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

A FIVE LEVEL HIERARCHY FOR THE MANAGEMENT OF SIMULATION MODELS †

François E. Cellier
Qingsu Wang
Bernard P. Zeigler

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721

ABSTRACT

This advanced tutorial describes the goal–driven automated generation of models from a set of design specifications. A five level hierarchy is introduced which supports the automated generation of both models and simulation experiments from an abstract description of an overall design, and from an abstract description of the goals of the simulation study. The aim is to be able to generate models and experiments in a top–down fashion from a description of static components and the couplings between these components by automating the stepwise refinement process. Detailed model descriptions are extracted from template files residing in model libraries. The paper emphasizes on the problems encountered in the automatic generation of continuous–system models since the synthesis of these models is more involved than the synthesis of discrete–event models.

The paper starts with an assessment of the need for the proposed automated model synthesis methodology. In the sequel, the advocated five level hierarchy will be presented in a bottom–up fashion starting with classical approaches to continuous–system simulation (the first and bottom layer of our hierarchy), and advancing to higher and higher levels of abstraction. The paper ends with the presentation of a complete example of the proposed methodology, now presented in a top–down fashion.

1. INTRODUCTION: THE NEED FOR AUTOMATED MODEL SYNTHESIS

Applications for automated model synthesis technology can be found in unmanned deep Space missions where robots must be able to make intelligent model–based decisions. This capability is important since it reduces the necessity for frequent communications with Earth which inevitably slow down the decision making process due to extensive communication time delays.

Applications can also be found in manufacturing. The design of either new parts or new tools from similar existing parts or tools is a complex process which is accompanied by much responsibility and few rewards. Automated design and simulation aids can make this process less painful, and can reduce the time needed to educate new design engineers.

Finally, applications can be found in real–time decision support. Military strategists could use this technology to describe proposed scenarios in abstract high–level terms. Detailed models for simulating the proposed scenarios could be automatically generated, and the strategist could watch the most likely effects of a proposed strategy to unfold before his/her eyes.

1.1 Space

When humanity will colonize other planets of our solar system such as planet Mars or the moons of Jupiter and Saturn, this colonization will have to occur in three phases. In a first phase, unmanned Space missions will deploy high–autonomy systems which must prepare the target planet for human arrival. These missions will be carried out by smart robots with a high degree of decision making capability. The second phase will consist of manned Space missions conducted by a few highly specialized astronauts who must be supported in their endeavors by an abundance of smart automatic devices. During this second phase, the need for high–autonomy systems will not decrease. On the contrary, it will grow. Only in the third phase will it be possible to send larger numbers of less specialized humans to the target planet, and only then will life on this planet start to resemble life here on Earth.

As a first (and very rudimentary) example of this multi–phased approach to colonization, the University of Arizona is currently developing a prototype of an automated system that will be able to produce oxidizer for rocket fuel on planet Mars from CO_2 extracted from the Martian atmosphere [Ramohalli et al 1989]. This high–autonomy system will be deployed by an unmanned mission to Mars which will not return to Earth. The deployed oxygen production plant will then produce oxygen in a basically unsupervised operational mode during a period of two years. A manned follow–up mission to Mars will then use the oxygen produced by this plant for its return flight to Earth. With this approach, manned missions to Mars can be made more economical since it will no longer be necessary to lift the (heavy) oxygen for the return flight out of the gravity well of planet Earth.

Intelligent decision making requires insight into the consequences of the decision made. Frequently, the decision making process consists of choosing among a series of alternative scenarios. The decision maker must be able to assess the pro's and con's of each alternative by predicting their future effects on the overall plan. For this purpose, he often wishes to simulate alternative designs prior to their implementation.

A robot roaming around on planet Mars is confronted with exactly this problem. To choose between a set of alternative scenarios, it should base its decision on a proper assessment of the effects of each one. Since the situation of the robot at the time of decision making depends heavily on previous decisions it has made, it is unfeasible to carry along individual simulation programs for all possible alternatives. Such an approach would quickly lead to myriads of simulation programs which are largely the same, but each one of which is slightly different from every other. Instead, the decision making process should operate on a "world model" [Albus et al. 1987] from which a specific simulation model for any given specific purpose can be automatically synthesized at any time [Zeigler et al. 1989a].

1.2 Manufacturing

Aircraft manufacturers maintain relational databases containing large numbers of construction plans for aircraft parts. Whenever a new part is needed, a process design engineer searches such a database for one or several similar designs, and modifies them until the new design is completed. The new design is then added to the database. This is an extremely stressful job which carries a lot of responsibility (if the design doesn't work, who is to blame?) and few rewards. Consequently, the average process design engineer "lasts" in this job for only 18 to 24 months, while his or her education consumes the first nine to 12 months of this period. Manufacturers obviously have a problem here.

An automated model generation system (AMGS) can help overcome such problems. With the new technology, a process

† This research is supported in part by NASA-Ames Co-operative Agreement No. NCC 2-525, "A Simulation Environment for Laboratory Management by Robot Organizations", and in part by the NASA/University of Arizona Space Engineering Research Center for the Utilization of Local Planetary Resources (SERC/CULPR).

design engineer would start by formulating (and formalizing) a set of goals and a set of constraints describing the new design. The AMGS would then automatically generate one or several alternative designs together with simulation models for these designs, alternatives which satisfy all the goals and violate none of the constraints. If the number of resulting designs is small, the process design engineer could then simulate all of the proposed designs and pick the one which seems to be most attractive. If the AMGS generates too many designs, the engineer could formulate additional goals and/or constraints to limit the number of acceptable designs, and if the AMGS does not find any solution, the engineer knows that he has overconstrained the problem. This new technology will make the job of the process design engineer more interesting and more joyful, it will reduce the time needed for his/her education, and it will enhance the quality of the end product since the design engineer can explore many more alternative designs than s/he would ever have time for in the current manual technology.

1.3 Military Strategic Planning

Strategic planning requires real-time decision making in a partly unknown environment. In current technology, this problem is tackled manually by assembling the general's staff in a room. The general proposes possible strategies (scenarios), and his staff will "simulate" the effects of the proposed strategies in a brainstorming exercise by throwing potential suggested outcomes back at the general. Finally, the general makes his decision taking into consideration all the proposed (and partly contradictory) outcomes that he received from his staff.

We propose that our new technology can provide a more sound basis for decision making. While it is impossible to generate ahead of time simulation models for every potential situation, it is possible to develop ahead of time a "world model" from which our AMGS can then synthesize on the fly simulation models for any proposed scenario. In this way, the effects of a proposed strategy could be simulated in real time and could support the general in making the right decision.

2. LEVEL ONE: CLASSICAL MODELS FOR CONTINUOUS-SYSTEM SIMULATION

Models used for similar purposes often contain similar components. For example, a model of a DC-motor can be used to describe the mechanism that drives a windshield wiper in a car, the sump pump behind a house, or the food processor in a kitchen. The model is always the same, just the parameters assume different values. It therefore makes sense to include a DC-motor template in the "world model", a generic DC-motor from which a specific DC-motor for any given purpose can be generated when needed. In this way, the DC-motor model needs to be debugged only once, and if the template model is ever modified, all its future instantiations automatically reference the updated template model. It is also possible to equip the model editor with a mechanism which allows it to trace all references to the edited model within the model library, and which automatically deletes the compiled versions of all those models from the library, thereby forcing the user to migrate model modifications through all simulation programs that make use of those models.

Object-oriented programming systems provide powerful environments to support such a modular approach. In a previous tutorial [Zeigler et al 1989b], we described how modular modeling can be developed for *discrete-event simulation*. In this tutorial, we deal with *continuous-system models*. Here, the situation is a little more complex.

