Mathematical and Computer Modelling of Dynamical Systems 1387-3954/03/0901-065$16.00
2003, Vol. 9, No. 1, pp. 65–90

Object-oriented Design of Reusable Model Libraries of Hybrid Dynamic Systems – Part One: A Design Methodology

A. URQUIA[1] AND S. DORMIDO[2]

ABSTRACT

An object-oriented methodology for the design of reusable model libraries, dedicated to the composition of hybrid simulation models, is proposed. It takes into consideration all the phases of the library life cycle: design, programming, maintenance, modifications, extension, etc. The methodology distinguishes between the role of the library designers and the role of the library users. One of its main goals is guaranteeing that the users are able to use the libraries without having to know their internal details. In particular, users should not be confronted with numerical problems. The library designers should guarantee the numerical efficiency of the models based on the library predefined-models. In this respect, some of the numerical aspects of the model library design are discussed. The concept of *library design rules* is introduced as the cornerstone of the proposed methodology.

Keywords: Object-oriented modeling languages, first principles modeling, hybrid models, chemical engineering models, mathematical models.

1. INTRODUCTION

The use of general-purpose, object-oriented modeling languages has been growing gradually during the last decade. The first commercial versions of the modeling languages emerged in the early 90s, in order to support the physical modeling. The development of efficient and well-tested versions of these software tools has made of this modeling paradigm (i.e., the physical modeling) an attractive alternative to the block diagram modeling.

The *block diagram modeling* can be considered as an evolution from the analog-computing paradigm [1]. It requires of explicit state models (ordinary differential equations) and the blocks have a unidirectional data flow from input to outputs. These restrictions strongly condition the modeling task. For instance, dummy dynamics

[1]Address correspondence to: A. Urquia, Department of Computer Science and Automatic Control, U.N.E.D., Senda del Rey 9, 28040 Madrid, Spain. E-mail: aurquia@dia.uned.es
[2]Department of Computer Science and Automatic Control, U.N.E.D., Madrid, Spain.

need to be introduced in the model to avoid the establishment of systems of simultaneous equations. The model programmer has to manipulate the model to transform its equations to the form of ordinary differential equations (ODEs). As a consequence, a considerable modeling effort is required. However, the main advantage of this paradigm is demonstrated during the numerical integration of the model (i.e., the simulation run): numerical integration of ODEs is a mature, well-developed and yet very active research field. ACSL Graphics Modeller, EASY5 and SIMULINK are some examples of general-purpose simulation languages supporting the graphical block diagram modeling.

The *physical modeling paradigm* is based on the modular modeling methodology. Typically, the basic stages of the physical modeling are [1, 2]: (1) definition of the system structure and partition of the system into subsystems; (2) definition of the interaction among the subsystems; and (3) description of the internal behavior of each subsystem, independently of each other, in terms of mass, energy and momentum balances and of material equations. The modeling knowledge is represented as differential, algebraic and discrete equations that may change by being triggered by events (i.e., hybrid models). A model is considered as a constraint between system variables [1].

Some examples of object-oriented modeling languages are ABACUSS II [3], ASCEND [4], Dymola [5, 6], EcosimPro [7], gPROMS [8], MODE.LA [9, 10], Modelica [11] and Omola [12]. Dymola (Dynamic Modeling Laboratory) was the first modeling language in the market [13]. The Modelica language [11, 14] has been designed by the developers of the object-oriented modeling languages Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+, Smile and a number of modeling practitioners in different fields. It is intended to serve as a standard format for the external model representation, so that models arising in different engineering fields can be exchanged between tools and users.

The common characteristics of these modeling languages are the object-oriented, non-causal modeling methodology and the need for automatic symbolic formula manipulation. They support a declarative description of the model, based on equations (i.e., equation-oriented modeling) instead of assignment statements. The information of what variable to solve for in each equation is not included in the model (i.e., non-causal modeling). This permits better reuse of models since equations do not specify a certain data flow direction. Thus a model can adapt to more than one data flow context. The software tools supporting these modeling languages implement algorithms to automatically decide which equation to use for calculating each unknown variable.

