Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

COMPUTATIONAL MECHANICS SOLVERS BASED ON OBJECT-ORIENTED DESIGN PRINCIPLES

Joseph Whitesell

Department of Mechanical Engineering
Michigan State University
East Lansing, Michigan 48824

John D. Reid

Engineering Building
General Motors Corporation
Warren, Michigan 48090

ABSTRACT

In this paper the methods of object-oriented programming are compared with the software needs of the field of computational mechanics and then accessed. It appears that object-oriented programming techniques can enable computational mechanics software structures with desirable characteristics not easily obtained within a conventional software perspective: (1) they can function more easily with different coordinate types; (2) they facilitate the construction of larger system models from proven (relocatable) sub-system models, and (3) they easily support multi-resolution computational processes.

1. INTRODUCTION

In recent years significant progress has been made toward the development of solvers for analyzing large and complex mechanical systems composed of several interconnected rigid or flexible bodies [Haug 1984]. Typical systems analyzed by these solvers range from relatively simple slider-crank systems to very complex ground or aero-space vehicles. These solvers often integrate several modes of analysis such as static equilibrium analysis, nonlinear dynamical analysis, kinematic analysis, inverse dynamic analysis and linearized dynamical system analysis. Static analysis is used to determine a system equilibrium state. Nonlinear dynamics analysis consists of producing time histories of body positions, velocities, accelerations and reaction forces. Kinematic analysis is used to compute body positions, velocities and accelerations when predetermined motion trajectories are imposed on one or more bodies. Inverse dynamical analysis is used to compute the forces necessary to cause prescribed motion. Linear dynamical analysis is carried out by automatically linearizing a nonlinear dynamics model and applying eigenanalysis methods.

Current trends are toward more advanced modelling capabilities, integration of more modes of analysis and the introduction of multi-disciplinary physical effects such as nonlinear material effects, feedback control elements and electro-mechanical or hydraulic actuators. There have also been efforts to integrate design optimization methods with the technology (e.g., the emerging field of design automation).We can expect, however, if traditional approaches are used to meet the objectives stated above that bulkier and more complex software structures will result and it is likely that reliability, responsiveness to technological change and maintainability will suffer. Recent developments in software engineering, especially object-oriented programming techniques, have been directed toward overcoming these problems by providing new ways of organizing software structures [Dahl et al. 1967; Cox 1986; Goldberg and Robson 1983; Pascoe 1986; Petersen 1987; Stefik and Bobrow 1986].

This paper describes our research into the design of multibody mechanics solvers based on the concepts of object-oriented programming. The results have been encouraging. In each of the examples that we have studied we have concluded that object-oriented programming methods not only provide solutions to the practical software problems mentioned above, but they also open up several new opportunities in the simulation of engineering systems.

The paper is organized as follows: In Section 2, a general statement of the main goals is presented. In Section 3 the approach is described. Section 4 presents conclusions and future directions. In Appendix A a brief overview of object-oriented programming is presented. In Appendix B the body of general computational engineering methods that we use is presented.

2. GOALS

Object-oriented programming is a relatively new idea in software programming that concentrates on the objects of a system rather than on the procedures that manipulate data, as conventional software does (e.g., FORTRAN or PL/I). Object-oriented programming has been compared with the notion of the interchangeable part in manufacturing since programmers produce components (objects) in which procedure (methods) and data (state) are packaged together so as to be reusable. Several practical benefits have been attributed to object-oriented programming. These include improved representational flexibility, improved reliability, reduced sensitivity to disruption by changes thereby promoting enhancements, lower development and maintenance cost, and a seamless transition from systems analysis all the way to the actual code.

Object-oriented programming is described in several sources [Cox 1986; Goldberg and Robson 1983; Pascoe 1986; Petersen 1987; Stefik and Bobrow 1986]. We present only an overview of the basic ideas in Appendix A. Our terminology is consistent with the object-oriented language Smalltalk-80 [Goldberg and Robson 1983].

The general goal of this work is to outline how the methods of object-oriented programming can be applied to the design of computational engineering software structures and show the advantages of the approach. We have selected the area of computational multibody mechanics due to its relatively high level of development, its generality and our familiarity with it (see Appendix B). In our work we have studied a multibody static equilibrium solver and a multibody kinematics solver [Sung 1989]. In a related project an object-oriented bond graph processor has been studied [Reid 1990].

