1

Integrated Continuous-System Modeling and Simulation Environments

François E. Cellier

University of Arizona
Tucson, Arizona

1.1 INTRODUCTION

The field of simulation software was last reviewed by me in 1983 [7]. A lot has happened since 1983. At that time, most continuous-system simulations were still performed on either CDC or IBM mainframes. Many engineers still wrote their simulation software in Fortran because the simulation languages of that era were not available at the site (mainframe software tended to be quite expensive), or not implemented on the particular hardware platform, or too slow for the intended purpose, or too restricted in their modeling capabilities.

Today's Engineering Workstations place more number-crunching power and a larger memory allocation on the average engineer's personal desk than the mainframes of one decade ago had to offer to an entire enterprise. We have seen a trend toward standardization of operating system software across different hardware vendors with, since the design of the RISC architectures, a strong trend toward accepting UNIX as the "universal" operating system language, and C as the "universal" programming language. We have seen standardization of graphics software with X-Windows becoming the de facto standard of low-level graphics, and Open Look and Motif the (unfortunately still two) de facto standards of higher-level graphic functions. We have seen a standardization of the ASCII representation of graphics in the form of the Postscript language, which, for the first time, allows engineers and scientists to make their papers (including figures) available electronically to their colleagues around the globe

by placing them in so-called "anonymous FTP" accounts. We have finally seen the general acceptance of the object-oriented (OO) programming paradigm as a means of managing large pieces of code in a modular fashion with C++ emerging as the most widely used OO programming language.

One decade ago, the operating system kernels offered on mainframes were extremely rudimentary. This was a deliberate choice since computer manufacturers wanted to make a large percentage of the scarce computer cycles and memory cells available to the end user, keeping the overhead of the operating software (both in terms of CPU cycles and occupied memory) as small as they could get away with. Today's trend is just the opposite. The operating system software is made as comfortable to use as possible, irrespective of how much resources the operating system consumes. The time of the engineer is a considerably more precious and scarcer commodity than either CPU cycles or memory chips. After all, the Engineering Workstation is idling most of the time, waiting for its slow single interactive user to issue the next command.

For this reason, the implementation of flexible integrated software environments was unthinkable at the time when my last survey was written. To be more precise, the first integrated simulation environment, TESS [39], was in its early design phase [38] around the time when my last review was written. However, the first version of TESS, released in 1985, offered a rather crude environment (operating) language, rudimentary graphics only, a painfully slow and not very robust database, and was generally a far cry from what can be achieved today. TESS deserves credit though for being visionary in predicting what simulationists would ask for in terms of simulation support software in the years to come.

In light of the rapid development of computer technology over the past decade, I was delighted when I was asked to undertake a new effort of surveying the state-of-the-art of continuous-system simulation software. However, whereas my previous review focused on features and capabilities of individual simulation languages, the current review places its emphasis on integrated modeling and simulation software environments, stating what has been achieved so far, and daring to predict what the near future might bring in addition.

1.2 SIMULATION SOFTWARE

Many of the simulation languages that were reviewed in 1983 are still in use. If anything, they have become more popular than ever. ACSL [31] is still the most widely used continuous-system simulation language on the market, and for good reasons. It provides flexible model specification capabilities, excellent integration algorithms, and both the ACSL preprocessor and the ACSL run-time system are satisfyingly robust.

One of the major reasons why I did not and could not use ACSL in my research projects at the time of my last review was ACSL's lack of capabilities to handle discontinuities properly [6]. However, shortly after my last review, the

schedule statement was introduced into ACSL, which now allows one to handle discontinuous models adequately. This feature is still not implemented in an optimal fashion because many of the built-in discontinuous functions (such as the *step* function) have not been recoded to make use of the new facility, but this does not prevent me from using ACSL; it only prevents me from using those built-in functions.

At the time of my last review, ACSL had been fairly new on the market and its preprocessor still contained an unhealthy number of bugs. However, Mitchell & Gauthier offer excellent software support. When I report a problem to them, I usually obtain a fix within 24 to 72 hours. In the mean time, ACSL has matured tremendously. Its preprocessor is now mostly bug-free. Over the past 2 years, I discovered only one new true bug in the ACSL compiler, which was related to a table overflow with handling an unearthly large model (ACSL provided for 10,000 generic variable names, whereas my program needed more). As usual, I received a bug fix within less than a day.

One decade ago, the interface between ACSL's run-time software and its integration algorithms, particularly the Gear algorithm, still had a few problems. However, in the meantime these have been fixed, and I have not discovered any new integration problems with ACSL in a long time.

The availability of ultrafast Engineering Workstations (45 Mips or more) makes it now feasible to apply ACSL even to very large and numerically difficult problems such as the solution of two-dimensional parabolic partial differential equations discretized using the method-of-lines approach.

The initial version of ACSL (running on VAX/VMS systems only) was still fairly expensive (around $10,000) though not as outlandishly expensive as some of its competitors. However, healthy competition and plummeting hardware prices have driven the price of the ACSL software down to a level where it is comfortably affordable to anyone who needs it. On a PC-based system, an educational version of ACSL now sells for a few hundred dollars.

The only remaining major weakness of ACSL is its inability to handle algebraic loops adequately. Although ACSL provides for an implicit loop solver, this tool is totally inadequate and inappropriate.

One of the major achievements of CSSL-type languages [2], such as ACSL, is their equation sorter. Users can group equations together in a fashion that is convenient from a modeling point of view, rather than having to worry about properties of the underlying numerical solution algorithms that may call for a drastically different statement sequence. The simulation preprocessor sorts the model equations into an executable sequence. This facility is no big deal as long as the user plays around with models consisting of 20 equations, but it becomes most essential when the size of the model grows to several hundreds or even thousands of equations, as this is now commonly the case.

Unfortunately, algebraic loops often cut across many different subsystems, forcing the user to group equations together that are involved in an algebraic

loop even if they logically belong to different subsystems. Algebraic loops are nasty from a numerical point of view, but the user should not have to worry about them. Use of ACSL's implicit loop solver forces the user to again think about the properties of the underlying numerical algorithm, which is exactly what we tried to prevent by introducing an equation sorter.

Also, algebraic loops can be quite formidable in size, especially when modeling chemical systems. A student of mine once formulated a dynamic model of a 50-tray distillation column. He ended up with an algebraic loop involving exactly 2573 equations! Agreed, my student was still fairly inexperienced when he wrote this model. A more experienced modeler would probably have been able to produce a more manageable model. However, the example is still quite realistic. ACSL's implicit loop solver is a rather inefficient tool for handling large algebraic loops. In such a situation, it may be more appropriate to employ an implicit numerical integration scheme, a so-called DAE-solver [5] that is able to handle problems of the type

$$\mathbf{f}(\mathbf{x},\ \dot{\mathbf{x}},\ \mathbf{u},\ t) = 0.0 \tag{1.1}$$

in place of the traditionally used explicit numerical integration schemes (the so-called ODE-solvers) that handle problems of the type

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\ \mathbf{u},\ t) \tag{1.2}$$

I recommended strongly that Mitchell & Gauthier add one or several DAE-solvers to their run-time package and modify the code generator of the ACSL preprocessor to automatically invoke the DAE-solver when algebraic loops are detected in the model (maybe while issuing a warning message to the user indicating the problem).

Notice, however, that the described weakness is not unique to ACSL, but is one that all currently available CSSL-type languages have in common.

There are a few special-purpose simulation software systems that are worth mentioning. There exist some systems geared toward simulating chemical reaction dynamics, such as DIVA [25] and SpeedUp [32 and Chapter 9]. Contrary to the traditional CSSL-type systems, these systems employ DAE-solvers instead of ODE-solvers in their run-time software. DIVA has been successfully applied to real-time simulations of bulky chemical processes such as distillation columns.

There also exist special-purpose tools for the simulation of analog electronic circuitry. Most prevalent among those are the various dialects of Spice. As in the case of the chemical simulation systems, circuit simulators employ implicit integration techniques to get around the algebraic loop problem. Circuit simulators have been around for quite some time and could profit tremendously from a reimplementation using modern DAE-solvers in place of the fairly primitive Newton iteration schemes employed in currently available versions.

Furthermore, there exist special-purpose simulators for mechanical manipulators (robots). Also these systems are plagued by the same disease. Each constraint (coupling between neighboring limbs) introduces nasty algebraic loops among outputs of integrators. I shall not discuss these highly specialized simulators here in any further detail.

