ObjectMath—An Object-Oriented Language and Environment for Symbolic and Numerical Processing in Scientific Computing

LARS VIKLUND AND PETER FRITZSON

Programming Environments Laboratory, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden

ABSTRACT

ObjectMath is a language for scientific computing that integrates object-oriented constructs with features for symbolic and numerical computation. Using ObjectMath, complex mathematical models may be implemented in a natural way. The ObjectMath programming environment provides tools for generating efficient numerical code from such models. Symbolic computation is used to rewrite and simplify equations before code is generated. One novelty of the ObjectMath approach is that it provides a common language and an integrated environment for this kind of mixed symbolic/numerical computation. The motivation for this work is the current low-level state of the art in programming for scientific computing. Much numerical software is still being developed the traditional way in Fortran. This is especially true in application areas such as machine elements analysis, where complex nonlinear problems are the norm. We believe that tools like ObjectMath can increase productivity and quality, thus enabling users to solve problems that are too complex to handle with traditional tools.

1 INTRODUCTION

The aim of the ObjectMath (an object-oriented mathematical language for scientific computing) project is to develop a high-level programming environment for scientific computing that supports programming in equations instead of low-level procedural programming. The high-level equational representation also gives better chances to utilize the inherent parallelism of a problem for generating efficient code for parallel hardware. There is a clear need for such tools because of the way most scientific software is currently being developed: in Fortran, the traditional way, manually translating mathematical models into procedural code and spending much time on debugging. We believe that this abstraction level is far too low.

As an initial example application domain, we have chosen machine element analysis. (A machine element can loosely be defined as "some important substructure of a machine.") This work is done in close cooperation with SKF Engineering & Research Center B. V.,* which enables us to apply the developed programming environ-

Received March 1993
Revised April 1994

Scientific Programming, Vol. 4, pp. 229–250 (1995)
CCC 1058-9244/95/040229-22

* Postbus 2350, 3430 DT Nieuwegein, The Netherlands.

ment to realistic problems, and get important feedback and suggestions on design decisions and problem-solving approaches.

This article is organized as follows: In Section 2 we describe our view on the software development process in scientific computing, and motivate the need for tools like ObjectMath. Section 3 describes the ObjectMath modeling language and in Section 4 an example of an ObjectMath model is presented. Section 5 gives an overview of how numerical code can be generated with ObjectMath, Section 7 presents related work, and Section 8 our conclusions.

2 BACKGROUND

Our view of the software development process in scientific computing is depicted in Figure 1. The input to the process is knowledge about the application domain in question, for instance knowledge about machine elements, materials, and geometry. The first step is to formulate a mathematical model describing the system being analyzed. To prepare for implementation of a numerical program, the equations in the model are simplified and rewritten symbolically. Then the model is translated into code in some programming language, or into input data for some existing mathematical software. Finally, after numerical experiments have been performed, the results are usually visualized graphically to make them more comprehensible. The program development process is highly iterative. Problems with the numerical implementation often arise, requiring changes to the numerical program or even to the mathematical model. It might also be the case that the numerical results do not correspond to practical experiments, which means that the model has to be refined, which subsequently requires changes in the numerical program.

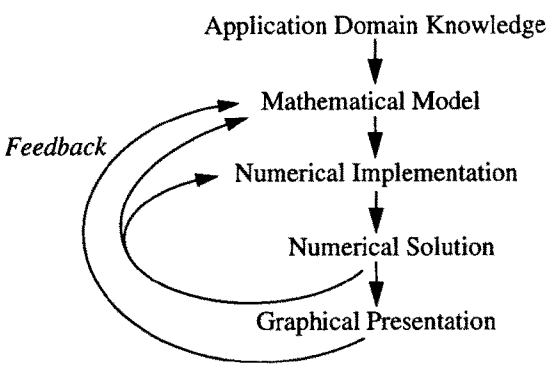

FIGURE 1 The software development process in scientific computing.

The current practice in software modeling and implementation for mechanical analysis can be described as follows: Theory development is usually done manually, using only pen and paper. Equations are simplified and rewritten by hand to prepare solving for the desired variables. This includes a large number of coordinate transformations, which are laborious and error prone to carry out. To perform numerical computations a program is written by hand, usually in Fortran. Existing numerical subroutines might be used, but large parts of the applications must still be implemented. Tools such as finite element analysis (FEM) or multibody systems analysis programs can at best be used for limited subproblems as the total computational problem usually is too complex. Frequently as much as 50–75% of the total time of a project is spent on writing and debugging Fortran programs [1].

The ideal tool for modeling and analysis in scientific computing should eliminate these low-level problems and allow the designer to concentrate on the modeling aspects. Note that we are not claiming the ability to eliminate the iteration in the development process, but with good tools each iteration cycle will be much quicker and the risk of introducing errors will be smaller. Some of the properties of a good programming environment for modeling and analysis in scientific computing are:

1. The user works at a high level of abstraction.
2. Modeling is done using formulate and equations, with good structuring support (for instance object-oriented techniques).
3. Support for symbolic computation is provided. Examples are symbolic simplification of equations and automatic symbolic transformation between different coordinate systems.
4. The environment should provide support for numerical analysis, in particular generation of code for parallel computers.
5. The environment should support changes in the model. A new iteration in the development cycle should be as painless as possible.

Symbolic computation capabilities, like those provided by computer algebra systems [2], are es-

sential in a high-level programming environment for scientific computing. By using a computer algebra system for the symbolic transformations that traditionally were done by hand, one can avoid a lot of tedious labor and reduce the risk of introducing errors. Existing computer algebra systems such as Macsyma [3], Reduce [4], Maple [5], or Mathematica [6] can be a very useful part of a programming environment for scientific computing, but they are not enough on their own. In particular, structuring support for complex models is too weak. Better support for combined symbolic/numerical computation is also needed. The lack of support for structuring of complex models is also the main reason why systems like ALPAL [7] (which utilizes symbolic computation to generate a numerical program for solving a system of partial differential equations) are unsuitable for the kind of problems we are aiming at.

