]]], {i, 1, Wn}] +
            M[ErG][Ext] +  Cross[p[ErG][Ext],
                 F[ErG][Ext]] == { 0, 0, 0 };
  Eq[3][i_] := F[ErG][BodyW[i]] ==
WNo[i]`TransformVector[F[WNo[i]][BodyW[i]],ErG];

  Eq[4][i_] := M[ErG][BodyW[i]] ==
WNo[i]`TransformVector[M[WNo[i]][BodyW[i]],ErG];
END BodyEr;
```

6.4 Examples of Multiple Inheritance

The first version of the ObjectMath language, which only supported single inheritance, was later 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.

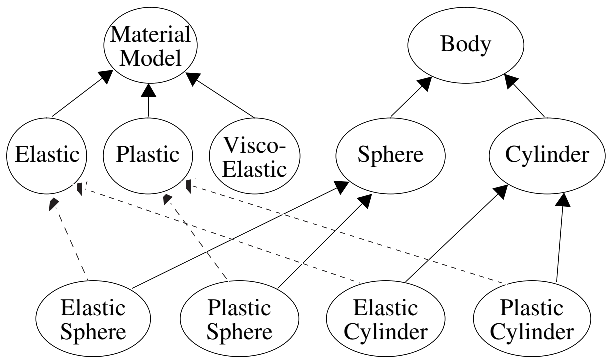

Figure 6: Multiple inheritance hierarchy of bodies of different materials and geometries.

Multiple inheritance is useful when combining orthogonal concepts. This is exemplified in Figure 6. The filled lines denote single inheritance, whereas the dotted lines denote additional inheritance, i.e. we have multiple inheritance. Since material properties and geometry are orthogonal concepts there are no collisions between inherited definitions in this example.

If instead we are forced to use a single-inheritance hierarchy as in Figure 7, we have to repeat the equations describing material properties twice, which is bad model engineering. Also, this precludes the creation of pure material library classes which can be combined with other classes. Another disadvantage is that such a single inheritance model structure is harder to maintain when changes are introduced into the model.

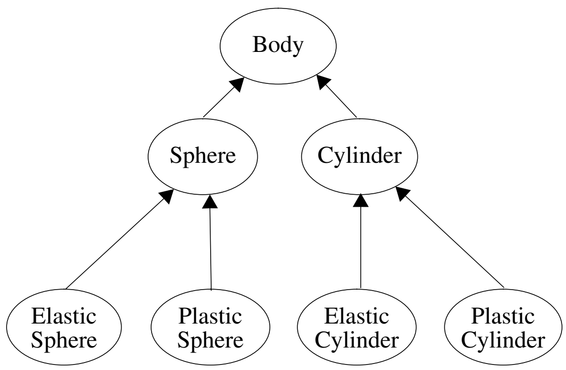

Figure 7: Single inheritance version of the material-geometry model of Figure 6.

The general form of multiple inheriting class declarations follows below:

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

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

These are some example classes from the material-geometry model in Figure 6:

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

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

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

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

Another useful case of *multiple-inheritance* is shown below, where an integration method is inherited into classes from two separate inheritance hierarchies shown in Figure 8: a hierarchy of contact classes containing integrated forces and moments between bodies, and classes describing bodies themselves including integrated moments of inertia, mass and volumes.

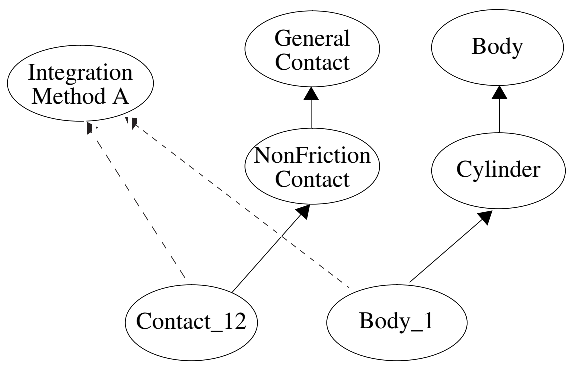

Figure 8: Example of multiple inheritance of a numerical integration method into two different classes. Here to be used for integrating forces or volumes. One class contains contact equations; another contains volumes, moments and equilibrium equations.

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

$$\vec{r}^{(C)}(u, v) = \begin{bmatrix} x(u) \cdot \cos v \\ x(u) \cdot \sin v \\ z(u) \end{bmatrix}$$

$$u \in [u_{min}, u_{max}] \qquad u_{min} = 0 \qquad u_{max} = 4$$

$$v \in [v_{min}, v_{max}] \qquad v_{min} = 0 \qquad v_{max} = 2\pi$$

$$x(u) = \begin{cases} R_a + (R_a - R_b)\,u & 0 \le u < 1 \\ R_a & 1 \le u < 2 \\ R_a + (R_a - R_b)\,(3 - u) & 2 \le u < 3 \\ R_b & 3 \le u \le 4 \end{cases}$$

$$z(u) = \begin{cases} \frac{1}{2}L & 0 \le u < 1 \\ -L\left(u - \frac{3}{2}\right) & 1 \le u < 2 \\ -\frac{1}{2}L & 2 \le u < 3 \\ L\left(u - \frac{7}{2}\right) & 3 \le u \le 4 \end{cases}$$