Traditionally, "modular" continuous-system models were expressed in most Continuous-System Simulation Languages (CSSL's) using a macro processor [Augustin et al 1967]. Macros are text templates. During compilation of the simulation program, the macro call is replaced by the macro definition body. Let us explain the macro replacement process by means of a simple DC-motor macro programmed in the macro language of ACSL [Mitchell and Gauthier 1986]. The following macro can represent an either armature-controlled or field-controlled DC-motor.

```
MACRO DCMOT(theta, omega, ua, uf, tauL, JL, ...
    Ra, La, Rf, Lf, k, Jm, Bm, flag, if0, ia0, T0, th0)
    MACRO redefine ia, iadot, if, ifdot, ui, psi
    MACRO redefine taum, Twist, Tdot
    MACRO standval if0 = 0.0, ia0 = 0.0, T0 = 0.0
    MACRO standval th0 = 0.0
    MACRO if (flag = IND)labind
        if      = uf/Rf
        ia      = (ua - ui)/Ra
    MACRO goto goon
    MACRO labind..continue
        ifdot   = (uf - Rf * if)/Lf
        if      = INTEG(ifdot, if0)
        iadot   = (ua - ui - Ra * ia)/La
        ia      = INTEG(iadot, ia0)
    MACRO goon..continue
        psi     = k * if
        taum    = psi * ia
        ui      = psi * omega
        Tdot    = taum - tauL - Bm * omega
        Twist   = INTEG(Tdot, T0)
        omega   = Twist/(Jm + JL)
        theta   = INTEG(omega, th0)
MACRO END
```

This macro exhibits a number of interesting features. Let us look at the armature equation:

$$u_a = u_i + R_a \, i_a + L_a \frac{di_a}{dt} \quad (1)$$

Usually, eq(1) will be implemented in the form of a differential equation in state-space form, i.e.:

$$\frac{di_a}{dt} = \frac{u_a - u_i - R_a \, i_a}{L_a} \quad (2)$$

This is done since we always wish to transform continuous models into a state-space form such that all differential equations can be numerically integrated rather than differentiated. However, the electrical time constants of the DC-motor are often so much smaller than the mechanical time constant that the effect of the armature inductance L_a on the overall system behavior can be neglected. In this case, we cannot operate on eq(2) since, if we set $L_a = 0.0$, this results in a division by zero. Instead, we must return to eq(1), delete the term in L_a from the equation, and rewrite it as:

$$i_a = \frac{u_a - u_i}{R_a} \quad (3)$$

Notice that this example already confronts us with two different versions of DC-motor models. Instead of creating two separate macros for these two cases, it was decided to code them as two variants within the same macro. The constant parameter *flag* can assume either the value *IND* or *NOIND* in the macro call. Depending on the setting of this compile-time parameter, the macro replacer will generate code either in the form of eq(2) or in the form of eq(3). Notice however that the "macro if" statement is not truly an "advanced feature" of the ACSL macro handler. It is just a crude way to implement only the simplest of the capabilities that a true model management system can offer.

The above macro can be invoked by a statement such as:

```
DCMOT(theta, omega = ua, uf, tauL, JL, ...
      Ra, La, Rf, Lf, kmot, Jm, Bm, "IND")
```

The last few parameters were omitted on the call. Parameters, which are assigned default values ("macro standval") in the macro definition, can be omitted in the macro call. The *flag* parameter was specified as *"IND"*, and therefore, both the armature inductance and the field inductance are included in the model equations generated in the process of macro text replacement.

Let us now look at the equations that the macro handler generates during the macro expansion. The two inductances

were included.

$$
\begin{aligned}
Z09996 &= (uf - Rf * Z09997)/Lf \\
Z09997 &= INTEG(Z09996, 0.0) \\
Z09998 &= (ua - Z09995 - Ra * Z09999)/La \\
Z09999 &= INTEG(Z09998, 0.0) \\
Z09994 &= kmot * Z09997 \\
Z09993 &= Z09994 * Z09999 \\
Z09995 &= Z09994 * omega \\
Z09991 &= Z09993 - tauL - Bm * omega \\
Z09992 &= INTEG(Z09991, 0.0) \\
omega &= Z09992/(Jm + JL) \\
theta &= INTEG(omega, 0.0)
\end{aligned}
$$

All local variables of the macro (which were declared inside the macro definition using the "macro redefine" statement) were renamed into generic names. This is necessary for the reason given below.

The (differential and algebraic) equations that make up a continuous-system model are basically *parallel code.* Contrary to the discrete-event case where the equal sign in an equation denotes an *assignment*, in continuous-system simulation, the equal sign in an equation denotes an *equality*. It means that the quantities on the left hand side of the equal sign are *at all times* equal in size to the quantities on the right hand side of the equal sign. Thus, continuous-system model equations are of a declarative nature.

Most CSSL languages offer an *equation sorter* which is activated after the macro replacement has been completed, and which will sort the model equations into an executable sequence. Of course, it cannot do so if the same variable appears more than once on the left hand side of the equal sign. Thus, in order to allow a macro to be invoked more than once in a program, it is necessary to replace all locally defined variables by new generic names in each new macro replacement.

In the above example, notice that e.g. the variable *Z09995* is used before it has been defined. The equation sorter will have to arrange the sequence of statements such that all variables except for those defined through integral equations are evaluated before they are used. Integral equations are different since all continuous-system simulation languages use explicit numerical integration schemes. For example, using forward Euler, we can write:

$$x(t + \Delta t) = x(t) + \Delta t \cdot \dot{x}(t) \tag{4}$$

Since this equation computes a new value for x for a time instant Δt time units into the future, we do not need to evaluate this equation simultaneously with the evaluation of the equation for $\dot{x}(t)$.

And yet, macros are *not* truly modular. The term "modularity" is often used in a narrow sense as a means to structuring program code into sections. However, we prefer to define the term "modularity" in a much wider sense as the capability of a program segment to represent a real object correctly irrespective of the environment in which it is used. Employing this definition, the term *modular modeling* becomes synonymous with *object-oriented modeling.*

Notice that DC-motors can be used in two different ways. They can be driven electrically (either through the armature or through the field) in which case they will transform a portion of the electrical energy entered into the system into mechanical energy, as a consequence of which the mechanical axle starts rotating. However, we can also rotate the mechanical axle by force, thereby entering mechanical energy into the system, and as a result, a portion of that energy will be converted into electrical energy, generating a voltage across the two terminals of the armature coil. In this case, the DC-motor is used as a DC-generator.

Clearly, the physical object is the same irrespective of whether it is used as a motor or as a generator. In an object-oriented modeling environment, we should therefore be able to represent both types of operation through one and the same software object.

Let us write down an ACSL macro describing the DC-motor in its generator mode.

```
MACRO DCGEN(theta, omega, ua, tauin, uf, RL, LL, ...
    Ra, La, Rf, Lf, k, Jm, Bm, if0, ia0, T0, th0)
    MACRO redefine ia, iadot, if, ifdot, ui, psi
    MACRO redefine tauL, Twist, Tdot
    MACRO standval if0 = 0.0, ia0 = 0.0, T0 = 0.0
    MACRO standval th0 = 0.0
    ifdot   = (uf - Rf * if)/Lf
    if      = INTEG(ifdot, if0)
    iadot   = (ui - (Ra + RL) * ia)/(La + LL)
    ia      = INTEG(iadot, ia0)
    ua      = RL * ia + LL * iadot
    psi     = k * if
    tauL    = psi * ia
    ui      = psi * omega
    Tdot    = tauin - tauL - Bm * omega
    Twist   = INTEG(Tdot, T0)
    omega   = Twist/Jm
    theta   = INTEG(omega, th0)
MACRO END
```

When the machinery is operated in its *motor* mode, the main input to the system is the armature voltage u_a. It causes a current i_a to flow through the armature coil. The armature current then causes a mechanical torque τ_m to be built up which is the cause of the mechanical rotation. However, once the motor rotates, i.e., once its angular velocity ω builds up, this angular velocity is responsible for an induced voltage u_i to be generated back on the electrical side which is subtracted from the armature voltage u_a and thereby reduces the armature current i_a until an equilibrium is reached. τ_L and J_L represent the mechanical load of the motor, i.e., they are variables belonging to an external data object. They are not part of the DC-motor data object. These two variables are additional inputs to the DC-motor model.