When working with the ObjectMath system the mathematical model is expressed directly in the ObjectMath language. Object-oriented techniques provide a way of structuring the mathematical model and facility reuse. The necessary symbolic simplifications and transformations are then done with computer support instead of with pen and paper. Finally the ObjectMath code generator can be used for generating numerical code, thus obviating the need to program in low-level procedural languages such as Fortran.

3 THE OBJECTMATH LANGUAGE

The ObjectMath programming environment 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, whereas the use of object-oriented modeling makes it possible to structure the model in a natural way.

We have chosen to use an existing computer algebra language, Mathematica, as a basis for ObjectMath. One advantage of this approach is that users who are familiar with the widespread Mathematica system can learn ObjectMath easily. The relationship between Mathematica and ObjectMath can be compared with that between C and C++. The C++ [8] 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 operations on state, in contrast to object-oriented programming languages such as C++. The ObjectMath language has been, and still is, evolving based on feedback from users.

The ObjectMath programming environment is designed to be easy to use for application engineers, e.g., in mechanical analysis, who are not computer scientists. It is interactive and includes a graphical browser for viewing and editing inheritance hierarchies. Other parts of the environment support routines for generation of numerical code and visualization. A class library containing general classes is also available. Viklund *et al.* give an overview of the ObjectMath programming environment and its implementation [9].

3.1 Object-Oriented Modeling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other software [10]. One way to handle this complexity is to use object-oriented techniques. Wegner [11] defines the basic terminology of object-oriented programming:

1. 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 operations of the object.
2. Classes are templates from which objects can be created.
3. 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 programming languages like C++ or Simula [12], where nonlocal access to instance variables is allowed.

More importantly, although 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 that implicitly define the behavior

of some class of physical objects or the relationships between objects. Functions should be side-effect free and regarded as mathematical functions rather than operations. Explicit operations on state may or may not 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 variables are explicitly defined in terms of other variables. If the solution process is numerical, it will compute new state variables from old variable values, and thus perform operations on the variables.

Below we define the basic terminology of object-oriented mathematical modeling:

1. An object is a collection of equations, mathematical functions, and operations that are related to a common abstraction and may share a state.
2. Classes are templates from which objects can be created.
3. 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. Two advantages of object orientation are:

1. It provides a way for grouping equations, functions, and operations.
2. It allows us to reuse equations, functions, and operations by means of inheritance.

To illustrate these ideas we will use 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 element, it is necessary 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, are modeled as separate objects. Properties of objects like these might include a surface description, a normal to the surface, forces and moments on the body, and a volume. These objects might define operations such as finding all contacts on the body, computing the forces on the body or its displacement, and plotting a three-dimensional picture of the body.

Abstract concepts can also be modeled as objects. Examples of such concepts are coordinate systems and contacts between bodies. The coordinate system objects included in the ObjectMath class library define methods for transforming points and vectors to other coordinate systems. Equations and formulae describing the interaction between different bodies are often the most complicated part of problems in machine element analysis. This makes it practical to encapsulate these equations in separate contact objects. One advantage of using contact objects is that we can substitute one mathematical contact model for another simply by plugging in a different kind of contact object. The rest of the model remains completely unchanged. When using such a model in practice, one often needs to experiment with different contact models to find one that 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 facilities reuse of equations and formulae. For example, a cylindrical roller element can inherit basic properties and operations from an exisiting 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 affect all objects that inherit from that class. Multiple inheritance facilitates the maintenance and construction of classes that need to combine different orthogonal properties.

The part-of relation is important for modeling objects that 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 are 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 that are instances of the same class, a situation that occurs frequently.

One way to treat encapsulation is to make the instance variables of an object accessible only to the operations of the object itself. For instance, in Smalltalk-80 [13] operations are always accessible from outside the object whereas 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.

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

Note that 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. Also, there have been no requests for such features from the current industrial users of ObjectMath. However, variable-sized sets of objects are 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.

3.2 ObjectMath Classes and Instances

In this section we use a number of small examples to explain ObjectMath language constructs such as *CLASS*, *INSTANCE*, and *PART* and their use to express inheritance, composition, and object creation. A formal definition of the syntax can be found in the Appendix 1. Here we focus on the object-oriented parts of the ObjectMath language. The rest of the language, i.e., the Mathematica subset, is described in detail in [6].

A *CLASS* declaration declares a class that 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, as mentioned above.

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 singleton instances compact. A similar mechanism exists in the BETA programming language [14]. 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 and equations. Mathematica syntax is used for these.

As an example of a simple ObjectMath class we consider a class `Coordinate System` that models a static coordinate system that may be rotated and translated in an arbitrary way:

```
CLASS CoordinateSystem(Reference, A, R) INHERITS AbstractCoordinateSystem
    InverseA := Inverse[A];
    FromGlobal := Append[Reference'FromGlobal, this];
END CoordinateSystem;
```

This class inherits from the class AbstractCoordinateSystem that defines transformation operations. Parameters to this class are the reference coordinate system relative to which this coordinate system is defined (Reference), the rotation matrix A, and the translation vector R. The parameters are specified when instantiating an object or inheriting from the class. InverseA is an instance variable that denotes the inverse of the rotation matrix. It is defined in the most general way, by calling a function for calculating the inverse of a matrix. FromGlobal denotes a list of coordinate system objects that form a path from the global system to this system. Notice the use of the reserved word this which denotes the object itself.

A coordinate system defined by three successive rotations around the axes, with no translation, can be represented by an instance of the specialized class TranslatedCoordinateSystem:

```
CLASS TranslatedCoordinateSystem(Reference, R)
  INHERITS CoordinateSystem(Reference, IdentityMatrix[3], R)
    InverseA := IdentityMatrix[3];
END TranslatedCoordinateSystem;
```

All basic definitions such as A, R, etc., are inherited from CoordinateSystem. The rotation matrix and its inverse are simply identity matrices. The general definition of InverseA can thus be replaced with a specialized version that is computationally more efficient and stable.

A set containing an undetermined 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 a number of nearly identical objects, e.g., the rolling elements in a rolling bearing. To represent differences between such objects, functions (methods) that are dependent on the index of the instance can be used. The implementation makes it possible to do symbolic computations where the number of elements in the set is left unspecified.