```
CLASS SimpleRing(C)
  (* The C parameter is the ring-centered
                               coordinate system *)
  ...
  (* Variables *)
  Rb;
  Ra;
  L;
  u[min] := 0; u[max] := 4;
  v[min] := 0; v[max] := 2 Pi;

  (* Definition of parametric surface r(u,v) *)

  x[u_] := Rb + (Ra - Rb)u       /; 0<=u< 1;
  x[u_] := Ra                    /; 1<=u< 2;
  x[u_] := Rb + (Ra - Rb)(3-u)   /; 2<=u< 3;
  x[u_] := Rb                    /; 3<=u<=4;

  z[u_] := 1 / 2 L               /; 0<=u< 1;
  z[u_] := -L (u - 3 / 2)        /; 1<=u< 2;
  z[u_] := - 1 / 2 L             /; 2<=u< 3;
  z[u_] := L (u - 7 / 2)         /; 3<=u<=4;

  r[C][u_, v_] := {x[u] Cos[v],
                   x[u] Sin[v],
                   z[u] };

  ...

END SimpleRing;
```

Figure 9: Parametric surface equations in both mathematical and in ObjectMath form. See also the appendix.

6.5 Modeling Part-Of Relations

As mentioned previously, the *part-of* relation is important for modeling objects which are composed of other objects. 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 exemplified below for a `Bicycle` class:

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

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

```
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 Implementation of Inheritance, Classes and Instances

Inheritance between *classes*, and between *classes* and *instances* is implemented by copying definitions of equations and other attributes. Only *instance* declarations actually give rise to generated Mathematica code, which of course contains all equations from classes inherited by the instance. An *instance* declaration with set brackets will generate one instance of Mathematica code and several data objects.

For example, the *instance* declaration with set brackets: **`INSTANCE`** `BodyW[i]` **`INHERITS`** `Roller(W[i])` can give rise to several data objects (indexed by **`i`**) which can be used during simulation of the model.

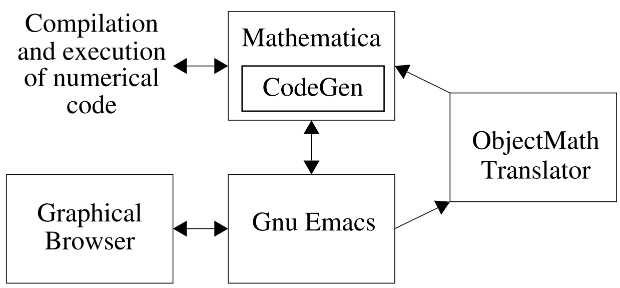

Figure 10: Process structure of the implementation. Interprocess communication is through Unix pipes, sockets and pseudo teletypes.

6.7 System Implementation

The current version of the ObjectMath environment and language has been implemented using a combination of Gnu Lisp, Scheme, Mathematica and C++ (see Figure 10), running on a Sun Sparcstation.