The symbolic manipulations that these software tools carry out on the model can be classified into two types according to their purpose. First, manipulations intended to translate the object-oriented description of the model into the so-called flat model [14]. The flat model contains the complete set of model equations and functions, with all the object-oriented structure removed. Second, manipulations intended to transform the

flat model into an efficiently solvable form. This second type of manipulations includes:

1. The efficient formulation of the complete-model equations, eliminating the redundant variables and the trivial equations resulting from the submodels connections.
2. The sorting of the equations.
3. The symbolic manipulation of those equations in which the unknown variable appears linearly.
4. The reduction of the system index to zero or one [15–17].

The modeling environments needs, for simulating hybrid models (i.e., a set of synchronous differential, algebraic and discrete equations), the following:

1. A simulation algorithm appropriate for hybrid systems (for instance, the Omola simulation algorithm is described in [12]).
2. An adequate treatment of the discrete events [18]: the detection, the accurate determination of the trigger time [18–21] and the re-start problem solution.
3. Algorithms to carry out the symbolic manipulation of the linear systems of simultaneous equations and to tear the nonlinear ones [22].

In addition, the modeling environment needs to include at least one DAE-solver algorithm (e.g., DASSL [15]). The simulation efficiency is notably increased with the use of inline integration algorithms [23].

1.1. Some Fundamental Concepts of the Object-Oriented Modeling Methodology

The physical modeling practice using object-oriented modeling languages considerably reduces the modeling effort. Object-oriented modeling methodology facilitates the design, the programming, the reuse and the maintenance of the models. As a consequence, it reduces the modeling costs [2, 24]. This design methodology incorporates the concepts from the modular and hierarchical modeling.

The basic stages of the *modular modeling methodology* are [2]: (1) definition of the system structure and partition of the system into subsystems; (2) definition of the interaction among the subsystems; and (3) description of the internal behavior of each subsystem independently of each other. In order to support the modularization, the modeling language must allow: (1) the description of each submodel independently of the others; (2) the connection of the submodels; and (3) the *abstraction* (i.e., the capability of using a submodel without knowing its internal details). A way of improving the *abstraction* consists in distinguishing between its interface and its internal description in the submodel description. The *interface* describes the interaction between the model and its environment. The *internal description* contains the information about the model structure and behavior. Abstraction and modularization

are closely related to the *information encapsulation*: only the interface variables of the model are accessible by other models. Information encapsulation facilitates the modification, the test and the maintenance of the model [12, 25].

In the same way, the *hierarchical description* of the model facilitates its design, programming, maintenance and reuse. It consists in describing the model progressing from lower to higher level of detail: the model is split into submodels and these are split into sub-submodels and so on. The modular and hierarchical description of the model leads to the classification of the models into *primitive* and *composed*. *Primitive* (or atomic) *models* are those not composed of other submodels: they contain the equations describing the behavior of the system. *Composed* (or molecular) *models* are those formed by the connection of a number of primitive or composed submodels.

The *object-oriented modeling methodology* is based in the three conceptual elements described previously: abstraction, information encapsulation and modularization. Other important concepts are *class* and *class instantiation* [12, 24]. A *class* is the description of a set of objects with similar properties. Models are represented as classes instead of as class instantiations, because usually a model is the description of a kind of system instead of the description of a particular system. The simulation is carried out on an *instantiation* of the model.

Composition and *specialization* are the two fundamental ways of reusing the model. *Composition* is the capability of defining new models as the connection between the previously defined models. *Specialization* is the capability of defining a new model by the specialization of other previously defined models. *Inheritance* is a way of information sharing by specialization: the *subclass* inherits all the attributes of its *superclass* (or superclasses in case of *multiple inheritance*). Therefore, the subclass can be considered as a refinement or specialization of the general concept defined in the superclass, to which new components or equations are added.

