OBJECT–ORIENTED MODELING: MEANS FOR DEALING WITH SYSTEM COMPLEXITY

FRANÇOIS E. CELLIER

Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona 85721-0104, U.S.A.

e_mail: Cellier@ECE.Arizona.Edu

URL: http://www.ece.arizona.edu/~cellier

Abstract This paper presents the concepts of and ideas behind the object–oriented modeling paradigm in the context of rapid prototyping of complex physical system designs. It is shown that object–oriented modeling software is an essential tool in flexible manufacturing, which helps reduce both the cost and the time needed to manufacture customized goods using pre–fabricated components.

Keywords: Object–oriented modeling; Software reuse; Rapid prototyping; Flexible manufacturing.

INTRODUCTION

For more than 30 years, engineers have been describing physical systems through either linear or nonlinear state–space models. The modeling and simulation tools developed during this era, such as ACSL (MGA, 1991), reflect this emphasis on causal models.

Physics, however, is essentially acausal (Cellier *et al.*, 1995). It is one of the most deep–rooted myths of engineering that algebraic loops in models result from neglected fast dynamics. This myth is based upon the engineers' infatuation with state–space models. Since state–space models are what the engineers know, they believe that this is also how the universe operates. Whenever they encounter an algebraic loop in a model, they introduce a "small capacitor" (a storage element) to break it, and claim that they actually represent the physical realities more faithfully in this way.

It was Sir Isaak Newton himself who was one of the spiritual fathers of the myth about causality. His law of *actio et reactio* stipulates that someone commits a willful act (the *actio*) to which the universe reacts (the *reactio*). He distinguishes between perpetrators and victims, between the culprit and the innocent.

Unfortunately, physics doesn't work that way. There is no physical experiment in the world that could distinguish whether someone drove his car into a tree, or whether it was the tree that drove itself into the car. To this date, no–one has come up yet with a convincing physical foundation of the concept of free will.

Well, maybe this doesn't matter. Engineering is all about simplifications, reasonable compromises, useful working hypotheses — much more so than about truth. Maybe, thinking about cause and effect relationships helps the engineer conceptualize his or her task better or more easily, and therefore, it makes sense to pretend that the universe is a causal one, even though physics seems to indicate that causality is a mere illusion?

It turns out that this is not the case. Chasing after the chimera of the physical whodunit makes understanding the mechanisms of physics, modeling them in terms of mathematical equations for the purpose of simulating these models to predict the future, more difficult rather than easier.

Every respectable electrical engineer knows that there are two types of electrical resistors. There is the current–flow–causes–voltage–drop type that obeys the law

$$u = R \cdot i \tag{1}$$

and there is the voltage–drop–causes–current–flow type that obeys the different law

$$i = \frac{u}{R} \tag{2}$$

and consequently, his or her toolbox must contain two types of mathematical models for dealing with these two species of resistors.

In reality, there is of course only one type of resistor, and the law describing it is:

$$u \cdot i = T \cdot \dot{S} \tag{3}$$

with the constraint that:

$$u - R \cdot i = 0 \quad ; \quad R > 0 \tag{4}$$

i.e., the law describes the conversion of electrical into thermal energy without a loss, and the constraint ensures that this conversion is irreversible, i.e., that a resistor cannot be used to produce electrical energy from heat.

The need to distinguish between different hypothetical types of otherwise identical objects, such as resistors, makes the life of the engineer very difficult indeed. It is this artificial necessity that is the main obstacle to complexity. If a system contains one resistor, the modeler needs to take into consideration 2 possible models. However, if the system contains 10 resistors, there are already 1024 different possibilities to take into consideration.

The object–oriented modeling paradigm does away with the need to describe physical laws through cause–and–effect relationships, and this makes it possible to describe even very complex processes with ease and confidence.

FLEXIBLE MANUFACTURING

Recent years have shown an increasing demand for pre–fabricated goods with lots of options that the customer can choose from. This is a compromise between mass–fabrication and individual customer design, whereby the customer is provided with as much flexibility as feasible in influencing the final product without driving either cost or delivery time very much up in comparison with the mass–fabricated product.

In earlier years, this compromise had simply not been available. Components of systems had not been designed for reuse, and any modification in design required a substantial re–evaluation, which made such systems almost as expensive or possibly even more expensive than individually designed ones. The markets for flexible manufacturing depend heavily on the ability of the producer to maximize flexibility, while keeping the cost down and providing as fast a response time as possible on customized orders.