A parser and translator for the hybrid language ObjectMath has been implemented in Scheme. Object oriented language features such as classes, inheritance and instances are implemented by transformations of the internal representation generated by the parser to appropriate Mathematica code. The code generator that generates C++ code from annotated Mathematica code produced by the ObjectMath translator is currently implemented as a module in Mathematica. This facilitates intermixing code generation and symbolic transformation of formulae.

The current ObjectMath environment supports a separate editing window/emacs buffer for each class or instance definition. One window for each object definition simplifies future extensions such as *versioning* of such definitions. A graphical *browsing* tool, see Figure 5, implemented using the ET++ class libraries, is also included in the ObjectMath environment. A new version of the browser which more clearly shows part-of relations and multiple inheritance has recently been developed based on the InterViews user interface library. The ObjectMath environment also provides an interaction window where the user can interactively perform simplifications, substitutions and other transformations on model equations using Mathematica.

7 Combined Symbolic/Numeric Computation

Analyzing a mathematical model expressed in ObjectMath also involves performing numeric computations. The Mathematica system can be used for some of these calculations, providing quick turn-around in the model-development cycle. However, there are problems associated with this approach. Mathematica code is interpreted and cannot be executed as efficiently as programs written in compiled languages such as C, C++, or FORTRAN. The difference in performance was measured to be a factor of 360 compared to compiled code generated from a small model. Bigger models are not feasible to execute within Mathematica since the difference in performance increases to a factor of several thousands. At most, a few specific points on the solution curve can be computed for the purpose of checking. This is a serious drawback, particularly when doing mostly numerical computations in realistic applications. Another drawback is that accuracy of numerical computations in Mathematica is hard to evaluate. We also want to be able to use existing, highly optimized, special purpose numerical routines.

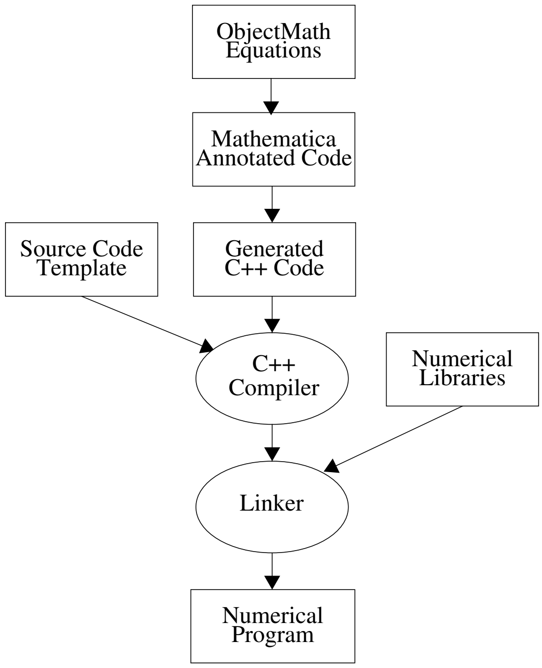

Figure 11: Generating a numerical program from a mathematical equation-based model

The ObjectMath environment provides a code generation facility that produces numerical code from ObjectMath models which have been translated into annotated Mathematica code. The annotations provide type information which is necessary for the code generator to produce efficient code. Common subexpression elimination is performed during the code generation process. This turned out to be necessary for producing efficient code (both a speedup and a size reduction of around a factor of 10 was measured for some examples) from the huge expressions generated through symbolic transformations, and also to split up these expressions in pieces that can be handled by the C++ compiler. The system also allows mixing ObjectMath and C++ code, see section 7.3, enabling us to take advantage of symbolic computation while still being able to write time-critical functions in a language that can be compiled to efficient code. C++ is used because it is object-oriented and reasonably efficient. An advantage is that vectors and matrices can be conveniently implemented as objects, and thus numerical code becomes quite compact and readable through operator overloading of vector and matrix operations. A new matrix and vector library in C++ has been developed to this end, which is a factor of 10 faster for small 3x3 matrix operations than more general commercial libraries such as M++, and approximately of the same efficiency as handwritten C code using C arrays.