To illustrate the effectiveness of the object-oriented design approach in this paper we describe an approach for achieving three advanced simulation design goals related to computational multibody mechanics software:

2.1 Design Goal 1: Reusable Model Elements

The first goal is to create reusable mathematically based engineering models which allow complex engineering models to be constructed from proven submodels. This is difficult to achieve with conventional software methods since interchangeability and relocatability of software structures are often inhibited by the normal requirement of data type agreements. Object-oriented programming methods overcome these difficulties by providing encapsulation, dynamic binding and polymorphism. In otherwords, since objects respond to messages according to their own internal methods, objects (models) of different types can respond to identical messages with different results. This simplifies the construction of models from encapsulated submodels as will be illustrated below.

2.2 Design Goal 2: Multi-Resolution Simulation

The second goal is to create multi-resolution computational mechanics structures. Multi-resolution computational processes (e.g., multi grid methodology) been extensively studied in connection with solving large computational mechanics problems but their many advantages have been difficult to obtain due to the difficulties associated with implementing them with conventional software structures [Schultz 1981]. Dynamic binding and

polymorphism offer a solution here, as well, since model components can be easily changed at run time.

2.3 Design Goal 3: Coordinate Free Design

One of the sources of complexity in mechanics is due to the various geometric coordinate frames that one can adopt. Indeed the analytic mechanics of Lagrange was a milestone largely because it provided a general statement of the equations of motion that did not depend on a particular geometric point of view (e.g., generalized coordinates and generalized forces). The final goal of this work is to develop computational mechanics software structures which function above the level of geometric detail. This goal is achieved by making the higher levels of the class hierarchy free of specific geometric references, such as geometric dimension, through the use of polymorphism.

3. GENERAL DESIGN APROACH

The designs utilize new classes and related subclasses which are created to support the needs of computational multibody mechanics as outlined in Appendix B. Figure 1 depicts a portion of a one class hierarchy used for representing and simulating multibody systems.

```
Object
  MechanicsObject
      MechanicsElement
          InertialElement
              PointMass
                  RigidBody
                      RigidLink
                          FemLink
      MechanicsPair
          ForcePair
              LinearSpring
              GravitationalForce
              KinematicPair
              RevoluteJoint
              TranslationalJoint
      Node
```

Figure 1. A Class Hierarchy for Multibody Mechanics

The **MechanicsElement class** is used to define the various mechanical components found in multibody mechanics such as point masses, rigid bodies, elastic bodies and springs, dampers and kinematic joints. It captures common behavior between its two subclasses; those being the **InertialElement** class and the **MechanicsPair** class.

The instance variables of members of the class **MechanicsElement** consist of an **elementList** which is a collection of **MechanicsElement** objects , a **pairList** which is a collection of inter-element connecting elements such as kinematic pairs or connecting forces and a **nodeList** which represent points of interest in the given mechanical element and any applied forces which act at the points.

Any instance of the **MechanicsElement** class possesses methods for assembling numerical equilibrium equations in the general forms described in Appendix B or reporting a fault if it cannot do so. This is done by sending appropriate messages to the members of its **elementList** instance variable (e.g. its subcomponents). This allows instances of class **MechanicsElement** to simulate its behavior by directly solving a set of element equilibrium equations. To support this capability the **MechanicsElement** class also possesses methods for reporting the current value of its Jacobian submatrix and its contribution to the system equilibrium vector. This allows any instance of class **MechanicsElement** to become a subcomponent of a larger mechanics element.

A key to meeting each of the goals listed above is that the messages sent to the mechanical elements for assembling the equilibrium equation are designed according to the principle of polymorphism. In otherwords, identical messages are sent to each **MechanicsElement** instance regardless of its specific nature. Thus it is assumed that each **MechanicsElement** instance possesses the appropriate specific methods and data for correctly contributing to the assembly of the system equations.