These demands can only be met by a high degree of flexible automation in evaluating and optimizing individualized designs. This goes hand in hand with a demand for flexible modeling and simulation tools, whereby hardware components are described by corresponding software modules that must be combinable in at least the same flexible manner as the hardware components themselves.

Both the cost of redesign and the time needed to accomplish it depend directly on the modularity and flexibility of the modeling and simulation tools available for the task at hand. The object–oriented modeling paradigm offers a good answer to all these demands.

SOFTWARE COST AND DEVELOPMENT TIME

The main factors that help reduce both the cost and development time of software are:

1. *Reusability:* A software design methodology that ensures optimal reusability of software components is the most essential factor in keeping the software development and maintenance cost down.
2. *Abstraction*: Higher abstraction levels at the user interface help reduce the time of software development as well as debugging. The conceptual distance between the user interface and the final production code needs to be enlarged. Software translators can perform considerably more tasks than they traditionally did.

The object–oriented modeling concept helps in both areas. It ensures optimal reusability of component models while supporting a high degree of abstraction. It is thus optimally suited to support flexible manufacturing in the rapid and cost–effective development of customized product designs.

OBJECT–ORIENTED MODELING: A KEY TECHNOLOGY FOR SOFTWARE REUSABILITY

Continuous–time processes need to communicate with each other at very high frequency. If one module needs a continuously varying variable computed by another module, that information must repetitively and at a high frequency be sent from the former to the latter. For this reason, it is very inefficient to use object–oriented programming in the design of continuous–time simulation software. Message passing between objects is unacceptably slow. The simulation code must be made monolithic. This is one of the main reasons why the production cost of simulation software for physical processes is so high, and why the code, once written, is so hard to maintain and upgrade.

Although the above statement is certainly correct, this does not imply that the user interface of the model has to be made monolithic. Modern compiler technology is capable of translating an object–oriented model definition down to a monolithic simulation program. What has been offered in traditional simulation languages, such as ACSL, in terms of model compilation is trivial. All that the ACSL compiler does is replace macro calls by macro bodies (code expansion), and then sort the equations into executable sequence. Considerably more can be done, and is indeed needed for true software modularity and proper software reusability.

The problem starts with the simple resistor model. In ACSL, the user would need two different macros to describe a resistor. If the resistor is placed in series with an inductor, then the current through the resistor is a state variable, and thus, it is the current flowing through the resistor that "causes" a voltage drop. However, if the same resistor is placed in parallel with a capacitor, the voltage across the resistor is a state variable, and hence it is now the potential difference between the two legs of the resistor that "causes" a current to flow.

Computational causality is a numerical artifact, related to the way in which differential equations are solved. It it not a physical concept. Physics is basically acausal.

In order to be able to model physical systems in a truly modular fashion, the modeling environment must reflect the acausal nature of physics. Models (object descriptions, or object class descriptions) must be collections of physical laws pertaining to the described object, and not assignment statements. The interface points of a model must be collections of variables that can be shared between models, not a list of input and output variables. Connections between objects must be non–directional. A connection between two objects simply declares that these two objects share a set of variables.

This concept has been coined the *object–oriented modeling paradigm*. It was originally invented in 1978 by Hilding Elmqvist as part of his Ph.D. dissertation (Elmqvist, 1978).

The object–oriented modeling paradigm shares many of the properties of object–oriented programming. Its main characteristics can be summarized as follows (Cellier *et al.*, 1995):

- *Encapsulation of knowledge*: The modeler must be able to encode all knowledge related to a particular object in a compact fashion in one place with well–defined interface points to the outside.
- *Topological interconnection capability*: The modeler should be able to interconnect objects in a topological fashion, plugging together component models in the same way as an experimenter would plug together real equipment in a laboratory. This requirement entails that the equations describing the models must be declarative in nature, i.e., they must be acausal.
- *Hierarchical modeling*: The modeler should be able to declare interconnected models as new objects, making them indistinguishable from the outside from the basic equation models. Models can then be built up in a hierarchical fashion.
- *Object instantiation*: The modeler should have the possibility to describe generic object classes, and instantiate actual objects from these class definitions by a mechanism of model invocation.

- *Class inheritance*: A useful feature is class inheritance, since it allows the encapsulation of knowledge even below the level of a physical object. The so encapsulated knowledge can then be distributed through the model by an inheritance mechanism, which ensures that the same knowledge will not have to be encoded several times in different places of the model separately.