7.1 Solving Systems of Equations

The ObjectMath environment supports the automatic generation of numerical programs for solving systems of linear or non-linear equations expressed in the ObjectMath language. Solving is done by numerical solvers implemented in FORTRAN which are linked to the generated C++ code. Our current main application is rolling bearing simulation, which primarily gives rise to ordinary differential equations (ODEs). After some experimentation, we have selected the FORTRAN-based ODE solver LSODA [Pet83] which is a version of the LSODE solver from ODEPACK that has been modified to automatically switch between several methods to become more efficient and have better numerical stability properties. One of the input parameters to this solver is a routine which calculates the values of functions which are referenced within equations from the model. This routine is generated using the ObjectMath→C++ code generation facility described previously.

Generated programs can either be transparently called from within the ObjectMath environment or be used independently of the environment as a computation kernel, for example together with a graphical front-end.

7.2 Generating Parallel Code

We have more recently done work on generating parallel code from mathematical models. The basic idea is to perform dependence analysis of equations in the model in order to detect groups of inter-dependent equations which then form strongly connected components in the dependence graph. Each such component represents a system of equations which often can be solved in parallel with other systems of equations extracted from the model.

Our experience so far is that for some application models this works nicely and you obtain a set of equation systems that can be solved in parallel, but for other applications most equations are closely interdependent and almost no coarse-grained parallelism in terms of separate systems of equations can be extracted from the model. However, for both types of models finegrained parallelism can be utilized during the ordinary differential equation solution process (our current applications lead to ordinary differential equations) by evaluation of equation right-hand sides in parallel. Some preliminary experience of both of these approaches is reported in [FA93]. Our current efforts in this area are primarily on generating efficient code, and efficient scheduling of the parallel code.

7.3 Integrating ObjectMath and C++

The ObjectMath environment also provides the possibility to mix ObjectMath and C++ code. This is useful when the user has to express efficient numerical code to implement his or her own numerical routines, for instance when standard solvers cannot be used. An ObjectMath class declaration can include one or more external C++ functions as methods (in addition to functions/methods expressed directly in ObjectMath).

As an example, consider the class `FindingContacts` taken from an ObjectMath model of an advanced contact problem. Instances of this class model contacts between two bodies. `FindingContacts` is defined by inheriting a contact class called `GeneralContact` from the ObjectMath class library. The class definition includes one method (`ppa`) defined as an external C++ function:

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

The facility described here of providing external routines within ObjectMath classes was provided with the second release of the system. However, it turned out to be less useful to the end users than expected. The reason is that this facility of being able to call isolated numerical routines solves a very limited part of the programming problem. For realistic models, numerical model-dependent code tends to reference definitions and formulae from many parts of the model. Thus, to conveniently generate efficient implementations, an ambitious code generator that can translate whole models is needed. Such a code generator, briefly described in the beginning of section 7, was later developed and included in the third release of the ObjectMath system.

8 Application and Usage Experience

As mentioned before, simulations of rolling bearing models have been our main applications so far. Given the mathematical model and transformations specified by the user, the system can generate a numerical program that computes interesting quantities. For example, given external forces, the displacements between the rings and rolling elements of the bearing can be computed, or alternatively, a program can be generated that computes the forces given the displacements.

The current version of the ObjectMath environment is the third which has been released to our industrial project partner since the start of the project more than 3 years ago. This version of the environment, although not perfect, is being used to generate efficient implementations of size around half a Megabyte of C++ code from mathematical models in mechanical analysis. Earlier versions of the environment mostly supported modeling aspects in a more limited manner, and were less stable for practical usage. For large applications, it turns out that about half of the code can be generated from the mathematical model. This code is more or less the model- or application dependent part of the implementation. The rest of the implementation is composed of standard or special procedural numerical algorithms which are hand-written or exist as library routines.

Some parts of the environment, such as the browser, are very easy to use, whereas other aspects such as mathematical modeling require substantially more expertise. Fresh graduate students used 2-4 weeks to become proficient users, able to design medium sized mathematical models. Most of this time was spent learning the Mathematica portion of the ObjectMath language for modeling and computer algebra transformations of equations. Other issues taking time were how to design the object-oriented structure (classes, inheritance and composition) of mathematical models. This is largely unexplored territory, but we have gained some experience in modeling which we intend to describe as guidelines for the benefit of new users of the system.