Parametrization is a key concept related to reusing the models. A parameter of a model is any of its properties that can be modified in order to adapt the model to its different applications. In a broader sense, a parameter may be as simple as a variable or as complex as a submodel structure [11]. Another related concept to reusing models is *polymorphism*. Two models are polymorphic when their interfaces have the same structure and they have an equal number of freedom degrees (that are defined as the difference between the total number of variables and the number of equations). Polymorphism is a first condition to be satisfied by models that can be used in the same context and that can be exchanged without modifying the rest of the system [8]. Frequently, polymorphic models have a superclass in common: their interface.

1.2. Contributions of this Paper

The use of an appropriate *design methodology* is a key element to guarantee the success of any modeling task. In particular, the design of model libraries dedicated to

the composition of simulation models based on predefined model elements. In this context, a distinction can be made between the library designers/programmers and the library users, who use the library to compose models to fulfill their necessities. Therefore, the *library design methodology* should define: (1) the relationship between the designers/programmers and the users; and (2) the role of the library designers/programmers.

The design methodology should take into consideration all the phases of the libraries life cycle: design, programming, maintenance, modifications, extension, etcetera. Modeling languages provide the resources to apply the object-oriented modeling principles. However, the application of these principles to achieve the best possible results depends on the *library design methodology* adopted by the design group.

In addition, the *library design methodology* should take into consideration the numerical aspects of the model simulation. The current modeling environments capabilities could erroneously suggest that the library designer can depend totally on the modeling environment, without considering the numerical aspects of the model. In other words, it could be erroneously assumed that the model designer can focus on carrying out a declarative description of the different system submodels and the modeling environment is in charge of the rest. However, frequently this is not so. A model library could be useless if the models obtained from their composition and specialization have poor numerical properties. In practice, these considerations may decisively condition the model library design.

A possible *object-oriented design methodology* that attempts to cover all the previously mentioned aspects is proposed in this paper. A distinction is made between library designers/programmers and library users. One of the main goals of the proposed methodology is to guarantee that the users are able to use the model libraries without having to know their internal details. In particular, the users should not be confronted with numerical problems. In order to have a well posed and efficient model description for simulation, the model library code needs to be designed in the ‘‘correct way’’ from the start [26]. To achieve these objectives, non-declarative techniques are recommended in the library design phase: causality analysis, explicit selection of tearing variables, decoupling via dummy dynamics, etc. This proposed approach implies considerable modeling effort compared with a declarative description of the models.

This aspect of the proposed design methodology contradicts the thinking of a part of the object-oriented modeling and simulation community. These researchers support the declarative modeling of physical systems, that is, not taking causalities, tearing or index problems into account. Their approach assumes that the existing modeling environments allow facing the numerical problems purely by numerical means, that is, symbolic manipulation of the set of equations and dedicated numerical algorithms. According to this point of view, the design methodology proposed in this paper could be considered ‘‘a step backward’’.

The library design methodology presented in this manuscript is mainly based on the modeling experience of the authors in the physical-chemical field, an application domain where the numerical problems can become critical. This fact has undoubtedly conditioned the development of the methodology. Hopefully, some of the ideas presented in this manuscript will help the reader who has particular modeling and simulation problems, to find an adequate design methodology.

2. NUMERICAL ASPECTS OF THE MODEL LIBRARIES DESIGN

The numerical aspects of the class library design are discussed: (1) the problems related to the numerical solution of nonlinear systems of simultaneous equations (abbreviated: nonlinear SSE); (2) the problems related to the establishment of high-index problems (i.e., models with index greater than one); (3) the appropriate selection of the state variables; (4) the intensive variables definition in order to avoid numerical errors during the simulation; (5) the problem of small oscillations crossing an event condition; (6) the need of extending the definition of the functions with several branches; (7) the need to express the model equations so that they have good numerical properties for all the computational causalities required for their interface; and (8) the infinite chains of events without any termination stage.