Composition is expressed with *PART* declarations. These are similar to *INSTANCE* declarations but are located inside a *CLASS* or *INSTANCE* declaration. The effect of a *PART* declaration is to create objects inside other objects. The class ThreeSegBody, taken from a model of a rolling bearing, exemplifies this. It consists of three objects of the class RotationSeg name sR1, sC, and sL1, respectively. Note that a *PART* declaration cannot add new definitions to the class inherited from. Instead an extra subclass would have to be introduced should this be desired. The reason for this limitation is that we think it keeps the models cleaner and encourages reuse.

```
CLASS ThreeSegBody(cRef)
  INHERITS GeomBody(cRef)
  PART sR1 INHERITS RotationSeg(cg);
  PART sC INHERITS RotationSeg(cg);
  PART sL1 INHERITS RotationSeg(cg);
  ...
END ThreeSegBody;
```

A *CLASS* or *INSTANCE* declaration can list several parents as multiple inheritance is allowed. An example of this is the following declaration of a set of instances, bW, which model the rollers in a bearing. It inherits from both the class ThreeSegRoller that defines the geometry of a roller consisting of three segments and from the class DynRoller that defines the dynamic behavior of a roller.

```
INSTANCE bW[j] INHERITS
ThreeSegRoller(cB), DynRoller(cB, cG)
  ...
END bW[j];
```

The inheritance graph is linearized depth first, left to right. In the example above this means that if there are conflicts between inherited definitions because they define the same name, definitions from ThreeSegRoller will override definitions from DynRoller.

There is no language support for enforcing encapsulation in the version of ObjectMath that is in use. However, we are currently implementing a new version of the language that has been extended with several features, among them a scheme for encapsulation. For each operation or instance variable it is possible to specify that it is not accessible at all from outside the object; accessible, but not assignable, from outside the object; and accessible and assignable from outside the object.

This scheme is more fine grained than the ones found in most other object-oriented languages as it separates between access and assignment. It is also possible to specify that the operation or instance variable should be accessible, and possibly assignable, from specific objects but not from other places in the model. A similar feature is found in the Eiffel language [15]. Experiences with modeling in ObjectMath have showed that such a flexible way of specifying encapsulation is desirable in object-oriented mathematical modeling. In particular, it is quite common that a variable should be accessible from several objects but assignable only from one particular object (e.g., for initialization.)

4 A MODEL OF A ROLLING BEARING

In this section 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 that had been previously implemented in C++ by SKF. The intention is that the performance of the generated code should be directly compared with the performance of the handwritten code. The bearing consists of an outer and an inner ring and a single roller. However, the model can easily be extended to include an arbitrary number of rollers. Figure 2 shows the geometry of the bearing. The z-axis is pointing inwards in the picture and the coordinate system

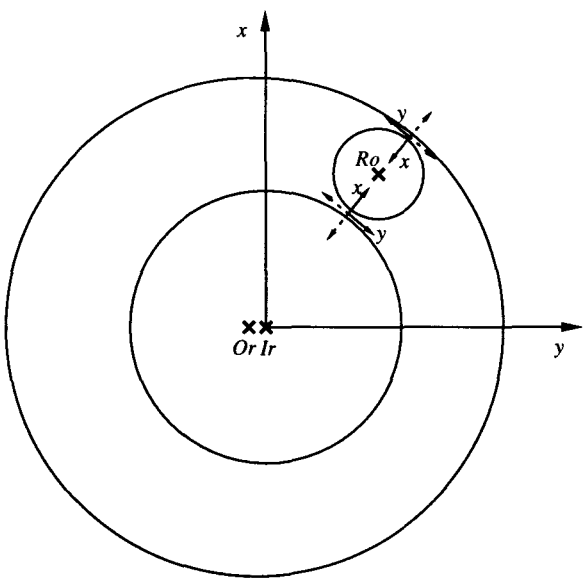

FIGURE 2 Geometry of the bearing.

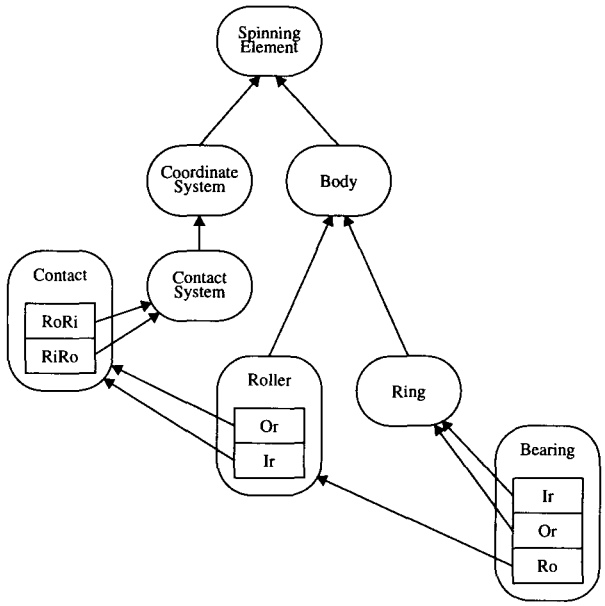

FIGURE 3 Classes in the bearing model.

is right oriented. Angles are measured clockwise from the x-axis.

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 that can be described by a system of ordinary differential equations. The class hierarchy of the model is shown in Figure 3.

As shown in Figure 3 we use the same graphical representation as in the graphical editor in the ObjectMath programming environment. The arrows represent 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 2 the same structure is shown using Coad and Yourdons OOA notation.

As can be seen in Figure 3, the object that 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 Figure 3 is that it makes it somewhat easier to extend the model to include an arbitrary number of rollers.

The abstract class `SpinningElement` models an object (not necessarily a physical body) that ro-

tates. This class introduces two instance variables: Origin and Turn that denote the origin and rotation of the object relative to the global coordinate system. The actual definitions of these are given in subclasses inherited from SpinningElement (cf. virtual member functions in C++ or deferred routines in Eiffel).

```
CLASS SpinningElement
  Origin;
  Turn;
END SpinningElement;
```

A local coordinate system is modeled by the class CoordinateSystem that inherits from SpinningElement. 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.

