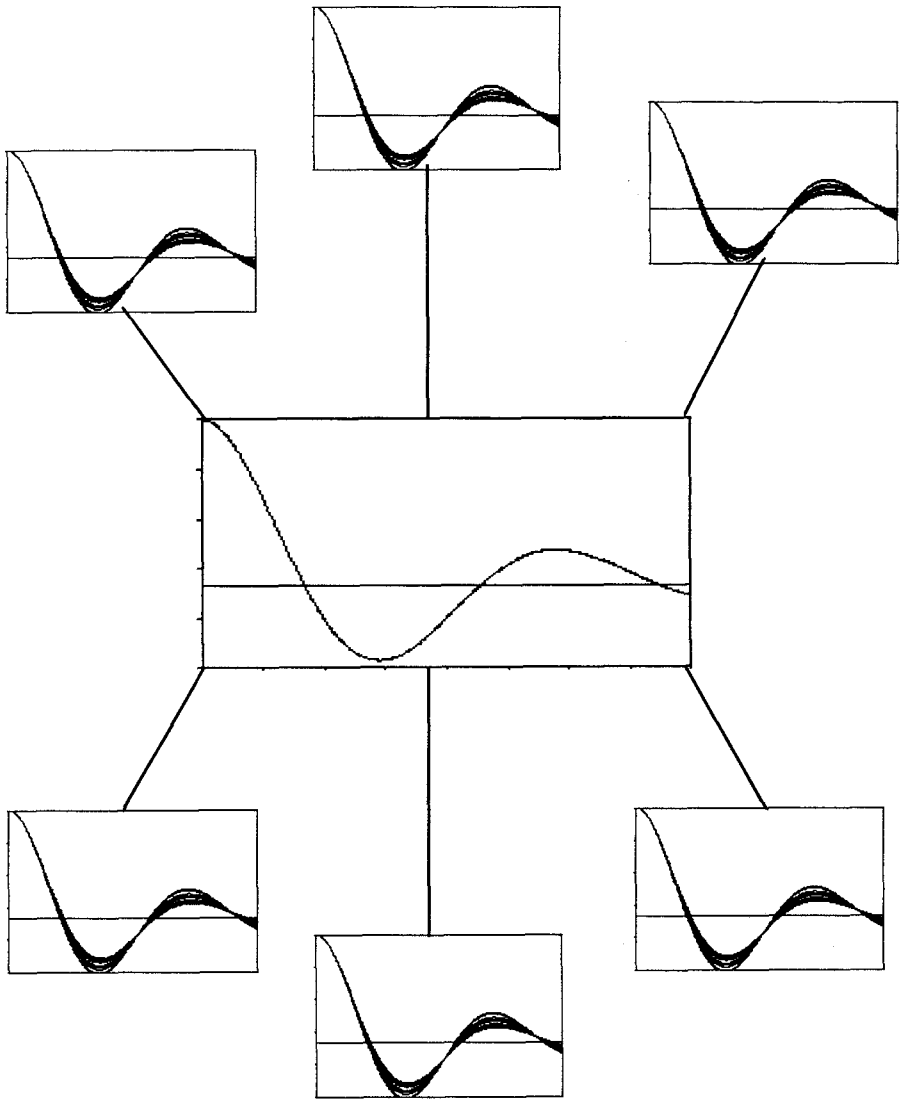
SIMULATION NEWS EUROPE

EUROSIM

ISSN 0929 - 2268



Number 10

March 1994

A EUROPEAN FORUM ON SIMULATION ACTIVITIES

Teaching Physical System Modeling at the University of Arizona

Francois E. Cellier
Dept. of Electr. & Comp. Engr., The University of Arizona
Tucson, Arizona 85721 U.S.A.
Cellier@ECE.Arizona.Edu

Abstract

This paper deals with the subject of teaching university students the art of physical system modeling, a topic notoriously downplayed in the education of our future scientists and engineers. It is shown that, although modeling is truly an "art" and not a "science", it is not a "black art", i.e., it is a subject that can indeed be taught successfully, since it observes a set of general rules and practices - almost like a science. Moreover, computer programs can be provided that gently push the user towards following recommended procedures in the making of physically sound models, that help him or her in verifying the consistency and completeness of these models, and support him or her in detecting errors and unraveling shortcomings in them.

Introduction

Students are trained to accept the fact that a program is "incorrect" as long as it produces error messages during compilation or at run time, but they assume it to be "correct", as soon as these error messages have disappeared. I have rarely seen students in my career as a university teacher who would turn around once they received beautifully looking multi-colored graphs from their simulation program to check whether their model carries any physical meaning or not.

Simulation languages have traditionally been optimized for short user programs, thereby sacrificing all redundancy - and most possibilities to discover errors in the code. For example, few of today's simulation languages require variables to be declared. Thus, mistyping of variable names will often go unnoticed. You write:

*variable*1 = *par*1 * *cons*2
*variable*2 = *variable*3 + *par*2 * *varaible*1
plot *variable*2

... but don't despair! You'll still get nice multi-colored graphs. *Variable1* is now a variable that is defined but never used (well, why not!), and *varaible1* is another variable that is used but never defined ... but the simulation compiler is "helpful" - it simply defaults *varaible1* to 0.0, making *variable2* equal to *variable3*, which can be plotted beautifully and elegantly. Thus, the program "runs," and our student can turn to the next homework problem - after all, time is money, right?