There exist many real-time training simulators for commercial airplanes, nuclear power plants, and some other complex industrial or military processes. These software systems are highly specialized, and a discussion of the solution techniques employed in their design does not contribute much to a general survey such as this.

There exist a few simulation languages geared toward truly combined continuous and discrete simulation, such as COSMOS [23] and SYSMOD [41]. Combined continuous/discrete simulation [6] requires more than just event handling. Such systems require at least waiting queues and enhanced capabilities for dealing with random numbers and distribution functions, but to be used comfortably, they also require mechanisms for process descriptions. Most of these systems, e.g., SIMAN [33] and SLAM [36], grew out of the discrete-event simulation world. Such systems offer only rudimentary facilities for continuous-system simulation. SYSMOD [41] evolved from the continuous simulation world and offers only a limited set of facilities for discrete-event simulation. COSMOS [23] is the only simulation system currently on the market that offers a fairly well-balanced palette of both continuous and discrete simulation capabilities. However, a more detailed discussion of these software systems is beyond the scope of this survey.

DESIRE [24] offers special facilities for modeling and efficiently simulating artificial neural networks and fuzzy control systems. For these types of applications, DESIRE is clearly the language of choice.

Finally, there exist special-purpose software systems for the qualitative description of continuous-time processes. The most prevalent among those systems is QSIM [26]. There also exist tools for mixed quantitative and qualitative simulation of continuous-time processes [12]. However, also these tools are too specialized to be discussed in more detail in a general survey such as this.

Although there exist still plenty of good reasons why special solutions may be needed for the simulation of special processes, ACSL has become the major workhorse for simulating effectively and efficiently large classes of continuous systems. ACSL has its largest customer base among control engineers.

The statement made in the previous paragraph is somewhat subjective. It is obviously influenced by my own exposure to and experience with the ACSL language. There exist several other simulation languages, such as DESIRE [24] and Simnon [17], with customer bases that are at least of the same order of magnitude. However, I have very good reasons for recommending ACSL, reasons that go beyond matters of personal preference and style. In the past, I have

used many simulation languages and usually gave them up after some time because they were not flexible enough. Whenever I wanted to model a type of system for which the software was not originally intended, I had to invent tricks over tricks to convince the software to do what I wanted it to do. ACSL is the first simulation language that I found that does not constrain me. When I recently decided to implement mixed quantitative and qualitative models in ACSL [12], I was able to achieve this goal quickly (with less than two weeks of work) and without a need to invent dirty tricks. I do not know of any other simulation language of which I could say the same.

1.3 MODELING SOFTWARE

Most simulation models coded in a CSSL-type language were still fairly short one decade ago with larger models usually requiring ad hoc solutions (mostly large and poorly maintainable Fortran programs). The situation has changed drastically by now. ACSL programs containing 10,000 lines of code are no longer a rarity. Unfortunately, ACSL's model description capabilities, although far superior to Fortran, are still not adequate for dealing with such large-scale applications. Such applications call for the object-oriented (OO) programming paradigm. Simple subsystems should be describable as atomic objects. Objects can be interconnected to form ever larger molecular objects.

CSSL-type macros do not provide for an adequate mechanism to encapsulate objects. This can be demonstrated by means of the simple circuit problem shown in Figure 1.1. A block diagram of this simple electrical circuit is shown in Figure 1.2. This circuit can be encoded in the following ACSL program:

```
Program Circuit

    Constant R1 = 100.0, R2 = 20.0, C = 0.1E - 6, L = 1.5E - 3
    Constant tmx = 0.01

    u0 = f(t)

    iC = uR1/R1
    uR2 = R2 * iL
    uC = INTEG(iC/C, 0.0)
    iL = INTEG(uL/L, 0.0)

    uR1 = u0 - uC
    uL = u0 - uR2
    i0 = iC + iL

    term(t.ge.tmx)

End
```

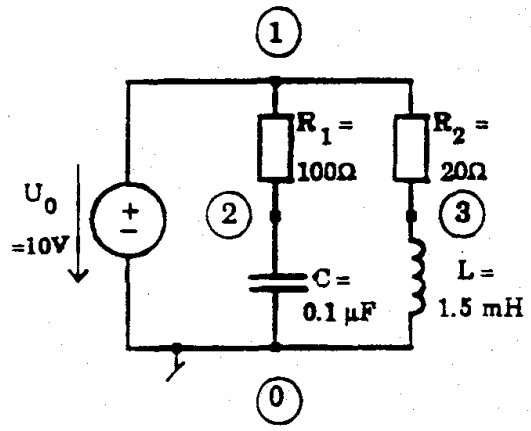

Figure 1.1 Simple electrical circuit.

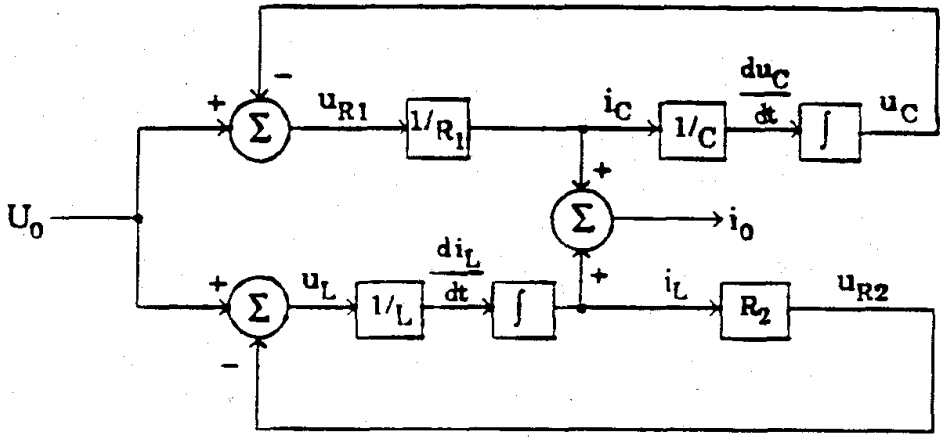

Figure 1.2 Block diagram of electrical circuit.

The first equation describes the input to the circuit, i.e., the voltage source; the next four equations describe the circuit elements themselves, i.e., the objects, whereas the last three equations describe the circuit topology, i.e., the interconnections between the objects (Kirchhoff's laws).

Notice that this circuit contains two objects of type resistor, and, yet, the equations describing these two objects look different. In the case of resistor R_1, the voltage drop across the resistor seems to "cause" a current flow, whereas in the case of resistor R_2, the current flow through the resistor seems to "cause" a voltage drop.

Clearly, the observed "causalities" are purely computational and have nothing to do with the physics of the problem. This example demonstrates that, although an equation sorter is a step in the right direction, it is obviously insufficient for truly modular object-oriented continuous-system modeling [13].

For this purpose, we require more advanced formula manipulation capabilities [11].

It is rather inconvenient that the ACSL user must determine the (numerically) correct causalities of dissipative elements, or more generally, the causalities of all energy transducers (transformers exhibit exactly the same problem as resistors). It would be much nicer if objects, such as a resistor, could be described once and for all in terms of their *physical* properties and their interactions with the environment. In the case of the resistor, such an approach would call for a description of the resistor itself (Ohm's law) and a separate description of how this equation interacts with other equations of the neighboring components (Kirchhoff's laws).

However, object-oriented continuous-system modeling [13] is much more than just a matter of convenience. State-space models suggest that each state variable changes with time according to some law that is expressed in the corresponding state equation. But why does this happen? The voltage across a capacitor does not change with time unless it has a good reason for doing so. Physics is strictly a matter of trade. The only tradable goods are mass, energy, and momentum. Consequently, it would be much safer if the modeling environment were to enable the user to formulate mass balances and energy balances rather than state equations. If a state equation is formulated incorrectly, a CSSL-type simulation language will happily accept the incorrect equation and trade it for beautiful looking multicolored graphs that may even seem plausible [9].

The modeling language Dymola [8,11,15] incorporates these concepts. In Dymola, a resistor can be described as follows:

```
model type resistor
    cut WireA(Va/i), WireB(Vb/ - i)
    main path P < WireA - WireB >
    local u
    parameter R = 1.0
        u = Va - Vb
        u = R * i
end
```

Ohm's law is described in the usual way. It involves the parameter R, which has a default value of 1.0, the local variable u, and the terminal variable i. The *cut* and *path* declarations are used to describe the interface to the outside world. Additional equations are formulated to specify the relations between the local variables and the terminal variables.