The greatest benefits in using the system are the computer supported mathematical modeling and the generation of efficient implementations which are consistent with the specification in terms of a mathematical model. The automatic generation of code instead of manual implementation eliminates many errors compared to manual implementation. It also provides flexibility in terms of being able to generate several implementations from the same model, for the purpose of different types of solutions. However, the user has to specify which symbolic transformations should be applied to parts of the model before invoking the code generator. This requires some expertise from the user, but also provides a means for the user to control the final code quality.

For the most complex models we have tried so far, the symbolic transformations during the code generation may take up to 2 hours on a standard sparcstation 2. For small and medium-sized models this time is closer to 5-15 minutes, but is also heavily dependent on the amount of symbolic transformations specified by the user. The system also provides the possibility of very quick turnaround in testing models by executing generated code for small examples interpretively within Mathematica, and avoiding final code generation. However, this is not feasible for non-trivial computations since this interpreted code tends to be several thousand times slower than final compiled model-generated code in those cases.

9 ObjectMath Compared to the Ideal Environment

In the beginning of this paper we mentioned a number of properties for an ideal programming environment in Scientific Computing. It is useful to examine to what extent the current ObjectMath environment fulfils these requirements. Below we repeat each requirement and compare it with ObjectMath.

- *Support in expressing systems of equations, including handling geometrical constraints and coordinate transformations*
  Mathematica already provides a general notation for expressing almost all kinds of mathematics. This is also part of ObjectMath. To handle geometry, we use parametric surface description techniques, since these usually work well from a numeric point of view, and also can be used to generate pictures. We have also defined an inheritance hierarchy of coordinate system classes, which automatically produces symbolic coordinate transformations when needed.
- *Integration of object-oriented techniques in mathematical modeling*
  This has been achieved in ObjectMath as previously described. Both modeling through classes, single and multiple inheritance, and composition of parts is supported. A graphical browser facilitates visualizing and editing the model structure.
- *Algebraic transformations of equations.*
  The Mathematica system, which is part of the ObjectMath environment, provides interactive computer algebra support for transformations. The environment translates ObjectMath models which are loaded into a Mathematica pro-

cess for this purpose. There are some differences in the structure of translated equations compared to how they appear in the model. To solve this problem, we have implemented a small translator which transforms single equations from the Mathematica form into the Objectmath form, thus making the interaction loop for algebraic transformations transparent to the user.

- *Compilation/transformation of model equations to efficient numerical programs*
  This important functionality is provided by ObjectMath, as has been described in this paper. However, the user has to provide guidance for the translation process by inserting type information, transformation rules and certain annotations which determine to what extent and to which parts algebraic transformations and other optimizations should be applied to obtain efficient code.
- *Transformation of equations for computation on parallel hardware*
  As mentioned earlier, we have recently started work in this area. For certain models, course-grained parallelism can be extracted, but for other models only fine-grained parallelism can be utilized during the solution of ODEs. This functionality has not yet been incorporated into the current release.
- *Automatic selection of proper numerical algorithms to achieve convergence*
  No support in currently provided by ObjectMath in this area. Some prototype systems have been developed elsewhere, based on expert system technology. This might be incorporated into ObjectMath in the future.
- *Evaluation of Numerical Experiments*
  An output evaluation program has been implemented to help the user visualize and evaluate numerical results from a particular application area, namely Rolling Bearing simulations. Arbitrary result parameters can be selected interactively for 2D plotting. Additionally, 3D pictures of the bearing can be generated from the mathematical description of the bearing combined with the results from the simulation. Animation showing a rotating bearing is also provided. In the future, a more generic evaluation package should be provided to cover a broader application area.

10 Conclusions

There is a strong need for efficient high level tools and languages in scientific computing. We feel that the ObjectMath approach is highly successful in satisfying part of this need. The object oriented features of the language allow better structure of models and permit re-use of equations through inheritance and composition. This conclusion is supported by the successful modeling of realistic applications, primarily in the domain of mechanical analysis for the simulation of rolling bearings.