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 for which the calculations are done as well as body2, 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 on the following page.

```
CLASS CoordinateSystem(t) INHERITS SpinningElement
  (* 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;
```

```
CLASS ContactSystem(body1, body2, contact, t) INHERITS CoordinateSystem(t)
  (* 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)
  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 that 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 righthand sides of the ordinary differential equations for the body. As usual, the class parameter t denotes the time.

```
CLASS Body(t) INHERITS SpinningElement
  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 sinus wave is added to the base radius to model imperfections in a real ring.

```
CLASS Ring(contact, t) INHERITS Body(t)
  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)
  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
  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
  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 that 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 that 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.

5 NUMERICAL COMPUTATION WITH OBJECTMATH

Analyzing a mathematical model expressed in ObjectMath also involves performing numerical computations. The Mathematica system can be used for some of these calcualtions. However, there are problems with this approach. Mathematica code is interpreted and cannot be executed as efficiently as programs written in compiled languages such as Fortran, C, or C++. This is a serious drawback, particularly when doing mostly numerical computations in realistic applications. We also want to be able to use existing, highly optimized, special-purpose numerical routines. Thus, an important tool in the ObjectMath programming environment is a code generator that generates numerical code from ObjectMath models. For a typical application, symbolic computation is heavily used to rewrite and simplify equations before code is generated (cf. Fig. 1).

The user interface of the code generator consists of a number of ObjectMath routines that can be called either from an ObjectMath model or interactively. These routines can be divided into two groups: (1) the function level interface that provides routines for generating code from variables and functions and (2) the system level interface that provides routines for generating code from whole systems of equations.

The function level interface (described in Section 5.1) is currently being used for industrial applications whereas the system level interface (described in Section 5.2) is still under development within our group.

Currently C++ is used as the target language by the function level routines and Fortran by system level routines. We are working on combining and generalizing the two parts of the code generator so that either of them can generate both C++ and Fortran.

5.1 Code Generation from ObjectMath Functions

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 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 that 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 because 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 3. The variable `B't` is a scalar (denoting time) whereas both `B'Ro'R` and `B'Ro'Fi` are functions that 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 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.

There are actually two different routines for generating code for functions. The first one translates an ObjectMath function definition into a corresponding function in the target language. The second one evaluates the ObjectMath function symbolically and then generates a function that computes the resulting expression. With this functionality, symbolic computation can be used to synthesize parts of the numerical code. Users have fine grain control over this process as they can use certain ObjectMath functions to determine which parts of the expressions should be evaluated.

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.

Both routines mentioned above have the limitation that the ObjectMath function body or expression (after evaluation) must only consist of a certain language subset. This subset includes:

1. Those of the operators that correspond to C++ operators. The arguments to these functions might be both scalars and vectors or matrices, as the C++ classes we use overload most operators.
2. Functions that have appropriate definitions in the standard mathematical library, e.g., trigonometric functions and logarithms.
3. Common functions that operate on vectors and matrices, e.g., dot product, transposition, and inverse.
4. Most control structures, e.g., `If`, `While`, and `For` expressions.
5. Function calls, including functions with multiple return values.

However, there are many built-in ObjectMath functions that are not included in the allowed language subset, most notable all functions that do symbolic computations. This limitation might be slightly troublesome as it is not always obvious which operators will be present after evaluating an expression syumbolically. The code generator is now and then being extended to handle a larger language subset when this becomes necessary.

5.2 Code Generation from Systems of ObjectMath Equations

Although the function level interface to the code generator can be used for generating large parts of an application, it is usually necessary to handwrite some glue code that implements the data flow between the generated functions and the chosen solver. Also, the function level code generation routines generate sequential code as there usually are few possibilities for parallelization on this level. (Of course there might be possibilities to exploit, for instance, vector parallelism within a function, but such parallelizations are performed by the compiler used for compiling the generated code, not by the ObjectMath code generator.) To produce a parallel program the user must usually introduce the parallelism in the handwritten glue code, e.g., by writing code that calls several of the generated functions in parallel. Obviously, this makes the glue code much more complex.

The routines supplied by system level interface of the code generator can generate a complete numerical simulation program from a system of ObjectMath equations. This process is depicted in Figure 4. Apart from freeing the user from implementing the necessary glue code, this higher level of abstraction opens up many possibilities for the code generator to perform automatic parallelizations. This is an important property as one of the main problems in extracting parallelism from application programs written in languages such as Fortran is the low level of abstraction.

Sometimes it is not convenient to express the whole problem to be solved as a system of equations. In these cases the two interfaces to the code

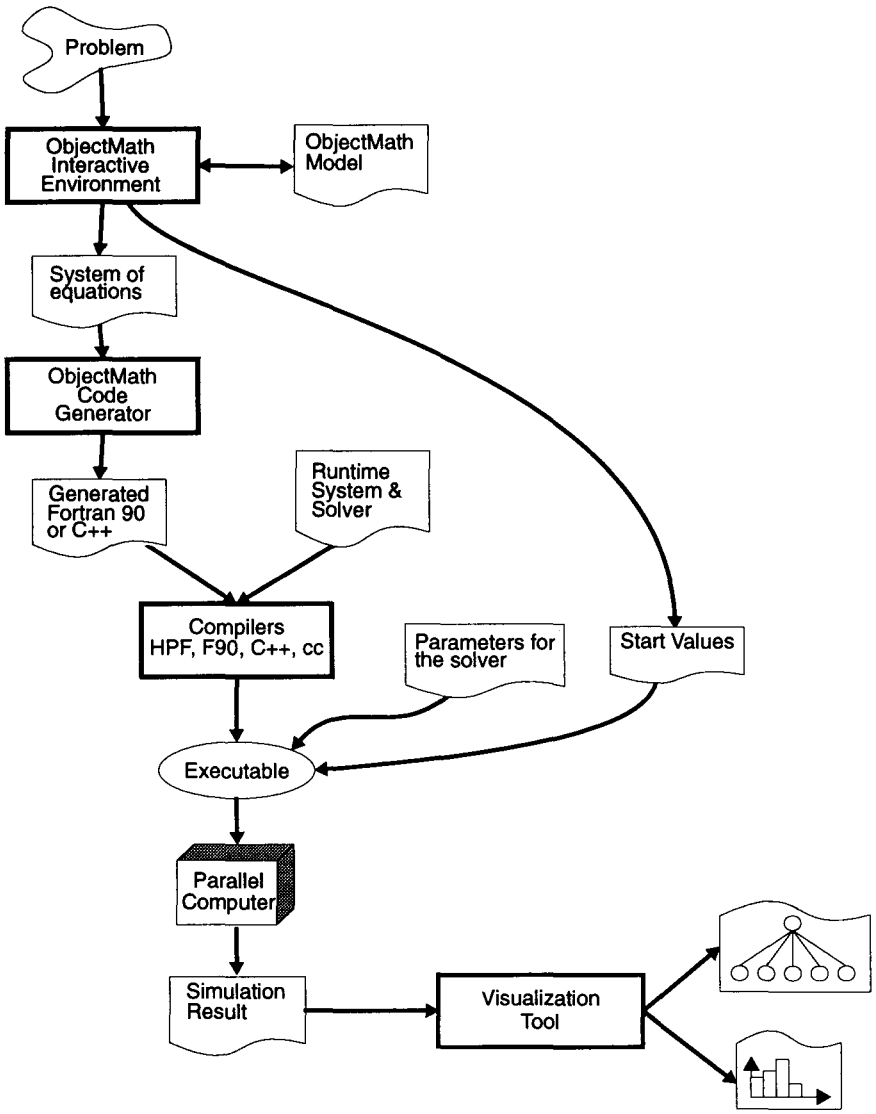

FIGURE 4 Generating a numerical simulation program with ObjectMath.

generator can be combined. The equations given as input to system level routines may contain function calls that are not evaluated before code generation. Instead the function level routines are used to generate code from these functions. Thus, pure mathematical equations can be combined with functions that are more naturally (or more efficiently) expressed imperatively.