Of course, the chosen approach also calls for a general mechanism to describe the couplings between different interconnected objects. In Dymola, the above circuit can, for example, be represented as follows:

```
model circuit
    submodel (vsource) U0
    submodel (resistor) R1 (R = 100.0), R2 (R = 20.0)
    submodel (capacitor) C(C = 0.1E-6)
    submodel (inductor) L(L = 1.5E-3)
    submodel Common
    node n0, n1, n2, n3
    input u
    output y1, y2

    connect Common at   n0,
            U0      from n0 to n1,
            R1      from n1 to n2,
            C       from n2 to n0,
            R3      from n1 to n3,
            L       from n3 to n0

    U0.V = u
    y1 = C.u
    y2 = L.i
end
```

The *submodel* declaration instantiates objects from classes. For example, two objects of type *resistor* are instantiated, one named $R1$ with a parameter value of $R = 100.0\ \Omega$ and the other named $R2$ with a parameter value of $R = 20.0\ \Omega$. The *connect* statement is used to describe the interconnections between objects. Notice that the connecting equations (Kirchhoff's laws) are not explicitly formulated at all. They are automatically generated at compile time from the topological description of the interconnections.

Upon entering the model, Dymola immediately instantiates all submodels (objects) from the model types (classes). It then extracts the formulated equations from these objects and expands them with the coupling equations that are being generated from the description of the interconnections between objects. For the above example, the result of this operation is the following.

```
U0      V = Vb - Va
R1      u = Va - Vb
        u = R * i
R2      u = Va - Vb
        u = R * i
C       u = Va - Vb
        C * der(u) = i
L       u = Va - Vb
        L * der(i) = u
```

Common	V = 0
circuit	U0.V = u
	y1 = C.u
	y2 = L.i
	R1.Vb = C.Va
	C.i = R1.i
	R.1Va = R2.Va
	U0.Vb = R1.Va
	R2.i + R1.i = U0.i
	R2.Vb = L.Va
	L.i = R2.i
	C.Vb = L.Vb
	U0.Va = C.Vb
	Common.V = U0.Va

The first 10 of these equations are extracted from the submodels. The next 3 equations are extracted from the circuit model. The last 10 equations represent Kirchhoff's laws. These equations are automatically generated from the connect statements that describe the interconnections between the objects.

The *partition* command in Dymola solves the causality assignment problem [11]. It also eliminates trivial equations of the type

$$a = b \tag{1.3}$$

The result of this operation is as follows:

Common	[L.Vb] = 0
U0	circuit.u = [R2.Va] − L.Vb
C	u = [Va] − L.Vb
R1	[u] = R2.Va − C.Va
	u = R * [i]
C	C * [der(u)] = R1.i
R2	[u] = R * L.i
	u = Va − [L.Va]
L	[u] = Va − Vb
	L * [der(i)] = u
circuit	L.i + R1.i = [U0.i]
	[y1] = C.u
	[y2] = L.i

In each equation, the variable to be solved for is marked by square brackets. Notice the different causalities for the two resistors.

At this point, further formula manipulation can be used to solve the equations in order to generate a state-space model. Dymola has rules about the inverse of

certain functions and handles the case of several linear occurrences of the unknown variable. Solving the following equation for x

$$\exp\left(a + \sin\left(\frac{[x]}{b} + c[x] - d\right)(\exp(e) + 1)\right)^2 - f = 2g \tag{1.4}$$

gives the result

$$x = \frac{\arcsin((\ln(\mathrm{sqrt}(2g + f)) - a)/(\exp(e) + 1)) + d}{1/b + c} \tag{1.5}$$

For the above circuit example, the result of the command

```
> output solved equations
```

is as follows:

```
Common    L.Vb = 0
UO        R2.Va = circuit.u + L.Vb
C         Va = u + L.Vb
R1        u = R2.Va - C.Va
          i = u/R
C         der(u) = R1.i/C
R2        u = R * L.i
          L.Va = Va - u
L         u = Va - Vb
          der(i) = u/L
circuit   UO.i = L.i + R1.i
          y1 = C.u
          y2 = L.i
```

Finally, the state-space model can be automatically encoded as a text file in any one of a series of simulation languages. For example, the commands

```
> language acsl
> output program
```

automatically generate the following ACSL program:

```
"------------------------------------------------------------"
" **************ACSL model generated by Dymola. **************"
"------------------------------------------------------------"

PROGRAM circuit

    INITIAL

    CONSTANT ...
        R1XR = 100.0, C = 0.1E-6, L = 1.5E-3, ...
        R2XR = 20.0
```

```
    CONSTANT tmax = 0.01
    CONSTANT ...
        u = 10.0
    END $ "of INITIAL"
    DYNAMIC
        DERIVATIVE
"-------------------------Submodel: Common"
            LXVb = 0
"-------------------------Submodel: UO"
            R2XVa = u + LXVb
"-------------------------Submodel: C"
            CXVa = CXu + LXVb
"-------------------------Submodel: R1"
            R1Xu = R2XVa - CXVa
            R1Xi = R1Xu/R1XR
"-------------------------Submodel: C"
            CXu = INTEG(R1Xi/C, 0.0)
"-------------------------Submodel: R2"
            R2Xu = R2XR * LXi
            LXVa = R2XVa - R2Xu
"-------------------------Submodel: L"
            LXu = LXVa - LXVb
            LXi = INTEG(LXu/L, 0.0)
"-------------------------Submodel: circuit"
            U0Xi = LXi + R1Xi
            y1 = CXu
            y2 = LXi
        END $ "of DERIVATIVE"
        termt (t.ge.tmx)
    END $ "of DYNAMIC"
END $ "of PROGRAM"
```

Notice that Dymola is not a simulation language in its own right. Dymola can be viewed as a sophisticated *macro processor* because it can be used as a frontend to a simulation language and thereby (among other things) assumes the role of its macro processor. Dymola can also be viewed as a *model generator* because it can generate models for a variety of different simulation languages. The cur-

rently supported languages are ACSL [31], DESIRE [24], and Simnon [17]. However, the most adequate interpretation is to view Dymola as a *modeling language*. Dymola has been designed to facilitate the object-oriented formulation of models of complex continuous systems.

Dymola offers the following features:

1. Modular formulation of atomic continuous-system models (objects)
2. Hierarchical composition and interconnection of atomic and molecular objects into objects of ever-increasing complexity with automatic generation of all coupling equations
3. Hierarchical data structures for connections (wires can be grouped into cables, and cables can be grouped into trunks)
4. Support of both across and through variables in connections (the values of all across variables connected to a node are the same, whereas the values of all through variables connected to a node add up to zero) with automatic generation of all coupling equations
5. Object instantiation (multiple objects can be instantiated from a single class)
6. Class inheritance (subclasses can inherit declarations of variables and equations from their parent classes)
7. Equation sorting and solving (equations automatically sorted into an executable sequence, and each equation automatically solved for the correct variable)
8. Index reduction (algebraic loops among state variables automatically reduced to algebraic loops among auxiliary variables by means of symbolic differentiation)
9. Linear algebraic loop solving (algebraic loops isolated and, if linear in the involved variables, automatically solved by means of formula manipulation)
10. Nonlinear function inversion (analytic functions automatically inverted during equation solving as needed)

Although Dymola was the first modeling language on the market, it is no longer alone. Omola [29] has been recently added to the language zoo. Omola's functionality is basically equivalent to that of Dymola. Omola offers features similar to those available in Dymola, except for linear algebraic loop solution, a feature that is not currently offered in Omola. Contrary to Dymola which translates models into ODE form (reduction to index 0), Omola translates models into DAE form (reduction to index 1). Dymola supports the generation of simulation programs in a variety of simulation languages, such as ACSL, whereas Omola comes with its own underlying simulation system, OmSim. OmSim contains several DAE-solvers. Dymola is now a commercially available product, whereas Omola and OmSim are still experimental systems.

1.4 GRAPHICAL MODEL EDITOR

Traditionally, models were always entered as text files. However, it is legitimate to ask whether this is the most convenient way to encode models. Atomic models are described by sets of equations, and there is nothing wrong with encoding those as text files. After all, atomic models are usually quite small anyway. But would it not be more convenient if each object could be associated with an icon on the screen (to be designed interactively by use of a so-called *icon editor*) and interconnections between objects could be described by means of graphical connections between icons?