In order to build up complex models from simpler models (see Section 2.1), one can proceed by installing the **MechanicsElement** objects which represent each member of a set of submodels into the **elementList** belonging to the higher level model. For example, in order to instantiate a four bar mechanism, one can send a message to the **MechanicsElement** subclass **FourBar** instructing it to create an instance of itself. The instantiation class method, which is coded in Smalltalk 80, is shown in Figure 2. It illustrates the process of installing instances of classes (e.g. **RigidLink** instances) into an instance of another class (e.g a **FourBar**.instance) A code fragment which uses this class method to instantiate a four bar mechanism is shown in Figure 3. In these examples the **RigidBody** subclass **RigidLink** describes a rigid body with two nodes at specific locations within the link's local coordinate system.

```
FourBar at:aPosition with:link1 with:link2 with:link3
|newFourBar|
newFourBar ← super new.
newFourBar
   installElement:link1 named:'Link 1';
   installElement:link2 named:'Link 2';
   installElement:link3 named:'Link 3';
   installNode: Node at: (link1 nodeFramePosition:'Node A') named:'Ground A';
   installNode:Node at: (link3 nodeFramePosition:'Node B') named:'Ground B';
   installRevolute: 'Revolute 1' from:self at:'Ground B' to:'Link 1' at:'Node A';
   installRevolute: 'Revolute 2' from:'Link 1' at:'Node B' to:'Link 2' at:'Node A';
   installRevolute: 'Revolute 3' from:'Link 2' at:'Node B' to:'Link 3' at:'Node A';
   installRevolute: 'Revolute 4' from:'Link 3' at:'Node B' to: self at:'Ground B';
 setPosition: aPosition.
^newFourBar
```

Figure 2. Class Instantiation Method for a Four Bar Mechanism

```
link1"RigidLink between:position1 and:position2
link2 "RigidLink between:position2 and:position3.
link3 "RigidLink between:position3 and:position3.
fourBar"FourBar at:aPosition with:link1 with:link2 with:link3.
```

Figure 3. Code for Instantiating a Rigid Four Bar Mechanism

This process is now illustrated in another example. Here (see Figure 4) a slider crank mechanism, a four bar mechanism and a simple link are installed into a **MechanicsElement** representing a more complex kinematic system.

In order to formulate multi-level representations, instances of the **MechanicsElement** class can function at more than one level of resolution. For example, calculating the dynamic response of a four bar mechanism in some circumstances requires details concerning the flexibility of its links whereas in other circumstances a rigid body assumption is adequate. In the instantiation method below (Figure five) a four bar mechanism is created from members of class **FemLink**. Class **FemLink** is a subclass of **RigidLink** whose **elementList** instance variable contains a set of plane triangular finite elements and point masses. Note that the same **FourBar** class instantiation method which was shown above in Figure 2 is used without change to instantiate the flexible four bar .

```
femLink1 " Femlink between:position1 and:position2
femLink2 " Femlink between:position2 and:position3.
femLink3 " Femlink between:position3 and:position3.
flexibleFourBar " FourBar at:aPosition
                  with:femLink1
                  with:femLink2
                  with:femLink3.
```

Figure 5. Code for Instantiating a Multi-Resolution Four Bar Mechanism

```
FourSlider at:aPosition
        with:aFourBar
        and:fourBarNode
      with:aLink
      with:aSliderCrank
        and:sliderCrankNode

| newFourSlider|
newFourSlider ← super new: aLabel.
newFourSlider
    installElement: aFourBar named:'FourBar';
    installElement: aLink named:'Link';
    installElement: aSliderCrank named:'SliderCrank';
    installNode: Node at:(aFourBar nodeFramePosition:'Ground A')
        named:'Ground A';
    installNode: (Node at:(aFourBar nodeFramePosition:'Ground B')
        named:'Ground B';
    installNode: (Node at: (aSliderCrank nodeFramePosition:'Ground A')
        named:'Ground C';
    installNode: (Node at: (aSliderCrank nodeFramePosition:'Ground B')
        named:'Ground D';
    reGround: from: self at:'Ground A' to:'FourBar' at:'Ground A';
    reGround: from: self at:'Ground B' to:'FourBar' at:'Ground B';
    reGround: from: self at:'Ground C' to:'Slider' at:'Ground A';
    reGround: from: self at:'Ground D' to:'SliderCrank' at:'Ground B'.
    aFourBar installNode: fourBarNode
        in:'Link 3' named:'Node C'.
    aSliderCrank installNode: sliderCrankNode
        in:'Link 2' named:'Node C '.
    installRevolute: 'Revolute 1'
        from:'FourBar' at:'Node C'
        to:'Link' at:'Node A';
    installRevolute: 'Revolute 2'
        from:'Link' at:'Node B'
        to:'Slider' at:'Point B';
    setPosition: aPosition.
^newFourSlider
```

Figure 4. Combining Encapsulated Model Elements

Each **FemLink** object can function circumstantially either as a rigid link or a elastic link. by responding to the messages **rigid** and **flexible.** We note that dynamic binding allows these operations to be carried out at run time. **MechanicsElement** instances in general respond to the messages **expand** and **contract.** The **expand** message causes the instance to be replaced by the contents of its **elementList** instance variable in the simulation. The **contract** message reverses the process. Experimental codes which explored this notion were written and tested successfully on static equilibrium solvers composed of springs and connection nodes.