The ObjectMath code generator generates parallel programs from systems of ordinary differential equations. In the future it will be extended to handle other classes of problems. A standard (serial) ODE solver is used and the righthand sides of the equations are evaluated in parallel. A simple supervisor–worker scheme is currently used to schedule the computation of the tasks (see Fig. 5) and some communcation analysis is needed to find out which data should be distributed. The generated code is linked with a small run-time system that contains special-purpose communication routines and a dynamic scheduler that schedules the tasks on the available processors.

Andersson and Fritzson [16] discuss parallel code generation from ObjectMath models in more detail and also give some performance measurements.

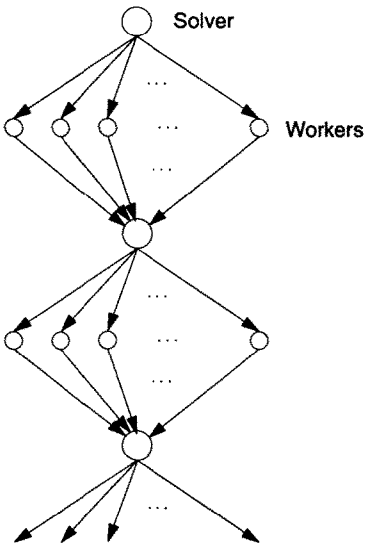

FIGURE 5 The supervisor–worker model of task scheduling.

6 STATE OF THE IMPLEMENTATION

The ObjectMath programming environment currently runs on Sun workstations (running Solaris 1 or 2) under the X window system. It includes a graphical browser for viewing and editing inheritance and composition hierarchies, the Gnu Emacs editor for editing ObjectMath equations and formulae, as well as the ObjectMath compiler and the code generator discussed in the previous section. Figure 6 shows the overall structure of the environment. The compiler generates Mathematica code, which is sent to the Mathematica system for symbolic computation.

Version 3 of the ObjectMath environment is in regular use on real problems at SKF ERC. It is also used for experiments at a few academic sites. The code generator in this version does not have a system level interface and only generates serial C++ code. Version 4 of the system, which features a complete rewrite of the compiler and the new code generator that generates parallel code, is currently in use within our group and will soon be released to external users.

7 RELATED WORK

Object-oriented modeling has been used in a number of different application areas, for instance, control theory [17], chemistry [18], biology [19], and mathematical modeling in general [20]. This is not surprising as object orientation appears to be a natural approach for modeling the world.

The object-oriented language constructs found in ObjectMath are fairly conventional with the exception of the proposed scheme for specifying encapsulation that seems rather novel. There are potential problems associated with using simple linearization to handle conflicts that arise from multiple inheritance. Some of these problems could probably be solved with more sophisticated rules for prioritizing the parent classes, e.g., the partial ordering used in CLOS [21]. However, in our experience conflicts of this kind seldom occur in practice. Thus, it seems unnecessary to add extra complexity to the language just to handle them.

A number of computer algebra systems with object-oriented features have been developed during recent years. One of the best known is the AXIOM system [22], a descendant of SCRATCHPAD [23], which has a type system that in some sense is object-oriented, even though the language constructs provided are different from the ones usually found in object-oriented languages. Another example is the Mathematica-inspired system *AlgBench* [24], which extends the pattern matching of Mathematica to inheritance-based unification. All of these systems (except ObjectMath) focus on using object-oriented techniques for implementing symbolic and algebraic routines, i.e., object-oriented programming rather than on object-oriented modeling.

An example of a system that aims at solving the same problem as the ObjectMath programming environment, but with a somewhat different approach, is the Sinapse program synthesis system [25]. The work on Sinapse emphasizes automation based on domain-specific knowledge. The user might describe the problem to be solved as a symbolic model using application-specific keywords. The symbolic model is at an even higher level of abstraction than a mathematical model for

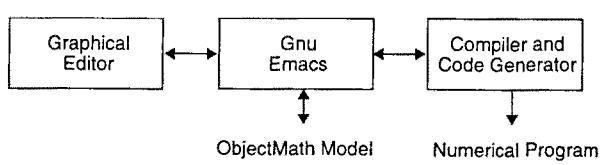

FIGURE 6 Structure of the ObjectMath environment.

the problem. Obviously, this is only possible for the rather limited domain which the system has enough knowledge about. The user may also supply a mathematical model, but there seems to be little structuring support for such models, even though the knowledge base of the system is organized with object-oriented techniques.

In the ObjectMath project we take the view that user interaction is acceptable in all steps in the software development cycle, if it is necessary to produce efficient programs. Thus, we have been able to build a general system that can handle problems from different application domains, but produces an executable program in a semiautomatic rather than a fully automatic fashion. Sinapse is implemented in Mathematica and includes a code generation package called MathCode [26], which seems quite similar to the function level part of the ObjectMath code generator.

Other related systems are object-oriented simulation languages and systems such as Dymola [27] and Omola [28]. These systems focus on automatic transformation of systems of equations without user interaction. Our view is that although such automatic transformations certainly are very useful, the possibility to manually specify transformation that ObjectMath provides is necessary for efficient solution of complex problems in some application areas (e.g., mechanical analysis.) Neither Dymola nor Omola currently supports generation of parallel code. Another difference is that these modeling languages only allow equations, no functions or operations. Our experience shows that, when working with complex models there are often small parts that have to be expressed procedurally rather than mathematically, something that is possible in ObjectMath. If one needs to do the same thing when working with Dymola, for instance, one has to write the procedural code in an ordinary programming language, compile it separately, and link it to the simulation program. Because of the symbolic transformations done by the Dymola system, the user might also be forced to supply another routine that calculates the inverse of the external routine. Obviously this might be very complicated to do efficiently in some cases.

The SimLab environment [29] is another system that is quite similar to Dymola and Omola. However, the SimLab modeling language is not object-oriented. Many of the ideas behind the SimLab approach seem to be very similar to the ideas behind ObjectMath, even though there are a number of differences between the systems. One is obviously that the SimLab language is not object-oriented, another is that SimLab, just as Dymola and Omola, focuses on totally automatic transformations. A successor to SimLab is Chains [30], a language for programming with algebraic-topology mathematical objects. It is interesting to note that the Chains implementation uses Mathematica for symbolic computations.

8 CONCLUSIONS