Graphical model editors were slow to come. TESS [39] offered a so-called network editor (for discrete-event models) in 1985, but the networks were flat. The network editor did not come with an icon editor to encapsulate subnetworks as molecular objects. The first commercially available graphical model editors for continuous systems were EASE+ [20], a generic model editor that comes with a programmable target interface, i.e., can be used to generate models for a variety of simulation languages, System-Build [22], a graphical simulation language added to the MATRIX_x [21] software, and a model editor incorporated in Boeing's EASY-5 simulation software [4]. Meanwhile, new graphical model editors are thrown onto the simulation market monthly. Model-C is a competitor of System-Build added to the CTRL-C [40] software; SimuLink was recently added to Matlab [28]; and there also exists meanwhile a block diagram editor for ACSL [31] called ProtoBlock. Simnon [17] offers a block diagram editor called ISEE-Simnon.

Why this sudden avalanche of new products? In the past, the development of graphics software was hampered by inadequate hardware and operating system support and a heavy hardware dependency. Products, such as EASE+, had to be reimplemented from scratch for each new hardware platform to which they were ported. Consequently, these systems were very expensive. Moreover, pixel graphics is bus-intensive. It is not meaningful to supply a fancy graphical model editor for a terminal that is connected to a main frame computer by a 1200-baud modem line. Only the availability of ultrafast and cheap Engineering Workstations made graphical model editors attractive. The very recent standardization of graphics software with X for the low-level functions and Open Look and Motif for the higher-level functions make the development of new graphics systems much simpler, faster, and thereby cheaper. Moreover, X and its widget toolboxes are hardware-independent, i.e., porting the software once developed to a new platform has become a relatively easy task.

Evidently there is a market for graphical model editors. A company that does not offer such a product is no longer competitive on the market. Consequently, new products are rushed out onto the market as fast as the simulation software producers can throw them together.

Although most of these new products look slick and professional, quality has suffered a bit under the hurry. The major problem is that all of these systems are simple block diagram editors, i.e., they are not truly modular. They allow one to draw on the screen a circuit as shown in Figure 1.2, but not one as shown in Figure 1.1. (For electronic circuits, there do exist so-called schematic capture programs, such as WorkView [42], that can be used to draw schematics on the screen and that then generate Spice programs, but these are special-purpose products for electronic circuits only.) Most of the block diagram editors meanwhile offer an icon editor, i.e., they are at least hierarchical in nature. However, if two of these blocks share 10 variables, 10 connecting lines must be drawn between their icons. There is no support for hierarchical data structures. Also, connections are strictly unidirectional, i.e., the user must specify which of the blocks is responsible for computing each of the shared variables.

Many of the graphical model editors on the market as of today are self-contained. They come with a very simple rudimentary simulation system integrated into the software. The block diagram is simply interpreted. In some cases, the model equations are numerically solved by a simple forward-Euler algorithm; other products offer at least a fourth-order Runge–Kutta algorithm. Years of development that went into today's commercial simulation software were simply thrown away to be able to offer a self-contained product that can be sold cheaply and does not require collaboration with a competitor company. The products are proudly presented at conference exhibitions, and all vendors demonstrate happily that they can solve the Van der Pol oscillator problem on them.

Some designers were a little wiser and decided not to reinvent the wheel. Instead of making their system self-contained, they offer a frontend to an existing simulation language, such as ACSL, thereby inheriting years of engineering that went into the design of robust simulation software. Molecular objects are translated into simulation macros. This approach is better, but still not good enough. The problem is that all the shortcomings of the macro solution are inherited. As explained earlier, macros are not truly modular.

What should have been done was to design the graphical model editor as a frontend to a modeling language, such as Dymola or Omola, rather than as a frontend to a simulation language. With this approach, composition knowledge could be expressed in terms of so-called *stylized block diagrams* [8]. Figure 1.3 shows a stylized block diagram of our simple electrical circuit. Each interconnection between blocks may represent multiple variables. In the above example, each interconnection represents exactly two variables, one of the across-type (the potential) and one of the through-type (the current). Connections between blocks are nondirectional, i.e., the user is relieved of the burden of having to solve the causality assignment problem manually. The code generated from the stylized block diagram is the previously shown Dymola circuit model. Of course, blocks in a stylized block diagram do not have to be square boxes. It is

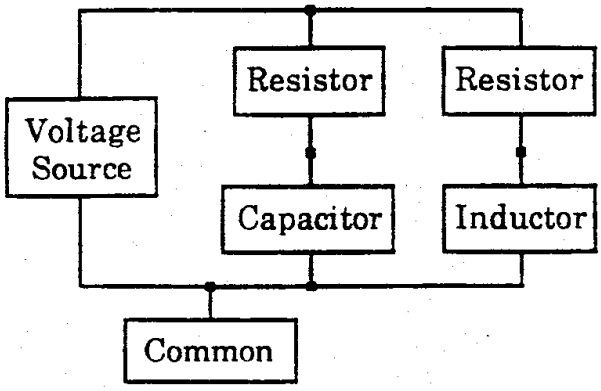

Figure 1.3 Stylized block diagram of electrical circuit.

perfectly compatible with the concept of a stylized block diagram to offer an icon editor for the custom design of blocks. With this additional facility, the circuit diagram of Figure 1.1 can be interpreted as a stylized block diagram.

Decomposition knowledge and taxonomic knowledge are encoded in a separate window by use of a so-called *system entity structure* [43]. Figure 1.4 depicts a cable reel system for the deployment of deep-sea fiber-optic communication cables.

A system entity structure for the cable reel system is shown in Figure 1.5. A decomposition that is indicated by a single vertical bar denotes a decomposition into parts. Each such decomposition is associated with a stylized block

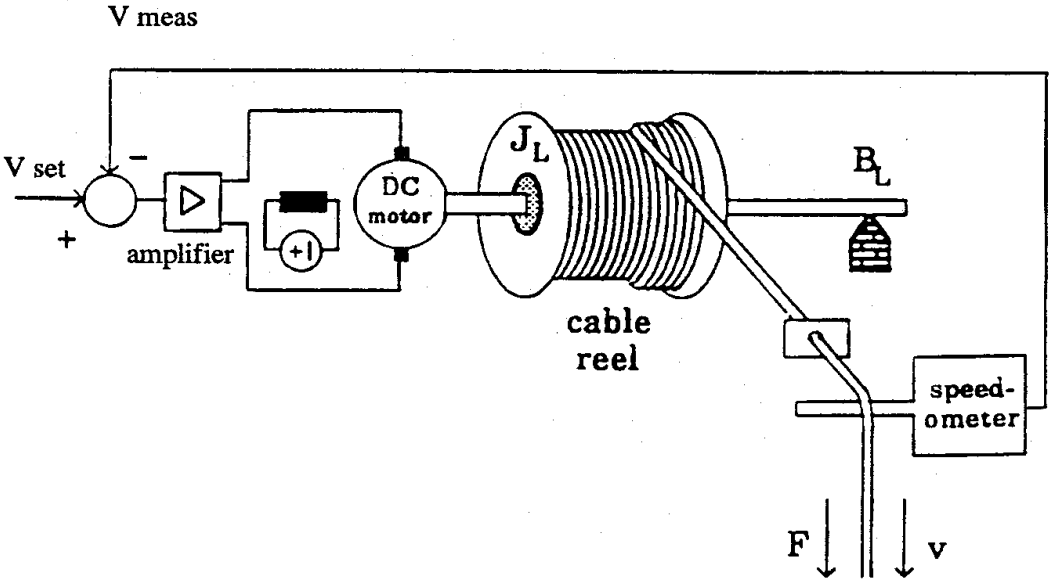

Figure 1.4 Functional diagram of a cable reel system.