When the machinery is operated in its *generator* mode, the main input to the system is the driving torque τ_{in}. It causes the motor to rotate. Once an angular velocity ω is built up, it induces a voltage u_i on the electrical side which causes a current i_a to flow through the armature coil. The armature current i_a causes a mechanical torque τ_L to be built up back on the mechanical side which opposes the driving torque τ_{in}. The armature current i_a is also responsible for building up an armature voltage u_a across the two armature terminals. The armature voltage u_a is subtracted from the induced voltage u_i thereby weakening the armature current i_a. This process continues until an equilibrium is reached. The load is now electrical, symbolized in our model by a resistive load R_L and an inductive load L_L which, in themselves, are not part of the DC-generator, and are therefore additional inputs to the DC-generator model.

For simplicity, we left the variant without inductances out. Notice that the equations are basically the same in the motor mode and in the generator mode. Minor differences exist as a consequence of the different loads. However, what is truly different in the two models is the assignment of inputs and outputs. In the motor mode, u_a is an input to the macro, whereas in the generator mode, u_a is an output.

We realize that CSSL type macros aren't truly modular with respect to the objects they represent. The same physical device calls for quite different macros depending on the environments in which they are supposed to operate. The simplest "macro" representing an electrical resistor, for instance, must be stored in the macro library in two different versions, one modeling the equation:

$$u_R = R \cdot i_R \tag{5a}$$

and the other modeling the equation:

$$i_R = \frac{u_R}{R} \tag{5b}$$

If the resistor is placed over a current source, the current i_R through the resistor is known, and we need to use the macro which reflects the model according to eq(5a), whereas if we place the resistor over a voltage source, the voltage u_R across the resistor is known, and we need to use the macro which reflects the model according to eq(5b). Obviously, an *equation sorter*

is insufficient. In an object-oriented continuous-system modeling environment, we require an *equation solver* which accepts general equalities of the type:

$$< expression >=< expression > \quad (6a)$$

or:

$$< expression >= 0.0 \quad (6b)$$

and which can solve these equalities for arbitrary variables.

3. LEVEL TWO: OBJECT-ORIENTED MODELING AND THE DYMOLA MODELING LANGUAGE

Notice that macro handlers, which are commonly considered an intrinsic part of a CSSL language, have in fact nothing to do with the simulation language itself. The macro text replacement must be performed at source level, and must be completed before any other activity of the compiler can begin. The macro handler is often implemented as the first path of the compilation, and is completely separate from everything that follows. It would make perfect sense to develop a macro handler independent of the simulation language for which it is being used. The same macro handler could easily be used as a front end to several different simulation language compilers.

This approach was taken with DYMOLA [Elmqvist 1978]. DYMOLA is a *modeling language* and not a *simulation language*. DYMOLA is considerably more general than the conventional macro handler, but it serves the same purpose. DYMOLA is a program generator which can generate code for a variety of different simulation languages. If DYMOLA is used as a preprocessor, the simulation language no longer needs a macro handler of its own; in fact, it no longer requires an equation sorter since DYMOLA will sort the resulting set of equations into an executable sequence after solving each of them for the appropriate variable. The syntax of statements in DYMOLA is that of eq(6a).

DYMOLA is able to solve arbitrarily non-linear equations for any variable as long as that variable appears linearly in the otherwise non-linear equation. DYMOLA cannot currently handle algebraic loops, not even linear algebraic loops (which is a pity). DYMOLA cannot currently handle most types of structural singularities as they occur when subsystems are coupled together in such a way that the overall system exhibits fewer degrees of freedom than the sum of the subsystems. Finally, DYMOLA cannot currently eliminate redundant equations as they appear frequently when subsystems are coupled together which have fuzzy borderlines between them, i.e., when one and the same equation can be viewed as belonging to one or the other of the subsystems depending on their utilization. Other than that, DYMOLA has all the properties needed for truly hierarchical modular modeling, i.e., for object-oriented modeling. More details about the current capabilities and limitations of DYMOLA can be found in [Cellier 1990b].

Let us discuss how the DC-motor example can be coded in DYMOLA.

```
model type DCMOT
    terminal theta, omega, ua, uf, tauL, JL
    local ia, if, ui, psi, taum, Twist
    parameter Ra, Rf, kmot, Jm
    parameter La = 0.0, Lf = 0.0, Bm = 0.0
    default ua = 25.0, uf = 25.0
        Lf*der(if)  = uf - Rf * if
        La*der(ia)  = ua - ui - Ra * ia
        psi         = kmot * if
        taum        = psi * ia
        ui          = psi * omega
        der(Twist)  = taum - tauL - Bm * omega
        Twist       = (Jm + JL) * omega
        der(theta)  = omega
end
```

This code is quite self-explanatory. However, let us discuss some of the special properties of DYMOLA model descriptions.

(1) DYMOLA variables belong to either the type *terminal* or the type *local*. They are of type terminal if they are supposed to be *connected* to something outside the model. They are local if they are totally connected inside the model.

(2) Terminals can be either *inputs* or *outputs*. What they are, often depends on the environment to which they are connected. The user can explicitly specify what he wants them to be by *declaring* them as "input" or "output" rather than simply as "terminal".

(3) Terminals can have *default values*. In this case, they don't need to be externally connected.

(4) DYMOLA constants can be declared to be of type *parameter*. For parameters, values can be assigned from outside the model. Parameters can have default values in which case it is not necessary to assign a value to them from outside the model.

(5) Derivatives are either expressed using the **der(.)** operator or a prime ($'$). It is also allowed to use a **der2(.)** operator or a double prime ($''$) to denote a second derivative, and even higher derivatives are admissible. Contrary to most CSSL's, DYMOLA permits the use of these operators anywhere in the equation, both to the left and to the right of the equal sign.

(6) Consequently, it is not possible to set initial conditions for the integrators inside a model which is clearly a disadvantage of DYMOLA.

(7) DYMOLA equations use the syntax of eq(6a). During the process of *model expansion*, equations are *solved* for the appropriate variable.

(8) Terms which are multiplied by a zero parameter are automatically eliminated during model expansion. Consequently, if $La = 0.0$, the model equation $La * \mathbf{der}(ia) = ua - ui - Ra * ia$ is first replaced by the modified model equation $0.0 = ua - ui - Ra * ia$ which then results in one of three simulation equations, namely (i) $ua = ui + Ra * ia$, (ii) $ui = ua - Ra * ia$, or (iii) $ia = (ua - ui)/Ra$ depending on the environment in which the model is used. However, if $La \neq 0.0$, the model equation is always transformed into the simulation equation $\mathbf{der}(ia) = (ua - ui - Ra * ia)/La$. This is a very elegant way to solve the "variant macro" problem of ACSL.