There is a strong need for good high-level tools for program development in scientific computing. Experience from using ObjectMath in an industrial environment shows that the ObjectMath system is a useful tool that can satisfy part of this need. So far we have mainly focused on the modeling phase of the development process for scientific software. In our experience, the combination of the object-oriented paradigm and mathematical modeling is suitable for this kind of modeling. The semantic gap between the system being modeled and the ObjectMath model is small. This results in models that are relatively easy to develop, well structured, and understandable for application experts, even if they are not also ObjectMath experts. The ObjectMath language has evolved based on feedback from users. For instance, multiple inheritance and composition were added to the language because we discovered that these features were necessary to model complex systems in a systematic way.

Symbolic computation appears to be an essential capability in high-level systems for scientific computing. Whole problems can almost never be solved symbolically, but support for simplification and automatic transformations are very time saving and can improve the quality of the produced software as the probability of introducing errors is far less than if such calculations are done by hand. We have also seen that the complexity of realistic models in mechanical analysis makes it essential that the system allows the user to supply extra information to guide the analysis. Even if it is sometimes possible to generate numerical programs totally automatically, it is desirable to take advantage of the extensive application domain knowledge of the engineer using the system. Thus, advanced problems can be solved more efficiently. Once the extra solution and transformation steps have been supplied, the transformation from high-level model down to executable numerical code is automatic. Although the ObjectMath system already includes some support for numeri-

cal implementation and visualization, much work remains to be done in these areas.

REFERENCES

[1] P. Fritzson and D. Fritzson, "The need for high-level programming support in scientific computing applied to mechanical analysis," *Comput. Structures*, vol. 45, pp. 387–395, 1992. Also as Tech. Rep. LiTH-IDA-R-91-04, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden.

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

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

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

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

[6] S. Wolfram, *Mathematica—A System for Doing Mathematics by Computer*, 2nd ed. Redwood City: Addison-Wesley, 1991.

[7] G. O. Cook, Jr., "ALPAL, a program to generate physics simulation codes from natural descriptions," *Int. J. Mod. Phys.*, vol. 1, pp. 1–51, 1990.

[8] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Reading: Addison-Wesley, 1990.

[9] L. Viklund, J. Herber, and P. Fritzson, "The implementation of ObjectMath—a high-level programming environment for scientific computing," in *Compiler Construction—4th International Conference, CC '92*, vol. 641 of *Lecture Notes in Computer Science*, U. Kastens and P. Pfahler, Eds. New York: Springer-Verlag, 1992, pp. 312–318.

[10] G. Booch, *Object Oriented Design with Applications*. Redwood City: Benjamin/Cummings, 1991.

[11] P. Wegner, "Concepts and paradigms of object-oriented programming," *OOPS Messenger*, vol. 1, pp. 87, Aug. 1990.

[12] SIS, Box 3295, Stockholm, Sweden, *SIMULA Standard, 1987*. Swedish Standard SS 63 61 14. ISBN 91-7162-234-9.

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

[14] B. B. Kristensen, O. L. Madsen, B. Möller-Pedersen, and K. Nygaard, "The BETA programming language," in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. Boston, MA: MIT, 1987, pp. 7–48.

[15] B. Meyer, *Eiffel: The Language*. New York: Prentice Hall, 1992.

[16] N. Andersson and P. Fritzson, "Generating parallel code from object oriented mathematical models, in *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice on Parallel Programming*, July 1995, pp. 48–57.

[17] 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 *Proc. of the IFAC Symposium Computer Aided Design in Control Systems*. July 1991, p. 1.

[18] 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, *Comput. Chem. Eng.*, vol. 12, pp. 53–72, 1991.

[19] C. Pierret-Golbreich, "Object-centered knowledge representation for modelling in biology," in *International Symposium on AI, Expert Systems and Languages in Modeling and Simulation*, June 1987, p. 281.

[20] T. W. Page, Jr., S. E. Berson, W. C. Cheng, and R. R. Muntz, "An object-oriented modeling environment," in *OOPSLA'89 Conference Proc.*, 1989, p. 287.

[21] D. G. Bobrow, L. G. DeMichiel, R. P. G. S. E. Keene, G. Kiczales, and D. A. Moon, "Common Lisp object system specification," ANSI X3J13 Document 88-002R, American Standards Institute, Washington, DC, June 1988.

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

[23] R. D. Jenks, R. S. Sutor, and S. M. Watt, "Scratchpad II: An abstract datatype system for mathematical computation," in *Mathematical Aspects of Scientific Software*, vol. 14 of *The IMA Volumes in Mathematics and Its Applications*, J. R. Rice, Ed. New York: Springer-Verlag, 1988, pp. 157–182.

[24] G. Grivas and R. E. Maeder, Matching and unification for the object-oriented symbolic computation system AlgBench, in *Design and Implementation of Symbolic Computation Systems*, vol. 722 of *Lecture Notes in Computer Science*, A. Miola, Ed. New York: Springer-Verlag, 1993, pp. 164–176.

[25] E. Kant, "Synthesis of mathematical modeling software, "*IEEE Software*, vol. 10, pp. 30–41, May 1993.

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

[27] H. Elmqvist, "Object-oriented modeling and automatic formula manipulation in Dymola, in *SIMS'93, Applied Simulation in Industry—Proc.*

of the 35th SIMS Simulation Conference, June 1993.

[28] S. E. Mattsson, M. Andersson, and K. J. Åström, "Object-oriented modelling and simulation," in *CAD for Control Systems*, D. A. Linkens, Ed. New York: Marcel Dekker, 1993, pp. 31–69.

[29] R. S. Palmer and J. F. Cremer, "SimLab: Automatically creating physical systems simulators," Cornell University, Ithaca, NY, Computer Science Tech. Rep. TR92-1246, 1992.

[30] R. S. Palmer, "Chain models and finite element analysis," Cornell University, Ithaca, NY, Computer Science Tech. Rep. TR94-1406, 1994.

[31] J. Heering, P. R. K. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF—reference manual," *ACM SIGPLAN Notices*, vol. 14, pp. 43–75, Nov. 1989.

[32] P. Coad and E. Yourdon, *Object-Oriented Analysis*, New York: Prentice Hall, 1991.

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

APPENDIX 1: OBJECTMATH SYNTAX

The syntax of the ObjectMath language is defined using the syntax definition formalism SDF [31]. The `sorts` sections of the specification declare the names of domains or nonterminals used in the other sections. In the `lexical syntax` section the rules of the lexical syntax are given, whereas the `context-free syntax` section defines the concrete and abstract syntax. The SDF context-free rule (function):