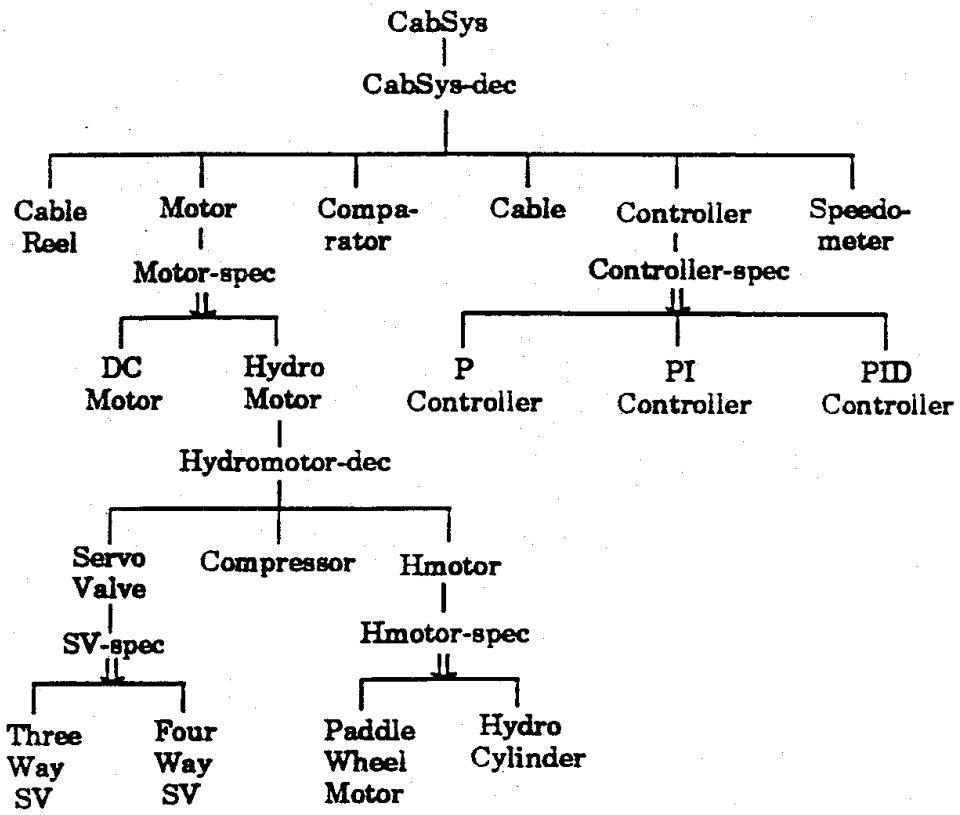

Figure 1.5 System entity structure of cable reel system.

diagram. Double-clicking on the CabSys-dec object opens up a new window that shows the stylized block diagram of Figure 1.6. Stylized block diagrams are hierarchical. For example, the motor block of Figure 1.6 contains another stylized block diagram. Double-clicking on the Hydromotor-dec object opens up another

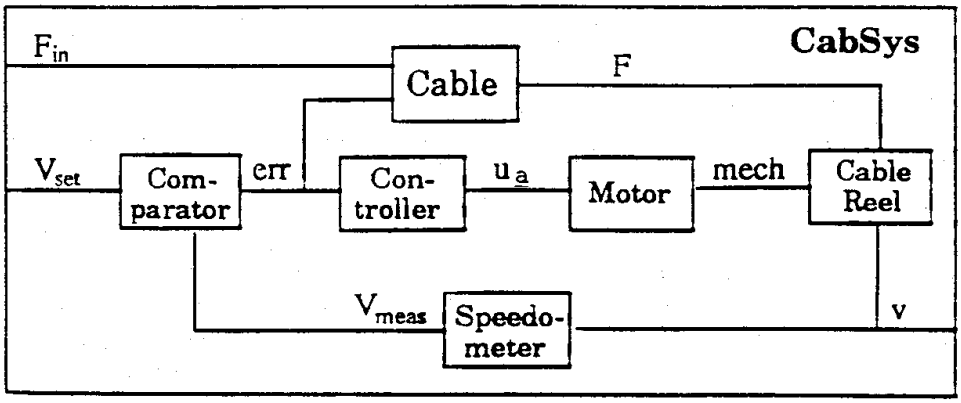

Figure 1.6 Stylized block diagram of cable reel system.

window that shows the stylized block diagram of Figure 1.7. A decomposition that is indicated by a double vertical bar denotes a decomposition into variants, a so-called specialization. For example, the motor is specialized into either a DC motor or a hydraulic motor. Double-clicking on the Motor-spec object opens up a new window with a rule-base editor to describe under what conditions the DC motor should be chosen and when the hydraulic motor should be selected.

Unfortunately, the above described system is currently vaporware. There exists a graphical frontend for Dymola, called LICS [16], but LICS was coded 10 years ago, at a time when neither the computer hardware nor the operating software were ripe to support such a development. LICS was developed on a VAX/11-780. It was controlled by a mouse with a home-built interface! While LICS was running, all other log-ins to the VAX were disabled. LICS was later ported over to a Silicon Graphics IRIS Workstation, and its name was changed to HIBLIZ [18]. However, the software was ported with minimal changes, and also the new version is heavily hardware-dependent. LICS (and HIBLIZ) offer stylized block diagrams (without icon editor) and translate into Dymola. Hierarchical decomposition is supported by means of a zoom/pan feature rather than the system entity structure approach that has been advocated in this chapter. A complete reimplementation of this software is needed. Using X and Motif, this should be a much simpler task than the development of the original system.

Of course, icon editors can be used not only to custom-design boxes, but also to custom-design connections between boxes. In this way, bond graph [10] editors also could be designed as special versions of stylized block diagrams, in the same way as the previously shown circuit diagram can be interpreted as a special type of a stylized block diagram. There are currently a few bond graph editors on the market; however, those that I have had an opportunity to experience and work with are all built in a fairly amateurish fashion employing ad hoc programming techniques, i.e., they were designed in total ignorance of basic computer science principles. A professionally built bond graph editor could be a very valuable element in an integrated modeling and simulation environment.

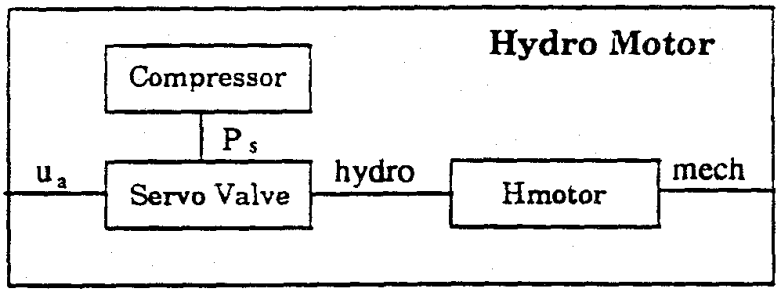

Figure 1.7 Stylized block diagram of the hydromotor.

1.5 SIMULATION ANIMATION

During simulation, users would like to see what is going on. Traditional CSSL-type simulation software does not offer any run-time display capabilities. The simulation is run first, and thereafter, the results can be viewed. This approach makes a lot of sense in a main frame/terminal environment with limited resources both in terms of CPU cycles and memory allocation. Today, this approach is obviously outdated. Some simulation systems, such as DESIRE [24], offer, at least, a run-time display. A few variables can be selected that are displayed graphically while the simulation run is executed. However, more can be done and has been achieved. In the context of graphical model editors, some systems offer a "probe" feature that allows the user to double-click on any connection in the block diagram during the simulation run with the effect that a new window pops up in which the selected variable is displayed. Some systems (such as SimuLink) offer an "oscilloscope icon" that can be attached to any block diagram connection for preselected run-time displays. A "floating oscilloscope" can be used to select an additional signal to be displayed on the spot.

However, even this facility is kind of primitive. It would be nice to be able to interactively design a cockpit of an aircraft with meters, needles, and gauges, and associate these with signals of the simulation software. Such a system has been built on top of KEE (running on Symbolics computers). However, the types of graphical elements needed to populate the animation depend strongly on the application area. Chemical process control requires quite different types of animations involving pictures of distillation columns, valves, and pipes. Simnon [17] offers a real-time simulator for process control operator training, called Simosa. Quite obviously, each application area has its special needs, and, yet, recent advances in computer graphics make it now feasible to develop quite general graphics systems that are flexible enough to custom-design screens for diverse types of applications. However, these advances are so recent that simulation animators that are currently on the market do not exploit them yet. Consequently, all commercially available simulation animators are invariably outdated. A new generation of simulation animators will quite certainly replace them in the near future.

1.6 POSTANALYSIS SOFTWARE

Besides the on-line simulation animator, there is still a place for off-line analysis of simulation results. After the simulation has been performed, users like to perform statistical analysis on simulation data, apply Fast Fourier Transforms (FFTs) to simulation trajectories, redisplay some of the curves for a closer look, etc. Graphical postanalysis of simulation results is a feature that traditional

CSSL-type simulation software has offered for many years. However, the post-analysis support offered was usually limited to the display of trajectory behavior. This is clearly insufficient.