(9) The above rule indicates that parameters with value 0.0 are treated in a completely different manner than all other parameters. This decision has a side effect. Parameters that are not set equal to zero are preserved in the generated simulation code, and can be interactively altered through the simulation program directly without a need to return to DYMOLA. Parameters with value 0.0 are optimized away by the DYMOLA compiler, and are not represented in the simulation code. However, the advantages of this decision are overwhelming, since this does away with an entire class of *structural singularities*.

The above model can then be invoked in the following way:

```
submodel (DCMOT) dcm1(Ra = 2.0, Rf = 5.5, kmot = 1.0, Jm = 15.0)
```

It can be *connected* to the outside world using a dot-notation:

```
dcm1.ua = kalph * err
dcm1.uf = 12.0
dcm1.JL = crl1.JL
dcm1.tauL = crl1.tauL
crl1.omega = dcm1.omega
```

where *crl1* is the name of another model of the same or a different type.

DYMOLA models are much more modular than ACSL macros since equations are automatically solved during model

expansion for the variable which is appropriate in the context of the model call environment. The utilization of *named parameters* instead of *positional parameters* upon invocation of a DYMOLA model helps with long parameter lists. Default values can and should be assigned to many parameters, and with the named parameter convention, the user can selectively specify values for those parameters only for which the default values are not appropriate. The connection mechanism as presented so far is very general, although a little clumsy. Each connection corresponds to connecting two points of a circuit with a wire.

It can be noticed that wires are frequently grouped into cables or buses. For example, consider an RS232 connector. The RS232 male connector has 25 pins, while the corresponding RS232 female connector has 25 holes. It seems natural that a modeling language should provide for an equivalent mechanism. DYMOLA does this by providing so-called *CUTs*.

Let us look at the above example once more. It can be noticed that the two models *dcm1* and *crl1* have three variables in common, namely *omega*, *tauL*, and *JL*. We can therefore go ahead and declare those three variables in a *cut* rather than as simple *terminals*. The modified model type *DCMOT* now looks as follows:

```
model type DCMOT
    terminal theta, ua, uf
    cut mech(omega, tauL, JL)
    local ia, if, ui, psi, taum, Twist
    parameter Ra, Rf, kmot, Jm
    parameter La = 0.0, Lf = 0.0, Bm = 0.0
    default ua = 25.0, uf = 25.0
        Lf*der(if) = uf - Rf * if
        La*der(ia) = ua - ui - Ra * ia
        psi        = kmot * if
        taum       = psi * ia
        ui         = psi * omega
        der(Twist) = taum - tauL - Bm * omega
        Twist      = (Jm + JL) * omega
        der(theta) = omega
end
```

In the main program, we can invoke a DC-motor *dcm1* of type *DCMOT*, and another model called *crl1* which is assumed to contain a declaration for the same cut type *mech*, and we can connect the cut *mech* of *dcm1* at the cut *mech* of *crl1*. This is accomplished as follows:

```
submodel (DCMOT) dcm1(Ra = ...)
submodel (CABREL) crl1(Bl = ...)
connect dcm1:mech at crl1:mech
```

The *connect statement* automatically generates the three model equations:

$$dcm1.omega = crl1.omega$$

$$dcm1.tauL = crl1.tauL$$

$$dcm1.JL = crl1.JL$$

Cuts can be hierarchically structured. We could e.g. modify the model type *DCMOT* once more as follows:

```
model type DCMOT
    terminal theta
    cut mech(omega, tauL, JL)
    cut elect(ua, uf)
    cut both[mech, elect]
    local ia, if, ui, psi, taum, Twist
    parameter Ra, Rf, kmot, Jm
    parameter La = 0.0, Lf = 0.0, Bm = 0.0
    default ua = 25.0, uf = 25.0
        Lf*der(if) = uf - Rf * if
        La*der(ia) = ua - ui - Ra * ia
        psi        = kmot * if
        taum       = psi * ia
        ui         = psi * omega
        der(Twist) = taum - tauL - Bm * omega
        Twist      = (Jm + JL) * omega
        der(theta) = omega
end
```

in which case we can either connect the cut *mech* and the cut *elect* separately, or we can connect *both* together. During expansion of the *connect* statement, DYMOLA checks that the connected cuts are structurally compatible with each other.

However, even this won't suffice in many cases. We may notice that, by connecting a wire between two points in an electrical circuit, we actually connect *two* variables, namely the *potential* at the two points, and also the *current* that flows through the new wire. However, the two connections work differently. While the potentials of all cuts that are connected at a point must be equal, the currents must add up to zero. Variables of type potential are called *across variables*, while variables of type current are called *through variables*. DYMOLA provides a connection mechanism also for this second type of connection. The generalized form of a DYMOLA cut looks as follows:

```
cut < cut_name > (< through_variables > / < across_variables >)
```

If three models $m1$, $m2$, and $m3$ have each a cut of type A declared as:

```
cut A(v/i)
```

we can use the connect statement:

```
connect m1 : A at m2 : A at m3 : A
```

which will generate the following model equations:

$$m1.v = m2.v$$

$$m2.v = m3.v$$

$$m1.i + m2.i + m3.i = 0.0$$

Notice that currents at cuts are normalized to point *into* the subsystem. If a current is directed the opposite way, it must take a minus sign on the cut definition.

This concept is more generally useful that just for electrical circuits. In a mechanical system, all positions, velocities, and accelerations are across variables, while all forces and torques are through variables. In a hydraulic system, water levels are across variables, while water flow is a through variable. In a thermic system, temperature and pressure are across variables, while heat flow is a through variable, etc. These similarities between different types of physical systems are particularly emphasized in the bond graph modeling methodology [Cellier 1990a].

One cut can be declared as the *main cut*. The main cut is the *default* cut in a connection, i.e. it suffices to specify the model name to connect the main cut of a submodel.

Sometimes it is useful to allow connections to take place *inside* a model instead of across model boundaries. For this purpose, DYMOLA provides a *node* declaration. Nodes are named, and cuts can be connected to nodes. Nodes are hierarchically structured the same way cuts are.

```
model M
    cut A(v1, v2), B(v3, v4), C(v5, v6)
    main cut D[A, B, C]
        ...
end
...
node N
connect M at (N, N, N)
```

This connect statement is equivalent to:

```
connect M : A at N, M : B at N, M : C at N
```

which is identical to saying:

```
connect M : A at M : B at M : C
```

which will result in the following set of equations:

$$M.v1 = M.v3$$
$$M.v3 = M.v5$$
$$M.v2 = M.v4$$
$$M.v4 = M.v6$$

Sometimes, it is also useful to connect a variable through from a source to a destination. For this purpose, DYMOLA allows you to declare a directed *path* from an input cut to an output cut.

Let us assume we have a model describing a pump which is declared as follows:

```
model pump
    cut inwater(w1), outwater(w2)
    path water < inwater - outwater >
        ...
end
```

Let us assume we have two more models describing a pipe and a tank with compatibly declared cuts and paths, then we can connect the water flow from the pump through the pipe to the tank with the statement:

```
connect (water) pump to pipe to tank
```

One path can always be declared as the *main path*. If the main path is to be connected, the path name can be omitted in the connect statement.

Besides the *at* and *to* operators, DYMOLA provides some additional connection mechanisms which are sometimes useful. The *reversed* operator allows us to connect a path in the opposite direction. The *par* operator allows for a parallel connection of two paths, and the *loop* operator allows us to connect paths in a loop.

3.1 Hierarchical Model Construction

Notice that, while there does not exist a strict rule which forbids mixing physical equations with connection statements in one model, it is good practice to avoid such a mix. At the bottom of the model hierarchy are models which are described solely through physical equations and which do not contain any connect statements. These models are called *atomic models.* Models which invoke other models and describe the connections between these submodels are called *coupling models.* Coupling models can, of course, refer also to other coupling models, not only to atomic models, i.e., coupling models can be hierarchically structured. [Cellier 1990a] contains an example (solar heated house) of a hierarchically structured modular model with five hierarchy levels.