We have also seen that the complexity of realistic systems of equations in mechanical analysis makes it essential that the environment allows the user to interactively supply extra information to guide the solution process. However, once these extra solution steps have been supplied, the transformation from high-level model down to executable numerical code is largely automatic.

The major part of the learning effort to use the system is the time (usually a few weeks) needed to learn how to use a computer algebra system, in this case Mathematica, since equations and algebraic transformations are expressed largely in the Mathematica subset of Objectmath. Some effort is also required to learn how to apply object oriented modeling. Note that mathematical modeling skills are required both for the traditional way, using pen and paper, or using ObjectMath. However, an important advantage the ObjectMath approach is the flexibility of generating several different implementations from the same model, and automatically keeping the high level specification consistent with the implementation. For large bearing simulation applications with irregular and complex geometry, we have the experience that approximately half of the code is generated from the model, whereas the other half is hand-written or library code containing numerical algorithms, input-output, etc. This is encouraging, since the application dependent part is mostly generated, and the rest consists of reusable components such as library routines.

11 Acknowledgments

This project is supported by NUTEK - the Swedish Board for Industrial and Technical Development, by TFR - the Swedish Board for Technical Research, and by SKF. Part of the work is done within the Esprit III project PREPARE with support from the European Commission.

Appendix: Parametric Surfaces

The concept of parametric surface descriptions will perhaps need some explanation. Normally a three-dimensional surface is described as an equation with three variables, for example the surface of a sphere is defined by $x^2 + y^2 + z^2 = R^2$. By using two additional variables (parameters), this equation can be transformed to parametric form.

By varying the two parameters u and v within their respective intervals, all points on the surface of the sphere can be described.

The surface $\bar{r}(u, v)$ of the ring object is most conveniently expressed in the local cartesian coordinate system of the body. The origin in this example is located at the center of mass. A rotation surface is obtained by rotating a rectangular curve. The curve is defined in the $x - z$ plane as a function of the parameter *u,* and consists of four curve segments. The parameter interval is 1 for each segment, which means that the total parameter interval is $u \in [0, 4]$. Note that u has no physical meaning. It is just a parameter needed to cause traversal of the border of the rectangle.

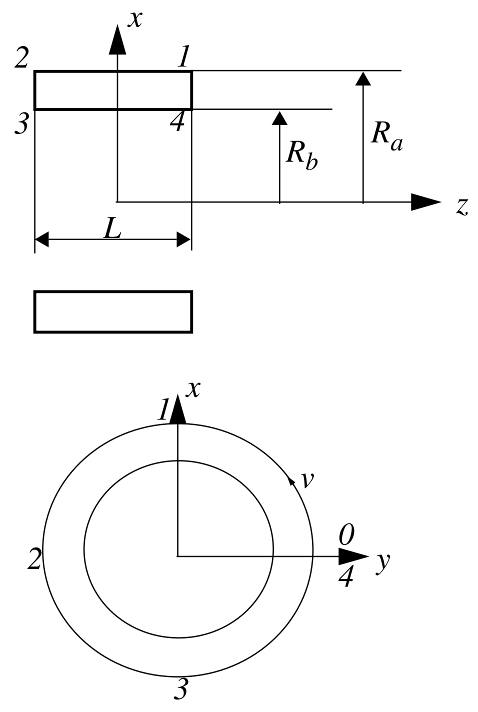

Figure 12: A ring defined as a rotation body. A rectangular curve is defined in the x-z plane as a function of u in the interval [0,4]. This rectangle is rotated to create the parametric surface which defines the ring body.