Already a few years ago, interfaces were created between ACSL on the one hand and CTRL-C [40] and Matlab [28] on the other. This is how I run *all* my ACSL simulations. I never make use of the so-called "ACSL run-time commands" (which are not run-time commands in any true sense, but rather pre-analysis and postanalysis commands—ACSL does not provide for any interactivity during the execution of a simulation run). Instead, I kick off the ACSL simulation from within either the CTRL-C or Matlab environment and import the simulation trajectories back into the environment in the form of vectors and matrices. These simulation data can then be flexibly manipulated in many ways. It is possible to look at a subset of data only (zoom), to apply a logarithmic transformation on the data without rerunning the simulation, to superpose curves in an arbitrary fashion, to apply an FFT transform to the data, to perform statistical analysis on them, etc.

It turns out that flexible postanalysis software must make provisions for a full-fledged high-level programming language (such as Matlab or CTRL-C) because it is impossible to foresee all meaningful postanalysis features and cast them once and for all into a fixed set of precoded postanalysis operations to be selected from a menu.

The postanalysis package should be intimately tied in with the *environment language* (to be discussed later in this chapter). CTRL-C [40], Matlab [28], and MATRIX_x [21] are excellent examples of powerful and flexible, yet user-friendly environment languages that can host the postanalysis software.

1.7 DOMAIN MODEL LIBRARIES

Although the previously discussed modeling software provides the capabilities for developing object-oriented modular models of arbitrary continuous-time systems, the end user does not care to develop models for the basic modeling components of his application area, such as transistors, compressors, robot arms, turbines, etc., on his own.

In addition to the modeling software itself, which is domain-independent, an integrated modeling and simulation environment should provide for domain-dependent model libraries.

A modeling/simulation environment for electrical circuit simulation should offer basic models of simple passive components (resistors, capacitors, inductors, transformers), of active components (voltage and current sources), but also of more complex components such as transistors (both BJTs and FETs) and diodes. Transistor models can be fairly sophisticated and quite complex [8]. These models make programs, such as Spice, powerful and valuable.

A modeling/simulation environment for electric power plants should contain basic models of pumps, turbines, heat exchangers, pipes, etc. MMS [19] is a modeling/simulation environment specialized for this purpose. MMS is currently available in two versions. One version contains a library of ACSL macros; the other contains EASY-5 modules. Thus, MMS was designed and implemented as a frontend to a simulation language. However, this decision places many unnecessary constraints on the MMS user. For example, MMS modules are characterized as either resistive modules or storage-type modules. Resistive modules can only be connected to storage-type modules, and vice versa. This rule encapsulates the fact that the user is indirectly responsible for solving the causality assignment problem. MMS would be much more flexible if it were implemented as a frontend to a modeling language rather than as a frontend to a simulation language. One of my students is currently reimplementing MMS in Dymola [45].

A modeling/simulation environment for thermal heating systems should contain basic models of rooms, walls, windows, rockbeds, sunspaces, trompe walls, etc. The end user should be able to make models of buildings by putting these basic models together and should not have to worry about the equations that describe these basic models themselves. Commercial products, such as CALPAS 3 [3] and DOE2 [1], are successful, not because they "know" the thermodynamic equations that describe conductive, convective, and radiative heat flow, but because they protect the end user from having to apply these equations directly. The end user is being offered a set of fairly sophisticated modules from which models of entire buildings can be thrown together within a few hours. Another student of mine is currently implementing a new system of this type on the basis of Dymola [45].

A modeling/simulation environment for chemical process modeling should contain basic models of different types of separation columns (distillation columns, stripping columns, rendering columns), of compressors and condensors, of vapor/liquid separators and oil/water separators, etc. ASCEND [35] and DESIGN-KIT [37] are modeling/simulation environments that have been designed for this purpose. These systems offer a fairly nice touch-and-feel, they are fully object-oriented, and they are carefully designed. ASCEND has been mostly used for steady-state analysis, but the language design is not limited to this type of application. DESIGN-KIT has a well-designed database interface. Both systems are tailored toward chemical process engineering, i.e., they do not make a clear separation between the domain-independent modeling language and the domain-dependent model library. Thus, they are a little less general than Dymola or Omola. I have currently two students working on a Dymola-based implementation of this type of model library. One of my students is working on a bond graph model of a distillation column [44]; the other student is working on a bond graph model of an oxygen production plant for planet Mars.

1.8 DOMAIN DATABASES

However, even availability of a domain model library is not sufficient. Models cannot be simulated with equations alone; they also require data.

A BJT model in an electronic circuit simulator contains more than 50 parameters [8]. Collections of well-matched sets of parameter values for various types of commercially available transistors may be equally if not more valuable than the transistor model itself. There are companies who sell such databases as a separate product [30]. The end user would like to describe a transistor used in his circuit simply by providing its part number. Spice supports such a feature.

An electric power plant simulator relies heavily on data characterizing various types of valves. It also depends on steam tables, and many other types of data. These data items should be physically separated from the models that capture the structural relations among variables, i.e., the model equations.

A chemical process plant simulator should have access to tables of enthalpies, evaporation temperatures, etc. Thick books have been written that are full of such tables and other data [34]. A process plant simulator should be able to access a computerized version of Perry's handbook stored in an SQL database [14], and the modeling language should provide for mechanisms to access such a database.

In this respect, all of today's modeling languages are deficient. Data are usually hard-coded into the models that use them. Neither Dymola nor Omola offer special mechanisms for accessing domain databases. However, the problem can be overcome by shifting the responsibility down to the underlying simulation software. Many SQL databases are Fortran callable. In ACSL, calls to Fortran subroutines can be encapsulated in macros that can then be referenced from within the Dymola program.

TESS offers a built-in database, but TESS is geared toward discrete-event systems, and moreover, the database is not particularly well-suited to hold permanent data. It is important to distinguish between sharable read-only databases (e.g., to store Perry's handbook) offering slow data storage but fast data retrieval, and nonsharable read-and-write databases for storing simulation trajectories, note book files, etc., offering fast data storage but slow data retrieval. The database provided as part of TESS is of the latter kind.

1.9 MODEL IDENTIFICATION AND PARAMETER ESTIMATION

Even the most complete model database cannot provide numerical values for all parameters. For example, in a mechanical system, numerical values of masses, inertias, and spring constants are fairly easy to come by, but numerical values for friction constants are almost impossible to obtain. Thus, a decent modeling

environment should offer facilities for estimating a subset of the model parameters from measurement data.

Physical parameters can be estimated by means of nonlinear programming packages. Such facilities are already available for ACSL, implemented in the form of two separate optional software tools called OPTDES and SimuSolv. OPTDES is most frequently used for mechanical and control systems, whereas SimuSolv is mostly used for chemical process engineering and pharmacokinetics. Matlab also offers two optimization toolboxes: a nonlinear programming toolbox (that can be used together with SimuLink) and a system identification toolbox for the computation of maximum likelihood estimators of linear systems.

Data to be identified often include statistical parameters of distribution density functions. There exist special programs on the market for just that purpose, e.g., UniFit II [27]. It would be very useful if UniFit II could also be invoked as a Matlab toolbox.

1.10 THE ENVIRONMENT LANGUAGE

At this point, it should be discussed how the various programs that were described in the previous sections of this chapter fit together. We need some sort of "operating system" that connects all these programs. However, traditional operating systems are concerned with file handling. For our purposes, this is too inconvenient. What we need is a system that can manipulate data structures and, most importantly, matrices and vectors. Such systems have been developed and are on the market for a few years. Good candidates for the environment language are CTRL-C [40], Matlab [28], and MATRIX$_x$ [21]. These three systems are very similar. They are easy to use. Their touch-and-feel is that of a comfortable pocket calculator that operates on double precision complex matrices rather than scalars. A high-level programming language to manipulate these matrices was also added. All of these systems offer excellent graphics capabilities for viewing data interactively. Some of these systems offer special-purpose toolboxes for such tasks as statistical analysis and parameter estimation. All three systems offer capabilities for simulating nonlinear models; all of them also offer graphics frontends (block diagram languages).

TESS [39] offers a much more primitive environment language. Its syntax is that of an adventure game: a verb followed by a noun, such as: "build network," or "graph facility," or "report rule." TESS understands four different verbs and about eight different nouns. Most combinations of a verb and a noun form a legal sentence. Each legal sentence invokes a particular program (language) with its own syntax and semantics. For example: "build network" invokes the graphical model editor, and "build icon" invokes the icon editor.