Traditionally, most classes on continuous-system simulation offered at our universities have focused on the process of translating models into the format required by the simulation language in use. This trend was encouraged by simulation language manuals that do exactly the same. Simulation languages *are* very useful. They support the user in encoding his or her model, and offer nice interfaces to the numerical software (the integration algorithms) and to the graphing software. However, simulation languages offer no support at all for helping the user in getting his or her equations right - zero, nil, zip! Yet, this is the most difficult part of the overall system analysis cycle.

For these reasons, we, at the University of Arizona, have split the task of teaching modeling and simulation into two. In a first semester, the student is taught how to model physical systems, i.e., how to get the equations of his or her model right, and in the subsequent semester, the student is then taught the trade of simulating these previously obtained equations. Thus, the first semester [2] has a flavor of *theoretical physics*, instructing the student what to watch out for when applying his or her meta-knowledge to a particular situation or when turning experimental observations into mathematical equations, whereas the second semester [6] has a flavor of *applied mathematics*, training the student in techniques of numerical integration of ordinary differential equation (ODE) systems and differential algebraic equations (DAE) systems, the conversion of partial differential equations (PDEs) into sets of ODEs by the method-of-lines, simulation with noise, and problems in numerical and semi-analytical parameter estimation and state identification.

Modeling Software

While the process of *modeling*, i.e., the codification of knowledge about physical systems into mathematical equations, cannot usually be fully automated, this does not say that computers and computer programs have no part in that process. Modeling programs can:

- support the user in organizing partial knowledge about the physical system under study,
- help the user with identifying missing equations, i.e., pointing out to him or her which phenomena have not yet been modeled,
- aid the process of verifying internal consistency within equations, and cross-consistency between equations, and finally
- preprocess the model through symbolic formula manipulation to generate a syntactically correct and semantically sound simulation program.

Dymola

The most advanced of the currently available modeling tools is Dymola [2, 5, 7, 8]. We found that Dymola is a big help in teaching modeling to students.

Dymola supports the user in several different ways:

- Dymola forces the user to declare all variables (helas!), and provides valuable diagnostic aids during the model translation process.
- Dymola is object-oriented and thereby truly modular. It enables the user to encapsulate descriptions of physical phenomena in hierarchically structured model classes. True, most simulation languages offer a macro capability, but macros are a much less powerful concept. Dymola supports the construction of truly modular domain libraries of properly debugged component models representing task-relevant aspects of physical objects.
- Dymola supports the topological connection of subsystems. Topology is a powerful concept in ensuring model correctness. If the component submodels are correct, the topologically connected higher-level model is most likely correct as well. Humans are quite good at discovering wiring errors in topological descriptions, and Dymola again introduces helpful redundancy to ensure the discovery of wiring errors - as it is on purpose made rather difficult to squeeze a European 220 Volt plug into a U.S. 110 Volt socket, and vice-versa, Dymola provides mechanisms to ensure the compatibility between software "plugs" and software "sockets."
- Maybe most importantly of all: Dymola supports the use of *bond graphs* [1, 2, 3, 4, 10]. Bond graphs are a very powerful object-oriented tool for ensuring correctness of models of physical systems. Bond graphs model the flow of power through a physical system. Thereby, strict observation of the energy conservation law is automatically enforced, an important concept in model validation.

This last item gives rise to an interesting comment. When Hilding Elmqvist designed Dymola in 1978 [7], he did not think about bond graphs at all. Yet, the model definition and connection capabilities designed into Dymola turned out to be powerful enough to support bond graph modeling *without necessity to change a single line of code in the Dymola translator*. This fact alone is a strong argument in favor of object-oriented modeling. However, it is by no means the only one that can be made. The object-oriented approach to modeling allows complete encapsulation of all relevant properties of a physical object in a software module called a *model class*. Many domain-specific model classes can be stored together in a *domain library*. In this way, mathematical descriptions (models) of physical objects can be encoded and carefully debugged once and for all for the benefit of later users of these models. Very elaborate domain libraries have already been made available, e.g. one for modeling all kinds of tree-structured multi-body systems, such as robots [9].

For all these reasons, Dymola has become a major backbone in supporting my modeling class at the University of Arizona [2]. Since Dymola has been introduced into the class, the quality of student simulation programs as well as the student understanding of the mechanisms of modeling have been drastically improved. Also, the students love it! They are *highly motivated* in the class because they understand that their learning focuses around fundamental properties of physical systems rather than the nitty-gritty details of the - necessarily contemporary - syntax of a programming language.