Notice however that the concept of structuring models in a hierarchical fashion is different from the five level hierarchy of model management which is the topic of this paper. Hierarchically structured, object-oriented models (with an arbitrary number of hierarchy levels) occupy just the second level of our hierarchical model management methodology, whereas the expanded, i.e. flat, simulation models occupy the first hierarchy level of the methodology.

Notice further that *object-oriented modeling* does not imply *object-oriented simulation* as well. DYMOLA clearly supports the concept of models representing physical objects in a modular, i.e. environment independent, fashion. Thus, DYMOLA supports object-oriented modeling. Yet, the generated simulation code is a flat simulation program expressed in any of a number of off-the-shelf simulation languages which clearly do not support object-oriented simulation. It is a commonly made mistake to believe that object-oriented program execution is necessarily desirable. It is the object-oriented user interface which is desirable since it simplifies programming. In the context of continuous-system simulation, an object-oriented approach to simulation would force us to exchange information between the simulation data objects using a mechanism of message passing. This is far too inefficient since continuous objects exchange information on a continuous basis. In the context of discrete-event simulation, an object-oriented approach to simulation is not mandatory, but at least, it makes some sense since such data objects exchange information only at event times. Thus, hierarchy flattening of discrete-event models is feasible and not harmful, but it is not absolutely necessary. However, hierarchy flattening of continuous-system models is a must from the point of view of run-time execution efficiency.

In the last two sections, we have discussed concepts involved in object-oriented continuous-system modeling. Object oriented discrete-event modeling is similar but somewhat less involved. In a discrete-event model, all connection variables are clearly specified as either input or output variables. It never happens that the direction of a path needs to be turned around due to the environment in which the model is used. Consequently, object-oriented discrete-event modeling environments do not require an equation solver. This simplifies their implementation to some extent.

4. LEVEL THREE: THE SYSTEM ENTITY STRUCTURE

One important property of object-oriented modeling has not been discussed yet. Object-oriented models hide details of the internal model structure from the outside. Only those properties of a model that transpire to its surface are noticeable from the outside. For instance, once a model has been encapsulated, only its input and output ports are still visible. From the outside, it is no longer evident whether the encapsulated model belongs to the class of discrete-event models or whether it belongs to the class of continuous-system models. Consequently, from now on, we no longer need to distinguish between discrete-event models *vs* continuous-system models. Everything that will follow (i.e., the hierarchy levels three to five) is valid for both model classes indistinguishably, since these hierarchy levels have no access to the internal model structure.

In other words, the *model dynamics* have been encapsulated at hierarchy level two, and therefore, from hierarchy level three on upwards, we will only deal with *objects* and their relations to each other.

The *System Entity Structure (SES)* is a mechanism to describe hierarchically structured sets of objects and their interrelations. The SES is a labeled tree with attached variables types, i.e., a graphical object that describes the decompositions of systems into parts. It is a knowledge representation scheme which formalizes the modeling of systems in terms of decomposition, taxonomic, and coupling relationships. The scheme supports structured knowledge acquisition for, and flexible restructuring of, families of large scale system designs.

Figure 1 shows a sketchy decomposition of the solar heated house of [Cellier 1990a] into parts. In order to keep Figure 1 small and understandable, only a few of the actual decompositions are shown.

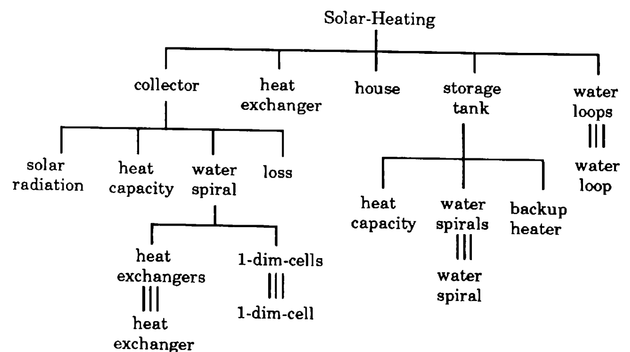

Figure 1. Decomposition of a solar heating into parts

Figure 1 is an *informal SES* of the solar heating system. It shows the decomposition of physical objects into parts. How-

ever, for practical utilization, the informal SES is insufficient. Somewhere, it must be shown *how* the physical objects are decomposed into parts. For this reason, we have developed *formal SES's* or simply SES's. In a formal SES, physical objects are called *entities*. The children of physical objects are always abstract objects called *aspects* and *specilizations*. The children of these abstract objects are always physical objects again. I.e., physical objects and abstract objects always toggle in a formal SES. The root object in a formal SES is always an entity. Also the leaf objects of the SES are always entities. They represent the *atomic models*. All entities which are not leaf objects represent *coupled models*.

It is not necessary to represent abstract objects differently from physical objects in the SES. It is easy to tell them apart simply by counting the number of levels in the SES. Objects at odd levels from the top or from the bottom of the SES represent always *entities* while objects at even levels of the SES are always either *aspects* or *specializations*. Physical decompositions are shown as aspects. We shall discuss the purpose of specializations at a later time. For the moment, it suffices to mention that it is exactly the distinction between aspects and specializations which forces us to formalize these abstract objects rather than implying the decompositions by attaching them to the parent entity directly.

The formal SES satisfies the following set of axioms:

(1) **Uniformity:** Any two nodes (objects) which have the same labels (names) have identical attached variable types and isomorphic subtrees.

(2) **Strict Hierarchy:** No label appears more than once down any path of the tree from the root entity to any of the leaf entities.

(3) **Alternating Mode:** Each node has a mode which is either *entity*, *aspect*, or *specialization*; if the mode of a node is *entity*, then the modes of its successors are either *aspect* or *specialization*; if the mode of a node is *aspect* or *specialization*, then the modes of its children are *entity*. The mode of the root is *entity*.

(4) **Valid Siblings:** No two siblings have the same label.

(5) **Attached Variables:** No two variable types attached to the same item have the same name.

(6) **Inheritance:** Every *entity* in a *specialization* inherits all the variables, *aspects*, and *specializations* from the parent of the *specialization*.

The SES is completely characterized by its axioms [Zeigler 1984, 1990]. However, the interpretation of the axioms cannot be specified, and thus is open to the user.

When constructing an SES, it may sometimes seem difficult to decide how to represent concepts of the real world in the SES. A meaningful decomposition of a system into parts is not always easy to accomplish, and yet, an inadequate decision can have serious consequences in terms of the number of wires that connect the various submodels to each other. One of the goals of a meaningful system decomposition is to limit the number of required connections between its subsystems. As of now, we don't have a tool to support the user in this process, i.e., we don't have a tool which can consider various alternative decompositions, and propose one which will result in few subsystem connections. Such a tool would require a detailed knowledge of the internal structure of all subsystems. It could infer this knowledge by tracing through the nested couplings as is done in hierarchy flattening.

4.1 Automated Model Generation and the DEVS–Scheme Modeling Environment

DEVS–Scheme [Zeigler 1990] is an application layer above the LISP–based PC–Scheme language [Texas Instruments 1986]. It supports the DEVS modeling formalism [Zeigler 1984]. DEVS stands for *Discrete EVent Specification*, i.e., DEVS was developed primarily for discrete–event models. Yet, as we explained above, the higher levels of the DEVS modeling hierarchy are the same for both discrete–event models and continuous–system models. DEVS–Scheme contains a set of PC–Scheme (i.e. LISP) procedures implementing the DEVS modeling formalism. They were written using the SCOOPS object–oriented LISP extensions which form an intrinsic part of the PC–Scheme environment. They were coded such that all PC–Scheme (LISP and SCOOPS) programming tools are transparently usable within DEVS–Scheme as well. In this light, DEVS–Scheme is not a specialization of PC–Scheme, but a superset of PC–Scheme.