Although this environment language provides for a loose framework linking the various programs that are part of the TESS software suite together, this does

not solve the real problems. The different modules (programs) communicate with each other, and the user should have the possibility to exert some control over this interaction. A true environment language, such as Matlab, will provide for the necessary flexibility; the environment language offered in TESS does not.

1.11 THE SOFTWARE ARCHITECTURE

There exist several ways the various software tools in the integrated modeling and simulation environment can interact with each other.

TESS [39] employs a *database architecture*. It is shown in Figure 1.8. In the center of this architecture is the SDL relational database [38]. Built around this database are the various modules that belong to the TESS software suite, such as the network editor, the icon editor, the SLAM simulation language, the postsimulation animator, and the postsimulation statistical analysis program. The different modules share a common core of Fortran service routines. All communications between the different modules go through the database. At the outskirt of the software, the user is protected from having to call each of these programs separately by the TESS environment language.

This architecture is very easy to realize. By demanding that all communications between modules go through the database, each module can be limited to offer exactly three interfaces: (i) a standardized interface to the relational database (ii) another standardized interface to the environment language, and (iii) a nonstandardized interface to the user. Notice that the TESS language does not

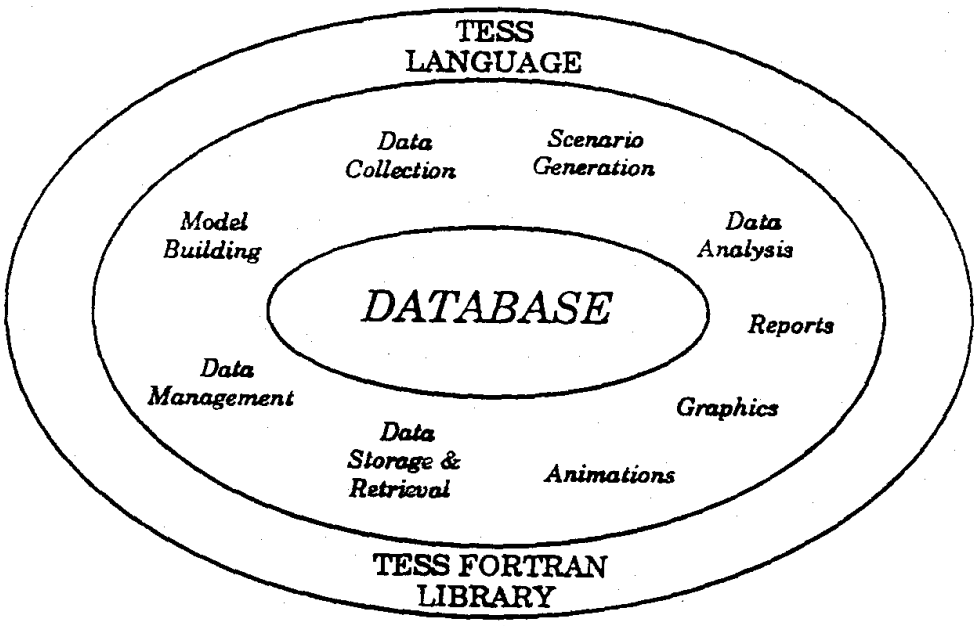

Figure 1.8 TESS architecture.

protect the user from understanding each of the modules individually—it only protects him from understanding the (VMS) operating system.

Although TESS (and its competitor shell, CINEMA) are fairly primitive from the perspective of modern window management systems and CASE architectures, these two discrete-event modeling and simulation environments are interesting from a historical perspective. They clearly represent pioneering efforts into the design of a new generation of modeling and simulation software environments.

Another approach to be considered is the more layered architecture shown in Figure 1.9. In this architecture, the user interacts with a (Matlab-like) environment language that is intimately linked to the read-and-write database (in Matlab, this database is simply the stack). The end user calls the graphical model editor through the environment language. The icon editor and the model library builder can be called in the same fashion, and also the animation editor can be called through the environment language. Once a model has been created, it is compiled down into the textual modeling language (e.g., Omola or Dymola). Library module calls are resolved through the link between the icon library and the model library. The textual OO-model is then compiled further into a simulation program. At this point, the object-orientation is lost. The resulting simulation program is a monolithic unstructured program that is optimized for execution

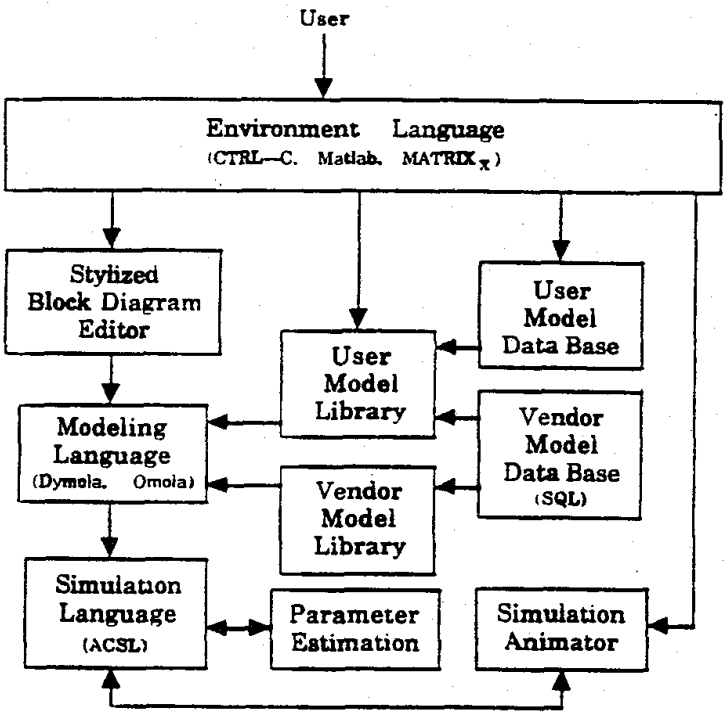

Figure 1.9 Layered environment architecture.

speed, not for human readability. The access to the model database can be resolved either at compile time (by extracting the necessary data items once and for all from the read-only SQL database and incorporating them directly into the generated simulation program), or at run-time (by creating a link between the simulation program and the SQL database). The former approach is more cumbersome, but may lead to more computationally efficient run-time code.

The interaction between the simulation program and the animation program should probably be direct, not going through the database. Again, this decision is based on efficiency considerations. However, it is not clear at this point whether it is better to have the simulation program call the animation program like a subroutine, whether the animation program should be in charge and call the simulation program whenever it needs new data, or whether both programs should be implemented as separate tasks with a standardized means of intertask communication (rendezvous). The latter approach may provide for the most modular solution, and in addition, it allows one to run the simulation task and the animation task on two separate processors. These two tasks have to be executed simultaneously, and both are computation-intensive.

A system as depicted in Figure 1.9 does not currently exist for any application domain. However, modern window management and CASE software have made the implementation of such a system feasible and affordable. I am convinced that several such systems for different application domains will appear on the simulation software market within the next few years.

Although the field of computer simulation has matured quite a bit over the past decade, modeling and simulation software design can still make for an exciting and rewarding research area.

1.12 CONCLUSIONS

In this chapter, a new architecture for an integrated continuous-system modeling and simulation environment has been presented. An overview of currently available software tools that might be used as components of such an integrated software environment was also given.

This is the third survey of continuous-system simulation software that I wrote. My first survey (1975) focused on concepts and components of continuous-system simulation languages; the second survey (1983) compared features of an extensive list of contemporary continuous-system simulation systems. The third survey (1992) focuses on concepts and components of integrated continuous-system modeling and simulation environments. Finally, I hope to be able to write a fourth survey around the turn of the century in which I intend to compare features of a palette of contemporary modeling and simulation environments.

My software surveys are usually quite critical. I do not mind stepping on various vendors' toes—after all, they are only virtual toes. In the past, my critiques

were usually followed quickly by new releases of several of the critiqued products, releases that removed many of the shortcomings that were discussed in my reviews. Whether this is purely accidental, I do not dare to say. However, if it is not accidental, I believe that my software critiques have made a significant contribution to the advancement of simulation science.