Teaching bond graphs, a modeling technique that many engineers and scientists out in the field still shun away from because the graphical representation looks unfamiliar and non-intuitive to them, turned out to be easily accepted by students. Contrary to accomplished engineers and scientists who already know (or at least believe to know) how physical systems are to be modeled and who don't have either the time or the inclination to learn something drastically different unless they are forced to do so, students are open-minded. It is their *raison d'être* to digest new ideas and master new concepts, and they have plenty of time set aside for this task. They find bond graphs neither obscene nor otherwise repulsive, and once they got the hang of it, they turn quickly into experienced physical system modelers. The problem is *not* jotting down some equations and getting them to compile without producing error messages ... that is the easy part. The problem is what to do if these equations turn out to poorly reflect reality. How do you go about determining which part of the model is inappropriate and needs to be modified? Bond graph modelers have a much better chance of getting

the model right the first time around, and they find it much easier to enhance/modify a given model in order to incorporate additional facets of reality into it.

It is the same as with programming in general. It is very hard to convince a programmer "of the old school", i.e., someone who grew up on Fortran and has years of experience in Fortran programming to try something new. The fact is that he or she indeed *can* program anything and everything in Fortran. However, it is easy to teach students structured programming, and they will undoubtedly and quickly become more reliable and efficient programmers in comparison with the old-timers.

Justification

Why is it important to train our students in the "art" of physical system modeling? Is it not true that a detailed and very specific domain knowledge is necessary for coming up with useful mathematical models in *any* domain? Do I truly and earnestly believe that I can turn my students into "renaissance men and women", into "scientists" who are at the same time mathematicians, physicists, engineers, and philosophers, i.e., into generalists that can compete with the domain specialists in solving the relevant problems of this world?

I believe strongly that there is still need for both the specialist and the generalist. It is true that in the past, at the time of an Isaak Newton maybe, it was possible for a physicist to know everything about all aspects of physics. At that time, there was no need for specialization. This is no longer true. Without the specialists who know everything there is to be known about a very small and specialized subset of physics, no true progress can be achieved any longer. Thus, the domain specialists have become an essential and valuable part of our scientific community, and indeed, there are many more specialists needed than generalists.

However, this does not mean that there is no longer any need for generalists at all. Many problems in science and engineering are interdisciplinary in nature. Our domain specialists usually suffer from a syndrome that the French call *déformation professionelle*. They feel so cozy and comfortable in their small niches of science that they have no inclination what-so-ever to look outside their realms and consider the interactions of their tiny empires with the larger world. This is where the generalists are still needed. Generalists are the mediators between domain specialists of different domains. They translate the language of one domain specialist to another. They look at the larger picture. It is also true that, even in this century, the most important new discoveries in physics were made by generalists rather than by specialists, and I predict that this will remain so.

A tree has many small branches and only one stem. The small branches are responsible for carrying the leaves and for keeping the tree alive. Small branches can eventually grow thicker and branch into new sub-branches. Yet, the overall shape of the tree can't be fundamentally changed by activities of the many small branches alone. Sometimes, an entirely new branch has to grow out of the stem directly, a branch that will then grow much more rapidly than the small branches and quickly develop an entire new subsystem of branches and sub-branches and leaves.

References

[1] Brooks, B.A., and F.E. Cellier, "Modeling of a Distillation Column Using Bond Graphs", *Proceedings ICBGM'93: SCS International Conference on Bond Graph Modeling*, San Diego, Calif., pp. 315-320, January 17-20, 1993.

[2] Cellier, F.E., *Continuous System Modeling*, Springer-Verlag, New York, 1991.

[3] Cellier, F.E., "Hierarchical Non-Linear Bond Graphs: A Unified Methodology for Modeling Complex Physical Systems", *Simulation*, **58** (4), pp. 230-248, 1992.

[4] Cellier, F.E., "Bond Graphs - The Right Choice for Educating Students in Modeling Continuous-Time Physical Systems", *Proceedings ICSEE'92: SCS International Conference on Simulation in Engineering Education*, Newport Beach, Calif., pp. 123-127, January 20-22, 1992.

[5] Cellier, F.E. and H. Elmqvist, "Automated Formula Manipulation Supports Object-Oriented Continuous-System Modeling", *IEEE Control Systems*, **13** (2), pp. 28-38, 1993.

[6] Cellier, F.E., *Continuous System Simulation*, Springer-Verlag, New York, 1995.

[7] Elmqvist, H., *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, 1978.

[8] Elmqvist, H., F.E. Cellier, and M. Otter, "Object-Oriented Modeling of Hybrid Systems", *Proceedings ESS'93, European Simulation Symposium* , Delft, The Netherlands, pp. xxxi-xli, October 25-28, 1993.

[9] Otter, M., H. Elmqvist, and F.E. Cellier, "Modeling of Multibody Systems With the Object-Oriented Modeling Language Dymola", *Proceedings NATO/ASI, Computer-Aided Analysis of Rigid and Flexible Mechanical Systems*, Troia, Portugal, Vol. 2, pp. 91-110, June 27 - July 9, 1993.

[10] Weiner, M., and F.E. Cellier, "Modeling and Simulation of a Solar Energy System by Use of Bond Graphs", *Proceedings ICBGM'93: SCS International Conference on Bond Graph Modeling*, San Diego, Calif., pp. 301-306, January 17-20, 1993.