- *Generalized Networking Capability*: A useful feature of a modeling environment is the capability to interconnect models through *nodes*. Nodes are different from regular models (objects) in that they offer a variable number of connections to them. This feature mandates the availability of *across* and *through* variables, so that power continuity across the nodes can be guaranteed.

The first task of the model compiler must be to extract all the physical laws from the submodel instantiations, adding the connection equations that are derived from the model topology. This set of equations generally constitutes a higher–index differential–algebraic equation (DAE) system, i.e., an implicit model of the type:

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{w}, \mathbf{u}, t) = 0 \tag{5}$$

where $\mathbf{x}$ is a set of state variables, $\mathbf{w}$ is a set of algebraic variables, $\mathbf{u}$ is a set of inputs, and t is the time. The number of state variables is usually larger than the degrees of freedom of the system, i.e., the true model order.

MIXED SYMBOLIC AND NUMERIC MODEL PROCESSING: THE KEY TO RUN–TIME EFFICIENCY OF GENERATED SIMULATION CODE

It has been recognized for some time that DAEs play an important role in physical system modeling (Brenan *et al.*, 1989). Due to the intrinsically acausal nature of physics, most models do not lead naturally to an explicit ordinary differential equation (ODE) model of the form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{6}$$

that can be solved with regular ODE solvers. The tendency has been to solve the resulting DAE set directly using a numerical DAE solver. Several powerful DAE solvers have been made available meanwhile, including DASSL (Petzold, 1982) and Radau (Hairer and Wanner, 1991).

The problems with this approach are twofold. On the one hand, these numerical DAE solvers are not suited for solving higher–index DAE systems. A symbolic algorithm is known (Pantelides, 1988) that makes it possible to automatically reduce the index of a DAE system down to index 1. This algorithm is very fast (linear complexity) and completely harmless, i.e., it does not lead to an explosion of the size of the model, one of the major drawbacks of many symbolic formula manipulation algorithms, and if implemented right, does not have any numerical drawbacks, such as drifting of state variables.

The resulting index 1 DAE system can now be solved numerically by means of a standard DAE solver, or it can be reduced further symbolically to an explicit ODE form using a graph–theoretical algorithm of linear complexity (Tarjan, 1972). This transformation is not harmless, though, and whether or not it is efficient to make this transformation depends on the application. However, the transformation to explicit form is not essential, since most larger physical models are very stiff, i.e., it is necessary to use an implicit integration algorithm anyway, and consequently, not much is gained by an explicit model formulation.

The second problem relates to the size of the Newton iteration needed as part of the DAE solver. If the resulting index 1 DAE model contains n state variables ($\mathbf{x}$) and k implicitly coupled algebraic variables ($\mathbf{w}$), DASSL would need to iterate on $(n + k)$ variables using Newton iteration once per integration step. The fifth–order Radau algorithm will even need to iterate on $3(n+k)$ variables. Newton iteration instead of fixed–point iteration is needed because a fixed–point iteration would destroy the numerical stability properties of the algorithm that were the primary reason for using an implicit method in the first place. This means that a linear system of equations of size $n + k$ (or $3(n+k)$ respectively) needs to be solved once per integration step. This calls for the repeated LU–decomposition of an often very large Jacobian matrix. This can be a very expensive proposition.

A new mixed symbolic and numeric DAE solution technique was recently discovered that revolutionizes the way in which DAEs can be solved efficiently. The approach has been coined *inline integration* (Elmqvist *et al.*, 1995). The principal idea is to merge the model equations with the solution equations, thereby symbolically converting the former DAE problem to an implicitly formulated set of difference equations. Tearing (Elmqvist and Otter, 1994) is then applied to the so transformed model to reduce the number of iteration variables to a minimum.

In a six degree–of–freedom Manutek R3 robot with drive trains, controllers, and control electronics, a model of 66^{th} order, this approach led to a reduction in the number of iteration variables from 66 to 6, and of the size of the Jacobian from 4356 to 36. The gain in execution speed of the Newton iteration was a factor of 18.

Whereas the original robot simulation (using DASSL as the DAE solver) was already as fast as the best of today's special purpose multibody system (MBS) simulators, the new code can now compete with the best that a human programmer could achieve in manually optimized simulation code. The conceptual distance between the original DAE model and the finally generated simulation code is now very large, and it would be almost impossible for a human programmer to derive such a code manually from the original model, and if he or she did, the code would be totally unreadable, very hard to maintain, and even harder to upgrade. The code would be extremely specific, and hardly any of the code could ever be reused again.