An additional class, the **Vector** class, was created to facilitate a computational mechanics methodology that effectively manages geometric complexity in the simulations (Section 2.3). This is achieved by designing a class structure whose higher levels were free of references to a specific coordinate geometry. Specifically, the methods that use vectors are designed to function without knowing the geometric type of the vector. For example, the method for computing the resultant force on an inertial element does not need to know whether the inertial body is represented in a two dimensional space or a three dimensional space.

MechanicsElement objects use **Vector** instances to represent position, velocity, acceleration and forces. Due to polymorphism, objects that send messages to vectors need not know their geometric type or their computational processes. In effect, much of the system becomes coordinate free as desired. This is apparent in the Smalltalk 80 code shown above where no specific geometric references are made.

4. SUMMARY AND FUTURE DIRECTIONS

In this study three advanced simulation design goals related to computational multibody mechanics software were posed and successfully achieved. These goals consisted of creating reusable model elements, implementing multi-resolution simulation and reducing geometric complexity by following a coordinate free design approach. The principles of object-oriented design led directly to effective solutions to these objectives.

Further development of the object-oriented computational mechanics simulators discussed in this paper is needed in order to solidify these benefits over the long range. Especially the question of overall computational efficiency of the solvers has not been addressed in our work. Unfortunately, the advantages of object-oriented design, to date, often have resulted in reduced computational efficiency. This is not necessarily a permanent situation. Work is being done to improve the efficiency of object-oriented languages. Additionally, we expect advanced physical system simulation to combine several physical domains. For example simulating an active suspension of an automobile involves concurrent simulation of mechanical, electrical, hydraulic and feedback control mechanisms. These issues are directly addressed in [Reid 1990].

APPENDIX A. OVERVIEW OF OBJECT-ORIENTED PROGRAMMING

Object-oriented programming proposes that large complex problems can be broken down into simple, easy to manage problems with the relatively few concepts of objects, classes, encapsulation, inheritance and messages.

Objects combine data and the code that operates on that data into a single useable structure. An object captures the state and the behavior of something. The code that operates on the data, or that represents the behavior, are called methods. The data, or state, of an object are referred to as instance variables.

The way to manipulate an object's data (or state) is to send that object a message. If the message is understood by the object it will perform the requested action using its appropriate method(s). If not, the object will respond that the message was not understood. The localization of data manipulation by the object itself is known as encapsulation. One cannot manipulate an object's data structure without sending it a message. Polymorphism is the ability to send the same message to different objects. That is, the same action can be requested from different objects without special processing.

Objects sharing common properties are grouped together in a class. A class provides a template for objects so that common objects are always stored and manipulated in a consistent manner. The class feature provides a consistency in the software that typically is lacking in more conventional languages. An object is an instance of a class.

In order to take further advantage of common data structures and methods, a class hierarchy is introduced. A particular class can have subclasses (dependents) and superclasses (parents or ancestors). This family-tree-like structure is referred to as the class hierarchy. The advantage of a class hierarchy is that an object inherits all of the properties of its superclasses. This means that the object not only has available to it its own data and methods, it also has access to the data structures and methods associated with its superclasses.

Inheritance of class description reduces the information needed to build up descriptions since each statement describes how a new class differs from a previous one in the class library. This of course contributes to the reusability of software existing classes can be modified to create new ones. Another consequence of inheritance is that the descriptions associated with the higher levels in the system hierarchy can be arranged to be more abstract while detailed information is hidden in lower system levels. Thus, inheritance organizes information structures so that messages have minimal detail and type sensitivity For example the model, state space and computational processes associated with multibody mechanics are hierachically decomposed from abstract to specific. We have found that this allows for a more robust and extendible implementation.