2.1. Numerical Solution of Nonlinear Systems of Simultaneous Equations

The intrinsic problem with the numerical solution of nonlinear SSE is that depending on how we iterate the system of equations, we may end up with one solution, or another, or none at all. Unfortunately, no general applicable method can be found that would deal with this problem once and for all [27].

Tearing is a technique to facilitate the numerical solution of certain type of nonlinear SSE. It is applicable to nonlinear SSE in which at least one of the unknown variables appears linearly in one equation. The purpose of the technique is, taking advantage of the previous fact, to reduce the number of variables to iterate for solving the SSE [22, 23]. The tearing of the nonlinear SSE $\mathbf{G}(\mathbf{y}) = 0$ can be viewed in the following way: $\begin{cases} \mathbf{G_1}(\mathbf{y_1}, \mathbf{y_2}) = 0 \\ \mathbf{G_2}(\mathbf{y_1}, \mathbf{y_2}) = 0 \end{cases}$, so that the next two conditions are satisfied:

1. The *Jacobian matrix* $\frac{\partial \mathbf{G_1}}{\partial \mathbf{y_1}}$ is lower triangular or lower triangular by blocks. All the diagonal blocks represent equations (scalar block) or SSE (non scalar block) in which the unknown variables to be evaluated from them appear linearly.
2. The dimension of $\mathbf{y_2}$ is as small as possible.

Once the SSE is written in this way, then only $\mathbf{y_2}$ is calculated by iteration. Having a known initial value of $\mathbf{y_2}$ then $\mathbf{y_1}$ can be calculated, solving $\mathbf{G_1}(\mathbf{y_1}, \mathbf{y_2}) = 0$ by means of techniques for linear SSE. With this value of $\mathbf{y_1}$, a new value of $\mathbf{y_2}$ is obtained iterating

$\mathbf{G_2}(\mathbf{y_1}, \mathbf{y_2}) = 0$, and so on. The unknown variables $\mathbf{y_2}$ are called the *tearing variables*. The equations $\mathbf{G_2}(\mathbf{y_1}, \mathbf{y_2}) = 0$, used to evaluate the tearing variables, are called *residue equations*. Some modeling environments support a more restrictive concept of tearing [22, 23]: the *Jacobian matrix* $\frac{\partial \mathbf{G_1}}{\partial \mathbf{y_1}}$ is lower triangular (i.e., all the diagonal elements are scalar).

Commonly, there is more than one possible selection of residue equations and tearing variables in a nonlinear SSE. Actually there is not an efficient algorithm to carry out automatically the optimal selection of the tearing variables and the residue equations [22]. Nevertheless, most object-oriented modeling environments incorporate algorithms to automatically propose one way (of the several possible ways) of tearing the nonlinear SSE. In recognition to the fact that other (possibly more efficient) tearing strategies could be feasible, some modeling environments allow the user to modify this tearing default option. A factor to consider when choosing the tearing variables is the relative ease to assign them accurate initial values. The residue equations selection is based on their good numerical properties, in order to facilitate the iterative algorithm convergence.

Two types of nonlinear SSE can be distinguished from the design methodology point of view: those internal to the class description and those established when connecting the classes. The library designer can study the numerical properties of the *nonlinear SSE internal to the class description*, in order to find an efficient tearing strategy or maybe to conclude in the need of reformulating the model. Some modeling languages (e.g., Dymola [28]) support the inclusion in the model class definition of tearing information. In contrast, the Modelica 2.0 language does not support this capability [11]. However, the tearing information capability is very useful in the author's opinion because it supports the consideration of information that is possibly critical for the numerical solution.