System Entity Structures can be coded in DEVS–Scheme in the form of a text file which contains a one–to–one translation of the graphical SES representation. An example of such a text file will be presented later in this paper. Currently, the translation must still be done manually, but automatic translations from the graphical representation to the textual representation are being developed. Once the SES has been coded, DEVS–Scheme offers an automated *transform* procedure which iteratively generates more and more complex *coupled models* by invoking submodels and by setting up the coupling relations that were specified between them. In this way, *transform* translates the SES into either one single DEVS model or a set of DEVS models. The *transform* procedure can be requested to either flatten the hierarchy or keep the hierarchy. For continuous systems, *transform* can also automatically generate the necessary DYMOLA coupling models. Only the atomic (i.e. leaf) models must be user coded. The coupled models can be automatically generated from the SES.

It is possible to store coupled models in the model library as if they were atomic models. If *transform* finds a coupled model in the model library, it will stop searching for its children, and simply use the coupled model as if there were no more children and as if this were an atomic model.

Let us now return for a moment to the example of the aircraft manufacturer. If we maintain all our designs at the first hierarchy level (as this is done in the current technology), we need to store millions of different designs in the database. Each design is flat and therefore fairly complex. If a design is found to have a flaw, and if that flaw has been traced back to a particular subcomponent which may be intrinsically used also in hundreds of other designs, we should edit all these hundreds of flat design descriptions to remove the bug from the database. Of course, this is never done (too much work), and therefore, the database is always inconsistent. It can therefore easily happen that another process design engineer at a later time bases a new design on another earlier design which contains the same flaw that had been discovered once before, but which was never totally removed from the database.

If we maintain our designs at the second hierarchy level, we still need to store millions of different designs in the database. However, these designs will not be flat. They will reference design templates that are stored in a hierarchically structured template library. The main design files will thus be much shorter and more readable since they reference these templates rather than contain them in an already expanded form. If a flaw is traced back to a bug in a design template, the template itself is updated, rather than the design that makes use of it. When an atomic template is edited, all coupled templates in the template library which reference the atomic template either directly or indirectly could be automatically deleted from the template library. In this way, all future designs will automatically reference the updated template rather than the original one.

If the designs are maintained at the third hierarchy level, we still need to store millions of individual designs in the database, but this time, they are stored in the form of different System Entity Structures. This does not seem to buy us much in comparison with the previous (i.e., level two) alternative, but already the next section of the paper will explain why an SES is indeed considerably more powerful than either a DEVS model or a DYMOLA model.

5. LEVEL FOUR: THE GENERALIZED SYSTEM ENTITY STRUCTURE

We had explained earlier that the SES contains two types of abstract objects: *aspects* and *specializations*. Specializations enable us to store several variants of similar SES's in a single generalized SES. In fact, from now on, we shall call the "gener-

alized SES" simply "the SES". An SES without any specializations is a special case of "the SES". It is sometimes referred to as a *pure SES*.

Figure 2 shows an example of an SES which decomposes a car into a few components. Notice that the component *engine* is specialized into either *V6* or *Diesel*.

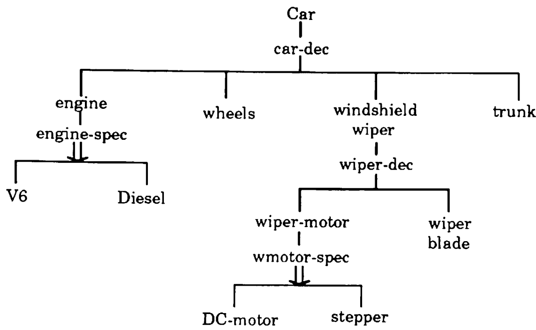

Figure 2. SES with specializations of a car

Pure SES's can be obtained from general SES's by a process called *pruning*. Pruning an SES means to cut away all specializations but one. The car SES of Figure 2 can be pruned to retain either the *V6* engine or the *Diesel* engine.

DEVS–Scheme [Zeigler 1990] contains a software tool called ESP–Scheme [Kim 1988] which can prune any SES. The output of the pruning process is a pure SES.

Returning once more to the aircraft manufacturing example, we notice that, at the fourth hierarchy level, we no longer need to store millions of different designs. Maybe, we can reduce all these designs to a few thousand basic designs with many specializations. This makes the SES of a system more powerful than either a corresponding DEVS model or a corresponding DYMOLA model. One single "model" expressed as an SES with specializations can represent many different variants of actual models of a process. This concept replaces the "macro if" statement of ACSL's macro handler, but it is considerably more powerful. One single SES can be used to generate hundreds, maybe thousands, of different models for different purposes.

From now on, we shall call an ensemble of one or several SES's with specializations which can be used to generate specific models for an entire application area a *world model*. Thus, a world model is an ensemble of System Entity Structures with specializations which can generate arbitrary models for an entire application area.

Returning to our example of the strategic planning tool, it now becomes evident how this problem is tackled. We generate a set of models that describe all feasible variants of a battle – not in the form of myriads of individual simulation programs, but in the form of a few SES's with specializations. Specific models for any actual situation can then be automatically generated on the fly. Any one of these specific models can be simulated, and will lead to a decent forecast of what the consequences of the simulated scenario would be (at the level of detail contained in the set of invoked atomic models).

6. LEVEL FIVE: GOAL–DRIVEN PRUNING

One problem that remains to be solved is the following. *How* do we prune an SES? Which branches do we cut, and which one do we retain? How do we choose between the *V6* engine and the *Diesel* engine?

The answer is fairly simple. We attach *rules* to the specialization objects in the same way that we attach coupling relations to the aspect objects. For the above example, the following rule could be coded:

```
if purchase_price = high and maintenance_cost = low
    then select Diesel
    else select V6
end if
```

Purchase_price and *maintenance_cost* are two attached variables of the parent node. They are enumerated variables which can assume the values "low", "medium", or "high".

The pruner always searches specializations for such rules. If it finds one or several rules that can be fired, it will do so. If this process results in a unique selection, the pruner accepts it and proceeds to the next node. If no rule has been specified, or if no rule can be fired, or if the fired rules do not lead to a unique selection, the pruner enters into an interactive mode and asks the user for a selection.

FRASES [Hu et al 1989] is an experimental software which implements some of these ideas. Unfortunately, this software has not yet been fully integrated with the DEVS–Scheme modeling environment. However, such integration is planned.

7. AN EXAMPLE: THE CABLE REEL PROBLEM

A new light weight fiber optics deep sea communication cable is to be laid through the British Channel between Calais in France and Dover in the United Kingdom. The cable comes on a huge reel which is placed on a ship. The ship moves slowly from one coast to the other by constantly leaving cable behind. A large motor unrolls the cable from the reel. A speedometer detects the speed of the cable as it comes off the reel. A simple proportional and integral (PI) controller is used to keep the cable speed v at its preset value V_{set}. V_{set} is the speed of the ship. A functional diagram of the overall system is shown in Figure 3.