APPENDIX B. COMPUTATIONAL MULTIBODY MECHANICS

References [Calahan 1977; Orlandia 1973; Haug 1984; Sohoni and Whitesell 1985]. describe the body of computational mechanics methods that we are studying. The research has concentrated on creating an object class hierarchy that is able to support some or all of the methodologies described below .

Any of the cases of state variables and equations of motion we are studying can be represented with mixed sets of differential-algebraic equations of the form

$$\mathbf{g}(\dot{\mathbf{y}},\mathbf{y},\mathbf{f}) = \mathbf{0} \qquad (1)$$

where the actual form of the vectors $\dot{\mathbf{y}},\mathbf{y},\mathbf{f}$ will depend on the dimension and type of state space chosen. One of the project goals is to formulate computational mechanics problems in terms of arbitrary state spaces. For the present we will assume that $\mathbf{y}$ has the form

$$\mathbf{y}^T = [\dot{\mathbf{q}},\mathbf{q},\lambda]^T$$

Here $\mathbf{q}$ and $\dot{\mathbf{q}}$ represent body position and velocity vectors. The vector $\boldsymbol{\lambda}$ represents Lagrange multipliers which correspond to algebraic kinematic constraint equations. The vector $\mathbf{f}$ has the form: $\mathbf{f} = \mathbf{f}(\dot{\mathbf{q}},\mathbf{q},\lambda,t)$ and it represents all non inertial forces. This choice amounts to a non-minimal set of body coordinates and it usually leads to large and sparse differential-algebraic equation sets An important advantage of this approach is that forming the system equations is simple since the coordinates are chosen without regard to model topology. Another advantage is the ready availability of body position velocity and force information. Numerical solution is more difficult however and special methods are sometimes necessary.

We also assume that the governing equations for the mechanical system as given by equation (1) can be linearized by taking variations about an operating point $\mathbf{p} = (\dot{\mathbf{y}}^*(t),\mathbf{y}^*(t),\mathbf{f}^*(t))$ as

$$\delta\mathbf{g} = [\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}}\delta\dot{\mathbf{y}} + [\partial\mathbf{g}/\partial\mathbf{y}]_{\mathbf{p}}\delta\mathbf{y} + [\partial\mathbf{g}/\partial\mathbf{f}]_{\mathbf{p}}\delta\mathbf{f} = \mathbf{0} \qquad (2)$$

where $\delta\dot{\mathbf{y}}$, $\delta\mathbf{y}$ and $\delta\mathbf{f}$ are variations about $\mathbf{p}$. The matrices $[\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}}$, $[\partial\mathbf{g}/\partial\mathbf{y}]|_{\mathbf{p}}$ and $[\partial\mathbf{g}/\partial\mathbf{f}]|_{\mathbf{p}}$ are partitions of the Jacobian matrix associated with the nonlinear function $\mathbf{g}(\mathbf{p})$. The linearized equation (2) is called the linear variational equation associated with $\mathbf{g}$.

B.1 Nonlinear Dynamical Analysis

To carry out the integration of equation (1) it is common to use a multistep predictor-corrector method . Usually, the corrector formula for these methods is based on a Newton-Raphson iteration scheme and has the form:

$$\mathbf{J}\Delta\mathbf{y} = -\mathbf{g}$$

where the matrix $\mathbf{J}$ depends on $[\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}}$ and $[\partial\mathbf{g}/\partial\mathbf{y}]|_{\mathbf{p}}$.

After predicting a new value for $\mathbf{y}$ on the basis of the history of $\dot{\mathbf{y}}$ and $\mathbf{y}$ over one or more preceding time steps, the residual of the governing equations , based on the predicted values of $\mathbf{y}$ is reduced by repeated applications of the corrector formula. The iterative procedure is stopped when the convergence criterion is satisfied. It has been observed that numerical difficulties can occur when attempting to solve differential-algebraic sets. However, for the purposes of this study it will be assumed that effective numerical approaches are available for integrating the equations.

B.2 Nonlinear Static Analysis

If the equation is satisfied by $\mathbf{p}$ such that $\dot{\mathbf{y}}^*(t) = \mathbf{0}$ then we say that the system is in a state of static equilibrium. Finding such a point constitutes a static equilibrium analysis. Having found such a point a linearized dynamic or linearized static analysis in a neighborhood of this point can be carried out.