In contrast, inline integration is a tool that can be completely automated, i.e., the human modeler can generate the object–oriented model in a highly abstract and very intuitive fashion. The entire translation down to the optimized simulation run–time code is fully automated. All parts of the model are modular and reusable, and the original model code is by at least a factor of 10 shorter in the number of lines or characters than the ultimately generated simulation code.

IMPLEMENTATION IN DYMOLA

Although the heart of the presentation is on the concepts of and ideas behind the object–oriented modeling paradigm, it shall be demonstrated also how these ideas have been realized in the Dymola suite of programs. The system consists of four programs:

1. *Dymodraw:* This software tool is a graphical general–purpose object diagram editor that enables the user to compose models of systems from models of subsystems by topologically connecting them in a fashion similar to how an experimenter would assemble a real system in the laboratory. The end product of the graphical model is an object–oriented textual model encoded in Dymola.

2. *Dymola*: This software tool is a symbolic formula manipulation system that assembles the equations from the submodels, adds the appropriate connection equations, and then transforms the equations to a form suitable for the simulation system. Dymola can generate code for a variety of simulation languages including ACSL, SimuLink, Desire, and a few more. It can also directly generate either Fortran (DSblock) or C (Dymosim) code.

3. *Dymosim*: This software is a C–based continuous–time simulation system. It supports both ordinary differential equations and differential–algebraic equations. It also supports proper discontinuity handling. Dymosim generates its results in tabular form for use by either Matlab or Dymoview.

4. *Dymoview*: This software is a graphical postprocessor that provides plotting of simulation results as well as 3D postanimation of mechanical systems, such as robots, vehicles, or missiles.

The heart of the system (Dymola) had originally been designed at the Lund Institute of Technology by Hilding Elmqvist as part of his Ph.D. dissertation (Elmqvist, 1978), was then further developed at the University of Arizona (Cellier, 1991), and is currently under further development by Dynasim AB (Elmqvist, 1995). The simulator (Dymosim) had originally been developed at the German Aerospace Research Establishment (DLR) in Oberpfaffenhofen (Otter, 1992). The two graphical tools are recent additions from Dynasim AB.

REFERENCES

Brenan, K.E., S.L. Campbell, and L.R. Petzold (1989), *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations*, Elsevier Science Publishers, New York, new ed. to appear in 1995.

Cellier, F.E. (1991), *Continuous System Modeling*, Springer–Verlag, New York.

Cellier, F.E., H. Elmqvist, and M. Otter (1995), "Modeling from Physical Principles," *The Control Handbook* (W.S. Levine, ed.), CRC Press, Boca Raton, FL.

Elmqvist, H. (1978), *A Structured Model Language for Large Continuous Systems*, Ph.D. Dissertation, Report CODEN: LUTFD2/(TFRT–1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1995), *Dymola: Dynamic Modeling Language — User's Guide*, Dynasim AB, Lund, Sweden.

Elmqvist, H., and M. Otter (1994), "Methods for Tearing Systems of Equations in Object–Oriented Modeling," *Proc. ESM'94, SCS European Simulation MultiConference*, Barcelona, Spain, pp. 326–332.

Elmqvist, H., M. Otter, and F.E. Cellier (1995), "Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential–Algebraic Equation Systems," keynote presentation, *Proc. ESM'95, SCS European Simulation MultiConference*, Prague, Czech Republic.

Hairer, E. and G. Wanner (1991), *Solving Ordinary Differential Equations II. Stiff and Differential–Algebraic Problems*, Springer–Verlag, Berlin.

Mitchell & Gauthier Assoc. (1991), *ACSL: Advanced Continuous Simulation Language — User Guide and Reference Manual*, Concord, Mass.

Otter, M. (1992), *DSblock: A Neutral Description of Dynamic Systems*, Version 3.2. Technical Report TR R81–92, DLR, Institute for Robotics and System Dynamics, Wessling, Germany. Newest version available via anonymous ftp from "rlg15.df.op.dlr.de" (129.247.181.65) in directory "pub/dsblock".

Pantelides, C.C. (1988), "The Consistent Initialization of Differential–Algebraic Systems," *SIAM J. Scientific and Statistical Computing*, **9**, pp. 213–231.

Petzold, L.R. (1982), "A Description of DASSL: A Differential/Algebraic System Solver," *Proc. 10th IMACS World Congress*, Montreal, Canada.

Tarjan, R.E. (1972), "Depth First Search and Linear Graph Algorithms," *SIAM J. Comp.*, **1**, pp. 146–160.