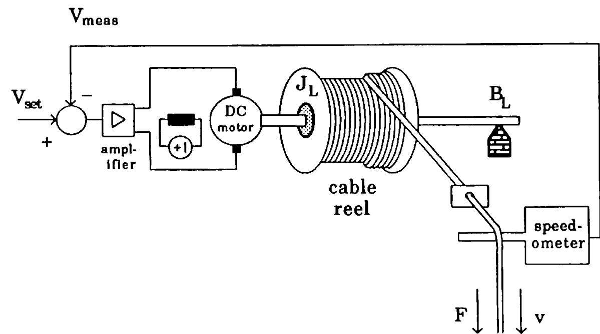

Figure 3. Functional diagram of the cable reel system

Control engineers are used to represent systems in the form of block diagrams. Figure 4 shows a block diagram of the overall control system.

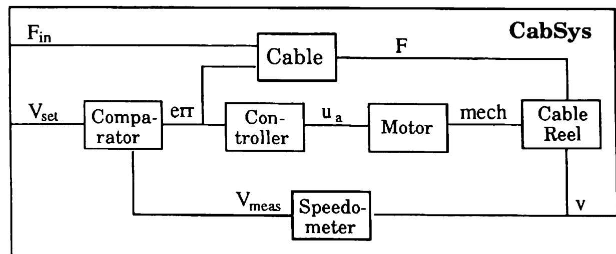

Figure 4. Block diagram of the cable reel system

Block diagrams are a good way to depict graphically a horizontal cut through a System Entity Structure. The SES is cut through an *aspect* node. The outer frame symbolizes the parent node, in this case the *root entity*, whereas the inner boxes symbolize the children of the aspect node. Wires connecting inner boxes (in our example: err, u_a, $mech$, v, V_{meas}, and F), are *internal connections* of the aspect node, whereas wires that extend to the outer frame (in our example: V_{set}, F_{in}, and v are

external connections of the aspect node, i.e., these are attached variables of the parent node. Notice that v appears both in the list of internal connections and in the list of external connections. However, this is only a question of convenience. We can easily transform the internal v connection into a second external v connection.

Notice that any one of the inner boxes could easily become an outer frame if it were further decomposed into smaller parts. In this sense, block diagrams are hierarchical. HIBLITZ is a graphical front end to DYMOLA which follows exactly this approach [Elmqvist 1982]. HIBLITZ allows us to zoom in on any of the internal boxes. Breakpoints are introduced to denote the magnification where the inner box suddenly becomes an outer frame, and where the internal structure of that box becomes visible, i.e., where we jump from one aspect node of the SES to another. In HIBLITZ, connections between boxes represent DYMOLA cuts, i.e., they are cables rather than wires. One single connection can contain many individual wires which can be either of the *across* type or of the *through* type, or both.

HIBLITZ could be used in our DEVS–Scheme modeling environment to graphically represent the connection relations that are attached to aspect nodes. However, HIBLITZ runs currently only on Silicon Graphics machines, and is therefore incompatible with our modeling environment. One of our students is currently working on a reimplementation of a subset of these capabilities to be integrated with the DEVS–Scheme modeling environment.

Figure 5 depicts the generalized SES of the cable reel problem.

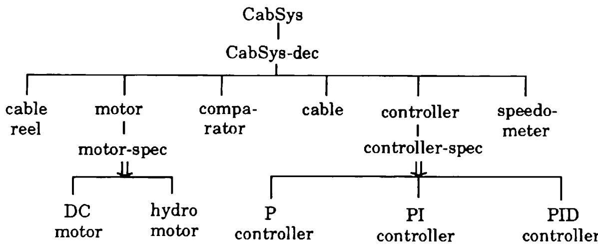

Figure 5. SES with specializations of the cable reel system

The root entity *CabSys* is decomposed into six parts. For this purpose, the aspect node *CabSys-dec* was introduced. Two of the component parts, namely the *motor* and the *controller* are specialized into variants. The *motor* can be either a *DCmotor* or a *hydromotor*. The selection is made in the specialization node *motor-spec*. Similarly, the *controller* can be either a *Pcontroller*, a *PIcontroller*, or a *PIDcontroller*. This selection is made in the specialization node *controller-spec*. Notice that the double lines in the SES qualify the abstract nodes *motor-spec* and *controller-spec* as *specializations* rather than as *aspects*.

The text file version of the SES is presented below.

```
;- Entity Structure for CabSys

(make-entstr 'CabSys)
(set-c-system-type e:CabSys 'system)

(ai e:CabSys asp 'Cabsys-dec)
(sci e:CabSys 'CabSys-dec)
(ai e:CabSys ent 'comparator)
(ai e:CabSys ent 'controller)
(ai e:CabSys ent 'motor)
(ai e:CabSys ent 'cablereel)
(ai e:CabSys ent 'speedometer)
(ai e:CabSys ent 'cable)

(add-variable e:CabSys 'input '(Fext Vdes)'())
(add-variable e:CabSys 'output '(radius velocity omega) '())

(acp e:CabSys 'comparator 'controller 'errport 'signal)
(acp e:CabSys 'controller 'motor 'command 'uport)
(acp e:CabSys 'motor 'cablereel 'mech 'mech)
(acp e:CabSys 'cablereel 'speedometer 'vport 'vport)
(acp e:CabSys 'speedometer 'comparator 'measport 'measport)
(acp e:CabSys 'comparator 'cable 'errport 'errport)
(acp e:CabSys 'cable 'cablereel 'fport 'fport)

(acpx e:CabSys 'CabSys 'comparator 'in1 'setport '(Vdes) '(Vset))
(acpx e:CabSys 'CabSys 'cable 'in2 'finport '(Fext) '(Fin))
(acpx e:CabSys 'CabSys 'cablereel 'rport 'out1 '(radius) '(R))
(acpx e:CabSys 'CabSys 'cablereel 'vport 'out2 '(v) '(velocity))
(acpx e:CabSys 'CabSys 'cablereel 'mech 'out3 '(omega) '(omega))

(sci e:CabSys 'controller)
(ai e:CabSys spec 'controller-spec)
(sci e:CabSys 'controller-spec)
(ai e:CabSys ent 'Pcontroller)
(ai e:CabSys ent 'PIcontroller)
(ai e:CabSys ent 'PIDcontroller)

(sci e:CabSys 'motor)
(ai e:CabSys spec 'motor-spec)
(sci e:CabSys 'motor-spec)
(ai e:CabSys ent 'DCmotor)
(ai e:CabSys ent 'hydromotor)

;—END—
```

The cable reel "world model" consists of:

(1) The SES with specializations shown in Figure 5.
(2) The block diagram shown in Figure 4 which explains how the decomposition of the system into parts is accomplished in the single aspect node *CabSys-dec*.
(3) Two rule bases, one for each specialization node, which formalize the knowledge when which of the specializations is to be used.
(4) A model library containing the nine atomic models (coded in DYMOLA) describing the nine leaf nodes of the SES.

From this (rudimentary) world model, any one of four different pure SES's can be obtained by means of *pruning*. Figure 6 shows one such pure SES.