B.3 Linearized Dynamic Analysis

If we assume that the mechanical system represented by g is in a state of equilibrium or other state (e.g. steady motion) at $\mathbf{p}^*$ such that matrices $[\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}}$, $[\partial\mathbf{g}/\partial\mathbf{y}]|_{\mathbf{p}}$ and $[\partial\mathbf{g}/\partial\mathbf{f}]|_{\mathbf{p}}$ are time invariant then we may carry out eigenanalysis. If we assume (2) is a homogeneous equation by choosing $\delta\mathbf{f} = 0$ and if we express $\delta\mathbf{y}$ as

$$\delta\mathbf{y} = e^{\beta t}\mathbf{z}$$

where

$\mathbf{z}$ ... complex constant vector

β... complex constant scalar

we may derive a generalized eigenvalue problem of the form

$$\{[\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}} - \beta[\partial\mathbf{g}/\partial\mathbf{y}]_{\mathbf{p}}\}\mathbf{z} = 0$$

from which we may determine $\delta\mathbf{y}$.

The matrix $[\partial\mathbf{g}/\partial\dot{\mathbf{y}}]_{\mathbf{p}}$,however, is singular if no λ terms appear in $\mathbf{g}$. Thus the eigenvalue problem is not well posed and special methods are needed.

B.4. Linearized Static Analysis.

In a state of static equilibrium $\mathbf{p}$ the linear variational equation becomes

$$[\partial\mathbf{g}/\partial\mathbf{y}]|_{\mathbf{p}}\ \delta\mathbf{y} + [\partial\mathbf{g}/\partial\mathbf{f}]|_{\mathbf{p}}\ \delta\mathbf{f} = \mathbf{0}$$

from which we may extract

$$[\partial\mathbf{g}/\partial\mathbf{q}\,|\,\partial\mathbf{g}/\partial\lambda]|_{\mathbf{p}}\ \ [\delta\mathbf{q},\ \delta\lambda]^T = -[\partial\mathbf{g}/\partial\mathbf{f}]|_{\mathbf{p}}\ \delta\mathbf{f}$$

From this equation we may determine variations in the displacement and reaction forces due to load variations $\delta\mathbf{f}$ by solving for $\delta\mathbf{q}$ and $\delta\lambda$.

REFERENCES

Benson, D. (1983), "The Simulation of Deformable Mechanical Systems Using Vector Processors," Ph.D. Thesis, Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, Ann Arbor, MI.

Calahan, D.A. (1977), *Computer-Aided Network Design*, McGraw-Hill, New York, NY.

Cox, B. (1986), *Object-Oriented Programming*, Addison Wesley, Reading, MA.

Dahl, O.J., B. Myrhaug, and K. Nygaard (1967), "SIMULA 67 Common Base Language," Norwegian Computing Center, Oslo, Norway.

Goldberg, A. and D. Robson (1983), *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA.

Haug, E.J., Ed. (1984), *Computer Aided Analysis and Optimization of Mechanical System Dynamics*, NATO ASI Series F, Vol. 9, Springer-Verlag, Heidelberg, W. Germany.

Orlandea, N.V. (1973), "Node Analogous Sparsity-Oriented Methods for Simulation of Mechanical Dynamic Systems," Ph.D. Thesis, Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, Ann Arbor, MI.

Pascoe, G.A. (1986), "Elements of Object-Oriented Programming," *Byte*, August, 39-144.

Peterson, G.E. (1987), *Object-Oriented Computing*, IEEE Computer Society, Volumes 1 and 2.

Reid, J. (1990), "Dynamic Physical System Simulation and Object Oriented Programming," Ph.D Thesis, Department of Mechanical Engineering, Michigan State University, E. Lansing, MI.

Stefik, M. and D. Bobrow (1986), "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, Winter, 40-62.

Schulz, M.H. Ed. (1981), *Elliptic Problem Solvers*, Academic Press, New York, NY.

Sohoni, V.N. and J. Whitesell (1986), "Automatic Linearization of Constrained Dynamical Models," *ASME Journal of Mechanisms, Transmissions and Automation in Design 108*, 8.

Sung, J. (1989), "The Development of a Kinematic Solver Based on Object-Oriented Programming Principles," M.S. Thesis, Department of Mechanical Engineering, Michigan State University, E. Lansing, MI.

Wielenga T.W. (1984), "Simplifications in the Simulation of Mechanisms Containing Flexible Members," Ph.D Thesis, Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, Ann Arbor, MI.