The curve is rotated by increasing the parameter v from 0 to 4. This creates one segment with a parameter interval $v \in [0, 4]$ which corresponds to $[0, 2\pi]$. The surface is closed with a positive normal vector $\bar{\boldsymbol{n}}$ directed outwards, so we have in fact a representation of a solid. The surface $\bar{r}(u, v)$ of the ring is defined as follows:

$$\bar{r}(u, v) = \begin{bmatrix} x(u)\cos\left(\frac{\pi v}{2}\right) \\ x(u)\sin\left(\frac{\pi v}{2}\right) \\ z(u) \end{bmatrix} \qquad \text{(Eq 1)}$$

$$u \in [u_{min}, u_{max}] \qquad u_{min} = 0 \qquad u_{max} = 4 \qquad \text{(Eq 2)}$$

$$v \in [v_{min}, v_{max}] \qquad v_{min} = 0 \qquad v_{max} = 4 \qquad \text{(Eq 3)}$$

$$x(u) = \begin{cases} R_b + (R_a - R_b)\,u & 0 \le u < 1 \\ R_a & 1 \le u < 2 \\ R_b + (R_a - R_b)\,(3 - u) & 2 \le u < 3 \\ R_b & 3 \le u < 4 \end{cases} \qquad \text{(Eq 4)}$$

$$z(u) = \begin{cases} \frac{1}{2}L & 0 \le u < 1 \\ -L\left(u - \frac{3}{2}\right) & 1 \le u < 2 \\ -\frac{1}{2}L & 2 \le u < 3 \\ L\left(u - \frac{7}{2}\right) & 3 \le u < 4 \end{cases} \qquad \text{(Eq 5)}$$

$$R_a > R_b \qquad \text{(Eq 6)}$$

$$R_b \ge 0 \qquad \text{(Eq 7)}$$

$$L \ge 0 \qquad \text{(Eq 8)}$$

The variables R_a and R_b denote the inner and outer radius of the ring, respectively, and L denotes the width of the ring. Figure 12 shows these variables and the intervals of the two parameters, u and v, in graphic form. Two help functions *x(u)* and *z(u)* have been introduced to clarify the description of $\bar{\boldsymbol{r}}$. It can also be noted that the definition of the function `r` below from the class `SimpleRing` in Figure 9 is practically identical to the mathematical definition above.

```
r[C][u_, v_] := { Cos[(Pi v)/2] x[u],
                  Sin[(Pi v)/2] x[u],
                  z[u] };.
```

References

[And90] Mats Andersson. Omola – an object-oriented language for model representation. Licentiate thesis, Department of Automatic Control, Lund Institute of Technology, P.O. Box 118, S-221 00 Lund, Sweden, May 1990.

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

[CGGMW88] Char, Geddes, Gonnet, Monagan, and Watt. *Maple Reference Manual*. WATCOM Publications, 5th edition, 1988.

[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.

[FF91b] Peter Fritzson and Dag Fritzson. The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers and Structures,* Vol 45, No 2, pp 387-395, 1992.

[FVHF92] Peter Fritzson, Lars Viklund, Johan Herber, and Dag Fritzson. Industrial application of object-oriented mathematical modeling and computer algebra in mechanical analysis. In Georg Heeg, Boris Magnusson, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems – TOOLS 7*. Prentice Hall, 1992.

[FA93] Peter Fritzson, Niclas Andersson. Generating Parallel Code from Equations in the ObjectMath Programming Environment. In *Proc of the 2nd Int. Conf. of the Austrian Center for Parallel Computation*, Oct 1993. In LNCS 734, Springer Verlag.

[JS92] Richard D. Jenks, Robert S. Sutor: *AXIOM - The Scientific Computation System.* Springer Verlag, 1992.

[Kan93] Elaine Kant. Synthesis of Mathematical Modeling Software. *IEEE Software*, May 1993.

[Pet83] Linda Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. SCI. STAT. COMPUT.* Vol. 4, No. 1, March 1983.

[PEWW91] 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 Chem. Engineering*, Vol. 15, No. 1, pp. 53-72, 1991.

[VF92] Lars Viklund, Peter Fritzson: *An Object-Oriented Language for Symbolic Computation - Applied to Machine Elements Analysis.* In Proc of ISSAC'92 - International Symposium on Symbolic and Algebraic Computation, Berkeley, July 1992.

[Wan86] Paul S. Wang. FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. *Journal of Symbolic Computation*, 2:305–316, 1986.

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