```
"CLASS" CLASS_NAME INHERITS
                          -> CLASS_HEAD
```

corresponds to the following BNF rule:

```
<CLASS_HEAD> ::= "CLASS"
                 <CLASS_NAME> <INHERITS>
```

Finally, in the `priorities` section relations between rules in the context-free syntax are defined as well as the associativity of groups of different operators.

SDF Specification of the ObjectMath Syntax

```
%% ObjectMath

exports

 sorts
    MODEL MODEL_HEAD MODEL_BODY PACKAGES
    GLOBAL_DECL OBJ_DECL CLASS_DECL INST_DECL
    CLASS_HEAD CLASS_NAME CLASS_END INST_HEAD
    INST_NAME INST_END INHERITS OBJ_BODY
    COMP_EXPR EXPR EXPR1 LIST QUOTE VAR
    PAT_TEST BLANK PATTERN TAG SLOT SYMB
    FILE_NAME INT REAL STRING DIGIT
    STRING_ELEM COMMENT_ELEM

 lexical syntax
    [ \t\n\r]                          -> LAYOUT
    "(*" COMMENT_ELEM* "*)"            -> LAYOUT
    ~[*]                               -> COMMENT_ELEM
    "*" ~[)]                           -> COMMENT_ELEM
    [0-9]                              -> DIGIT
    DIGIT+                             -> INT
    DIGIT* "." DIGIT+                  -> REAL
    DIGIT+ "." DIGIT*                  -> REAL
    ~[\"]                              -> STRING_ELEM
    "\\\""                             -> STRING_ELEM
    "\"" STRING_ELEM* "\""             -> STRING
    [a-zA-Z'$] [a-zA-Z'$0-9]*          -> SYMB
    ['']*                              -> QUOTE
```

```
    SYMB BLANK                          -> PATTERN
    BLANK                               -> PATTERN
    "_"                                 -> BLANK
    "_."                                -> BLANK
    "__"                                -> BLANK
    "___"                               -> BLANK
    "#"                                 -> SLOT
    "#" INT                             -> SLOT
    "##"                                -> SLOT
    "##" INT                            -> SLOT
    "::" SYMB                           -> TAG
    "::" STRING                         -> TAG

context-free syntax
   MODEL_HEAD MODEL_BODY                -> MODEL
   "MODEL" SYMB ";"                     -> MODEL_HEAD
   PACKAGES GLOBAL_DECL OBJ_DECL*       -> MODEL_BODY
   "PACKAGES" { STRING "," }+ ";"       -> PACKAGES
                                        -> PACKAGES
   COMP_EXPR                            -> GLOBAL_DECL
   CLASS_DECL                           -> OBJ_DECL
   INST_DECL                            -> OBJ_DECL
   CLASS_HEAD OBJ_BODY CLASS_END        -> CLASS_DECL
   "CLASS" CLASS_NAME INHERITS          -> CLASS_HEAD
   SYMB "(" { SYMB "," }+ ")"           -> CLASS_NAME
   SYMB                                 -> CLASS_NAME
   "END" SYMB ";"                       -> CLASS_END
   "END" ";"                            -> CLASS_END
   INST_HEAD OBJ_BODY INST_END          -> INST_DECL
   "INSTANCE" INST_NAME INHERITS        -> INST_HEAD
   SYMB "[" SYMB "]"                    -> INST_NAME
   SYMB                                 -> INST_NAME
   "END" INST_NAME ";"                  -> INST_END
   "END" ";"                            -> INST_END
   "INHERITS" SYMB "(" LIST ")"         -> INHERITS
   "INHERITS" SYMB                      -> INHERITS
                                        -> INHERITS
   COMP_EXPR                            -> OBJ_BODY
   { EXPR ";" }+                        -> COMP_EXPR
   { EXPR ";" }+ ";"                    -> COMP_EXPR
                                        -> COMP_EXPR
   EXPR ">>" FILE_NAME                  -> EXPR
   EXPR ">>>" FILE_NAME                 -> EXPR
   EXPR "=" EXPR                        -> EXPR    {right}
   EXPR ":=" EXPR                       -> EXPR    {right}
   EXPR "^=" EXPR                       -> EXPR    {right}
   EXPR "^:=" EXPR                      -> EXPR    {right}
   SYMB "/:" EXPR "=" EXPR              -> EXPR
   SYMB "/:" EXPR ":=" EXPR             -> EXPR
   EXPR "=."                            -> EXPR
   SYMB "/:" EXPR "=."                  -> EXPR
   SYMB "::=" SYMB                      -> EXPR
   STRING "::=" SYMB                    -> EXPR
   SYMB "::=."                          -> EXPR
```