This survey has, quite noticeably, a somewhat personal touch. It could not possibly be totally impartial or objective. The survey draws heavily from my own personal experiences with the various software tools. This is the reason why I decided to write this chapter in the active voice rather than in the more traditional passive voice. An account of personal experiences reported in passive voice reads strangely, and use of the active voice softens my statements down since the reader is constantly reminded of the subjective nature of the account. However, to those readers (except for software vendors) who might feel offended by my personal writing style, I present my sincere and humble apologies. I only try to serve my scientific community in the way I can do it best.

REFERENCES

1. *Micro-DOE2 User Manual*, Acrosoft International, Inc., Denver, CO, 1987.
2. Augustin, D. C., Fineberg, M. S., Johnson, B. B., Linebarger, R. N., Sansom, F. J., and Strauss, J. C., The SCi continuous system simulation language (CSSL), *Simulation*, *9*, 281–303 (1967).
3. *CALPAS 3 User Manual*, Berkeley Solar Group, Berkeley, CA, 1982.
4. *EASY5/W—User's Manual*, Boeing Computer Services, Engineering Technology Applications (ETA) Division, Seattle, WA, 1988.
5. Brenan, K. E., Campbell, S. L., and Petzold, L. R., *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, North-Holland, Amsterdam, 1989.
6. Cellier, F. E., *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D. Dissertation, Diss ETH No. 6483, Swiss Federal Institute of Technology, Zürich, 1979.
7. Cellier, F. E., Simulation software: Today and tomorrow, *Proceedings IMACS Symposium on Simulation in Engineering Sciences*, (J. Burger and Y. Jarny, eds.), Nantes, France, May 9–11, 1983, pp. 426–442.
8. Cellier, F. E., *Continuous System Modeling*, Springer-Verlag, New York, 1991.
9. Cellier, F. E., Bond graphs—The right choice for educating students in modeling continuous-time systems, *Proceedings 1992 Western Simulation MultiConference on Simulation in Engineering Education*, Newport Beach, CA, January 22–24, 1992, pp. 123–127.
10. Cellier, F. E., Hierarchical nonlinear bond graphs: A unified methodology for modeling complex physical systems, *Simulation*, *58(4)*, 230–248 (1992).
11. Cellier, F. E., and Elmqvist, H., The need for automated formula manipulation in object-oriented continuous-system modeling, *Proceedings CACSD '92—IEEE*

Computer-Aided Control System Design Conference, Napa, CA, March 17–19, 1992, pp. 1–8.

12. Cellier, F. E., Nebot, A., Mugica, F., and de Albornoz, A., Combined qualitative/ quantitative simulation models of continuous-time processes using fuzzy inductive reasoning techniques, *Proceedings SICICA '92, IFAC Symposium on Intelligent Components and Instruments for Control Applications*, Málaga, Spain, May 20–22, 1992, pp. 589–593.
13. Cellier, F. E., Zeigler, B. P., and Cutler, A. H., Object-oriented modeling: Tools and techniques for capturing properties of physical systems in computer code, *Proceedings CADCS '91—IFAC Computer-Aided Design in Control Systems Conference*, Swansea, Wales, July 15–17, 1991, pp. 1–10.
14. Date, C. J., *An Introduction to Data Base Systems*, 5th ed., Addison-Wesley, Reading, MA, 1991.
15. 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.
16. Elmqvist, H., *LICS—Language for Implementation of Control Systems*, Report CODEN: LUTFD2/(TFRT-3179), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.
17. Elmqvist, H., Åström, K. J., Schönthal, T., and Wittenmark, B., *Simnon—User's Guide for MS-DOS Computers*, SSPA Systems, Gothenburg, Sweden, 1990.
18. Elmqvist, H., and Mattson, S. E., Simulator for dynamical systems using graphics and equations for modelling, *IEEE Contr. Syst. Mag.*, *9*(1), 53–58, 1989.
19. *Modular Modeling System (MMS): A Code for the Dynamic Simulation of Fossil and Nuclear Power Plants*, Report: CS/NP-3016-CCM, Electric Power Research Institute, Palo Alto, CA, 1983.
20. *EASE+—User's Manual*, Expert-EASE Systems, Inc., Belmont, CA, 1988.
21. *MATRIX$_x$ User's Guide*, *MATRIX$_x$ Reference Guide*, *MATRIX$_x$ Training Guide*, *Command Summary and On-Line Help*, Integrated Systems, Inc., Santa Clara, CA, 1984.
22. *SYSTEM-BUILD User's Guide*, Integrated Systems, Inc., Santa Clara, CA, 1985.
23. Kettenis, D. L., COSMOS: A simulation language for continuous, discrete and combined models, *Simulation*, *58*(1), 32–41, (1992).
24. Korn, G. A., *Interactive Dynamic-System Simulation*, McGraw-Hill, New York, 1989.
25. Kröner, A., Holl, P., Marquardt, W., and Gilles, E. D., DIVA—An open architecture for dynamic simulation, *Comput. Chem. Eng.*, *14*(11), 1289–1295 (1990).
26. Kuipers, B., and Farquhar, A., *QSIM: A Tool for Qualitative Simulation*, Internal Report: Artificial Intelligence Laboratory, The University of Texas, Austin, TX, 1987.
27. Law, A. M., and Vincent, S. G., *UniFit II User's Manual*, Simulation Modeling and Analysis Company, Tucson, AZ, 1990.
28. Mathworks, Inc., *The Student Edition of MATLAB for MS-DOS or Macintosh Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
29. Mattson, S. E., and Andersson, M., The ideas behind OMOLA, *Proceedings CACSD '92—IEEE Computer-Aided Control System Design Conference*, Napa, CA, March 17–19, 1992, pp. 23–29.

30. *ACCULIB User's Manual*, Mentor Corp., Palo Alto, CA, 1987.
31. Mitchell, E. E. L., and Gauthier, J. S., *ACSL: Advanced Continuous Simulation Language—User Guide and Reference Manual*, Mitchell & Gauthier Assoc., Concord, MA, 1986.
32. Pantelides, C. C., Speed Up—Recent advances in process simulation, *Comput. Chem. Eng.*, *12*(7), 745–755 (1988).
33. Pegden, C. D., *Introduction to SIMAN*, Systems Modelling Corp., State College, PA, 1982.
34. Perry, R. H., Green, D. W., and Maloney, J. O. (eds.), *Perry's Chemical Engineers' Handbook*, 6th ed., McGraw-Hill, New York, 1984.
35. Piela, P. C., Epperly, T. G., Westerberg, K. M., and Westerberg, A. W., ASCEND: An object-oriented computer environment for modeling and analysis—Part 1: The modeling language, *Comput. Chem. Eng.*, *15*(1), 53–72 (1991).
36. Pritsker, A. A. B., *Introduction to Simulation and SLAM-II*, 3rd ed., Halsted Press, New York, 1985.
37. Stephanopoulos, G., Johnston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis, M., and Siletti, C., DESIGN-KIT: An object-oriented environment for process engineering, *Comput. Chem. Eng.*, *11*(6), 655–674 (1987).
38. Standridge, C. R., and Pritsker, A. A. B., Using data base capabilities in simulation, in *Progress in Modelling and Simulation* (F. E. Cellier, ed.), Academic Press, London, 1982, pp. 347–365.
39. Standridge, C. R., and Pritsker, A. A. B., *TESS—The Extended Simulation Support System*, Halsted Press, New York, 1987.
40. *CTRL-C, A Language for the Computer-Aided Design of Multivariable Control Systems, User's Guide*, Systems Control Technology, Inc., Palo Alto, CA, 1985.
41. *SYSMOD User Manual*, Release 1.0, D05448/14/UM, Systems Designers, plc, Ferneberga House, Farnborough, Hampshire, UK, 1986.
42. *WORKVIEW Reference Guide*, Release 3.0, Viewlogic Systems, Inc., Marlboro, MA, 1988.
43. Zeigler, B. P., *Multifaceted Modelling and Discrete Event Simulation*, Academic Press, London, 1984.
44. Brooks, B. A., and Cellier, F. E., Modeling of a distillation column using bond graphs, *Proceedings 1993 Western Simulation MultiConference on Bond Graph Modeling*, San Diego, CA, January 18–20, 1993, pp. 315–320.
45. Weiner, M., and Cellier, F. E., Modeling and simulation of a solar energy system by use of bond graphs, *Proceedings 1993 Western Simulation MultiConference on Bond Graph Modeling*, San Diego, CA, January 18–20, 1993, pp. 301–306.