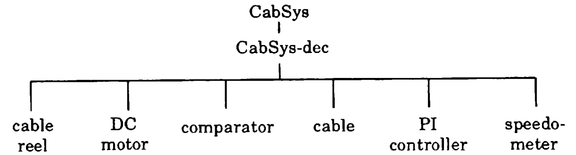

Figure 6. Pure SES of the cable reel system

The pure SES can now be *transformed*. This process will generate the previously missing DYMOLA coupling model shown below.

```
model CabSys

    submodel comparator
    submodel PIcontroller(kint = 0.2, kprop = 6.0)
    submodel DCmotor(Ra = 0.25, Rf = 1.0, kmot = 1.5, -- >
                     Jm = 5.0, Bm = 0.2)
    submodel cablereel(Rempty = 0.6, W = 1.5, D = 0.0127, - >
                     rho = 1350.0, BL = 6.5, J0 = 150.0)- >
                     (ic R = 1.2)
    submodel speedometer
    submodel cable

    input Vdes, Fext
    output radius, velocity, omega

    connect comparator:errport at PIcontroller:signal
    connect PIcontroller:command at DCmotor:uport
    connect DCmotor:mech at cablereel:mech
    connect cablereel:vport at speedometer:vport
    connect speedometer:measport at comparator:measport
    connect comparator:errport at cable:errport
    connect cable:fport at cablereel:fport

    cable.Fin = Fext
    comparator.Vset = Vdes
    radius = cablereel.R
    velocity = cablereel.v
    omega = cablereel.omega

end
```

The *transform* procedure will then automatically invoke the DYMOLA preprocessor which flattens the hierarchy and generates, in our example, a (flat) DESIRE simulation program [Korn 1989] representing the selected variant of the cable reel system. However, the simulation program could also have been generated for a number of alternative simulation languages.

```
--------------------------------------------
-- CONTINUOUS SYSTEM CabSys
--------------------------------------------
-- STATE R Twist theta x err
-- DER dR dTwist dtheta dx derr
-- OUTPUT radius velity Cs$oma
-- INPUT Vdes Fext
-- PARAMETERS and CONSTANTS :
kship = 10.0 | Rempty = 0.6 | W = 1.5 | D = 0.0127
rho = 1350.0 | BL = 6.5 | J0 = 150.0 | pi = 3.14159
Ra = 0.25 | Rf = 1.0 | kmot = 1.5 | Jm = 5.0
Bm = 0.2 | k = 3.0 | kint = 0.2 | kprop = 6.0
-- INITIAL VALUES OF STATES :
R = 1.2 | Twist = 0 | theta = 0 | x = 0 | err = 0
--------------------------------------------
TMAX = 3500 | DT = 0.05 | NN = 2000
Vdes = 15.0 | Fext = 100.0
scale = 1 | XCCC = 1
label TRY | drunr
if XCCC < 0 then XCCC = -XCCC | scale = 2 * scale | go to TRY
            else proceed
--------------------------------------------
DYNAMIC
--------------------------------------------
-- Submodel : speedometer
sr$Vms = k * x
-- Submodel : comparator
Pr$err = Vdes - sr$Vms
-- Submodel : cable
cale$F = lim(kship * Pr$err - Fext) + Fext
-- Submodel : cablereel
DCr$JL = 0.5 * pi * W * rho * (R ∧ 4 - Rempty ∧ 4) + J0
-- Submodel : DCmotor
Dr$oma = Twist/(Jm + DCr$JL)
-- Submodel : CabSys
Cs$oma = Dr$oma
-- Submodel : cablereel
d/dt R = -D * D/(2 * pi * W) * Cs$oma
Dr$taL = BL * Cs$oma - cale$F * R
-- Submodel : PIcontroller
u = kprop * Pr$err + kint * err
-- Submodel : DCmotor
uf = 25.0
DCr$if = uf/Rf
psi = kmot * DCr$if
ui = psi * Dr$oma
ia = (u - ui)/Ra
taum = psi * ia
d/dt Twist = taum - Dr$taL - Bm * Dr$oma
d/dt theta = Dr$oma
-- Submodel : cablereel
velity = R * Cs$oma
-- Submodel : speedometer
d/dt x = velity - k * x
-- Submodel : PIcontroller
d/dt err = Pr$err
-- Submodel : CabSys
radius = R
--------------------------------------------
OUT
term Rempty - R
radius = radius * 10
dispt radius, velity, Cs$oma
--------------------------------------------
/--
/PIC 'CabSys.PRC'
/--
```

The detailed equations contained in the nine atomic models are of no direct concern to this paper. More details can be found in [Cellier 1990b]. Notice that the equations stemming from the various submodels were both solved and sorted. Also, a number of variables were automatically renamed to conform with the DESIRE syntax.

The experiment description (i.e., the specification of the simulation run length, the termination conditions, the desired outputs, etc) was synthesized from a second SES in the same way as the model was. The SES, and to a large extent also DYMOLA, support a clean separation of the experiment description from the model description as mandated in [Zeigler 1976]. In the synthesized DESIRE program, the statements describing the model and those describing the experiment are wildly interspersed. However, this doesn't matter. It is at the user interface where a clean separation between model description and experiment description is important, not at execution time.

8. SUMMARY

In this paper, we have demonstrated the usefulness of automatic model synthesis tools. We have shown how a particular implementation of such a tool, the DEVS-Scheme modeling environment, can be used to tackle a number of important realistic application problems. The explicit example chosen to illustrate the automatic generation of continuous-system models, i.e. the cable reel problem, had obviously to be a very simple one in order to fit onto the limited space provided in this paper. However, considerably more complex examples have been developed and tested, and several (thick!) research reports describing these examples are available from the authors upon request.

REFERENCES

Albus, J.S., H.G. McCain, and R. Lumia (1987), *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, Robot Systems Division, Center for Manufacturing Engineering, National Technical Information Service, Gaithersburg, MD.

Augustin D.C., et al (1967), "The SCi Continuous System Simulation Language (CSSL)," *Simulation 9*, pp. 281-303.

Cellier, F.E. (1990a), "Hierarchical Non-Linear Bond Graphs: A Unified Methodology for Modeling Complex Physical Systems," keynote address, *Proceedings European Simulation MultiConference*, B. Schmidt, ed.), Nürnberg, FRG, June 10-13, 1990, pp. 1-13.

Cellier, F.E. (1990b), *Continuous-System Modeling and Simulation – Volume I: Modeling*, Springer Verlag, New York.

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. (1982), "A Graphical Approach to Documentation and Implementation of Control Systems," in: *Proceedings 3rd IFAC/IFIP Symposium on Software for Computer Control, SOCOCO'82*, Madrid, Spain, Pergamon Press.

Hu, J., J.W. Rozenblit, and Y.M. Huang (1989), "FRASES — A Knowledge Representation Scheme for Engineering Design," *Proceedings SCS MultiConference on Advances in A.I. and Simulation*, Tampa, FL, pp. 141-146.

Kim, T.G. (1988), *A Knowledge-Based Environment for Hierarchical Modelling and Simulation*, Ph.D Dissertation., Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721.

Korn, G.A. (1989), *Interactive Dynamic-System Simulation*, McGraw Hill, New York.

Mitchell, E.E.L., and J.S. Gauthier (1986), *ACSL: Advanced Continuous Simulation Language — User Guide / Reference Manual*, Mitchell & Gauthier Assoc., 73 Junction Square, Concord, MA 01742.

Ramohalli, K., E. Lawton, and R. Ash (1989), "Recent Concepts in Missions to Mars: Extraterrestrial Processes", *Journal of Propulsion and Power 5*, 2, pp. 181-187.

Texas Instruments (1986), *PC-Scheme Users Manual*, Science Applications Press.

Zeigler, B.P. (1976), *Theory of Modelling and Simulation*, John Wiley & Sons, New York.

Zeigler, B.P. (1984), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London, U.K.

Zeigler, B.P. (1990), *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, Boston.

Zeigler, B.P., F.E. Cellier, and J.W. Rozenblit (1989a), "Design of a Simulation Environment for Laboratory Management by Robot Organizations," *Journal of Intelligent and Robotic Systems 1*, pp. 299-309.

Zeigler, B.P., J. Hu, and J.W. Rozenblit (1989b), "Hierarchical, Modular Modelling in DEVS-Scheme," in: *Proceedings 1989 Winter Simulation Conference, WSC'89*, Washington, D.C., pp. 84-89.