```
STRING ":=."                          -> EXPR
EXPR "//" EXPR                        -> EXPR    {left}
EXPR "&"                              -> EXPR
EXPR "+=" EXPR                        -> EXPR    {right}
EXPR "-=" EXPR                        -> EXPR    {right}
EXPR "*=" EXPR                        -> EXPR    {right}
EXPR "/=" EXPR                        -> EXPR    {right}
EXPR "/." EXPR                        -> EXPR    {left}
EXPR "//." EXPR                       -> EXPR    {left}
EXPR "->" EXPR                        -> EXPR    {right}
EXPR ":>" EXPR                        -> EXPR    {right}
EXPR "/;" EXPR                        -> EXPR    {left}
SYMB ":" EXPR                         -> PATTERN
EXPR ".."                             -> EXPR
EXPR "..."                            -> EXPR
EXPR "||"                             -> EXPR    {left}
EXPR "&&" EXPR                        -> EXPR    {left}
"!" EXPR                              -> EXPR
EXPR "===" EXPR                       -> EXPR    {left}
EXPR "=!=" EXPR                       -> EXPR    {left}
EXPR "==" EXPR                        -> EXPR    {left}
EXPR "!=" EXPR                        -> EXPR    {left}
EXPR ">" EXPR                         -> EXPR    {left}
EXPR "<" EXPR                         -> EXPR    {left}
EXPR ">=" EXPR                        -> EXPR    {left}
EXPR "<=" EXPR                        -> EXPR    {left}
EXPR "+" EXPR                         -> EXPR    {left}
EXPR "-" EXPR                         -> EXPR    {left}
EXPR "*" EXPR                         -> EXPR    {left}
EXPR "/" EXPR                         -> EXPR    {left}
"+" EXPR                              -> EXPR
"-" EXPR                              -> EXPR
EXPR EXPR                             -> EXPR    {left}
EXPR "^" EXPR                         -> EXPR    {right}
EXPR "." EXPR                         -> EXPR    {left}
EXPR "**" EXPR                        -> EXPR    {left}
EXPR "!" EXPR                         -> EXPR
EXPR "/@" EXPR                        -> EXPR    {right}
EXPR "//@" EXPR                       -> EXPR    {right}
EXPR "@@"                             -> EXPR    {right}
EXPR "~" EXPR1 "~" EXPR1              -> EXPR
EXPR1                                 -> EXPR
EXPR1 "@" EXPR1                       -> EXPR1   {right}
EXPR1 "++"                            -> EXPR1
EXPR1 "--"                            -> EXPR1
"++" EXPR1                            -> EXPR1
"--" EXPR1                            -> EXPR1
QUOTE                                 -> EXPR1
EXPR1 "[" LIST "]"                    -> EXPR1
EXPR1 "[[" LIST "]]"                  -> EXPR1
VAR                                   -> EXPR1
VAR "?" VAR                           -> VAR
SYMB                                  -> VAR
PATTERN                               -> VAR
```

```
    SYMB TAG                                -> VAR
    INT                                     -> VAR
    REAL                                    -> VAR
    "(" LIST ")"                            -> VAR
    "{" LIST "}"                            -> VAR
    SLOT                                    -> VAR
    STRING                                  -> VAR
    { COMP_EXPR "," }*                      -> LIST
    STRING                                  -> FILE_NAME
    SYMB                                    -> FILE_NAME

priorities
    "(" LIST ")" -> VAR <
    {"CLASS" CLASS_NAME INHERITS ->   CLASS_HEAD,
     "INSTANCE" INST_NAME INHERITS -> INST_HEAD}

priorities
    { right: "=", ":=", "^=", "^:=" } <
    "//" <
    EXPR "&" -> EXPR <
    { right: "+=", "-=", "*=", "/=" } <
    { left:  "/.",  "//." } <
    { right: "->", ":>" } <
    "/;" <
    SYMB ":" EXPR -> PATTERN <
    { EXPR ".." -> EXPR, EXPR "..." -> EXPR } <
    "||" <
    "&&" <
    "!" EXPR -> EXPR <
    { left: "===", "=!=" } <
    { left: "==", "!=", ">", "<", ">=", "<=" } <
    { left: EXPR "+" EXPR -> EXPR,
            EXPR "-" EXPR -> EXPR } <
    { left: EXPR "*" EXPR -> EXPR,
            EXPR "/" EXPR -> EXPR } <
    "+" EXPR -> EXPR <
    "-" EXPR -> EXPR <
    EXPR EXPR -> EXPR <
    "." <
    "**" <
    EXPR "!" -> EXPR <
    EXPR "!!" -> EXPR <
    { right: "/@", "//@", "@@" }

priorities
    "@" <
    EXPR1 "++" -> EXPR1 <
    EXPR1 "--" -> EXPR1 <
    "++" EXPR1 -> EXPR1 <
    "--" EXPR1 -> EXPR1 <
    EXPR1 "[" LIST "]" -> EXPR1 <
    EXPR1 "[[" LIST "]]" -> EXPR1 <
    EXPR1 QUOTE -> EXPR1
```

APPENDIX 2: OOA DIAGRAM OF THE BEARING MODEL

This is the same class structure as in Figure 3, drawn using the OOA notation by Coad and Yourdon [32]. 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 [33], take the same view as ObjectMath on this issue.