The numerical properties analysis of the *SSE established when connecting the classes* is problematic from the design methodology point of view. The library designers should be able to develop each part of the model (i.e., each class of the model library) without knowing the internal details of the other parts of the model. In case of *SSE established when connecting the classes*, including tearing information in the classes implies knowledge of the internal description of the other classes, which is clearly contrary to the abstraction principle. Therefore, the library designer should not be in charge of this type of analysis (and even less the user). In consequence, the numerical solution of these SSE is left completely in hands of the modeling environment. If any of these nonlinear SSE (that nobody has previously analyzed) has poor numerical properties, then the simulation run will be inefficient or even impossible. Reached this point, the user (ignorant of the internal details of the model and unable to understand the numerical problem diagnosis shown by the modeling environment) has not the resources to solve this problem. In consequence, this problem should be faced at the library design phase. This is one of the reasons because

the proposed design methodology avoids the establishment of SSE when connecting the classes. A second reason is discussed next: the convergence problems during the re-start problem solution.

One important difficulty of the hybrid models simulation is the numerical solution of the nonlinear SSE. It is especially critical at the model initialization and at the restart problem solution (i.e., when solving the model after the discrete-time events execution). Some modeling languages and environments (for instance, Modelica 2.0/Dymola 5.0 [11, 29–31]) provide effective support to the user in order to face the initialization problem. At the end, the success in solving the initialization problem depends on the user ability to provide initial values ''close enough'' to the true values. The re-start problem is similar to the initialization problem, but in this case a trial-and-error method is not applied to determine the appropriate initial values. After the discrete-time event execution, the initial values for iterating the nonlinear SSE are the variable values just before the event execution. When these initial values are not appropriate, the iteration does not converge and the simulation is aborted. The solution to fix this numerical problem (faced by the library user) requires the model reformulation (an attribution of the library designer). In consequence, the library design methodology should prevent the user from being confronted with this type of numerical difficulties.

In order to avoid the establishment of (potentially nonlinear) SSE when connecting the classes, the proposed design methodology states that:

1. The library designer should guarantee that the computational causality of the class interface variables does not change during the simulation run. A sufficient condition (it is not necessary) for the establishment of a SSE when connecting a class is that the computational causality of some of its interface variables changes during the simulation run (i.e., they change from acting as computational inputs to outputs or vice versa). When the variable to evaluate in a given equation changes during the simulation run, then the equation is part of a SSE [32, 33].
2. The library designer has to break the computational causality loops for each of the allowed computational causalities of the class interface. Integrators and time-dependent variables are used to complete this task. This point is explained below.

The SSE frequently represent modeling approximations arising when considering ''instantaneous'' the dynamics which are much faster than those relevant for the simulation purpose [22]. The designer always can opt for describing the dynamic, instead of considering it instantaneous, avoiding the establishment of the SSE. The dynamic can be arbitrarily modeled, but it should satisfy the following two conditions:

1. It should be fast enough, in comparison with the simulation relevant dynamics, to be irrelevant for consideration.
2. It should not be faster than strictly necessary, so that it does not introduce unnecessarily large eigenvalues. This avoids the system becoming too stiff.

Example Consider the macroscopic linear momentum balance at steady state applied to a liquid mixture in a pipe. P represents the liquid linear momentum and f_{total} is the sum of the forces acting on the mixture (pressure, friction, gravity, etc.) in the motion direction. The steady-state balance formula, $0 = f_{total}$, means that the fluid motion follows instantaneously the applied force, so it generally leads to the establishment of a SSE [34]. An alternative approach is to consider that the fluid inertia is very small, negligible for the simulation purpose: $\varepsilon_P \cdot \frac{dP}{dt} = f_{total}$, where $\varepsilon_P \ll 1$. Therefore, the fluid linear momentum acts as a state variable, breaking the computational causality loop. The user of the model has to assign a value to ε_P in correlation with the model application [34].

2.2. High-Index Problems

There are no reliable general-purpose numerical DAE solvers for high-index problems (i.e., with index greater than one) [15, 30]. As a consequence, the model index must be reduced to zero or one in order to simulate more efficiently the model. The index reduction procedure [16, 17] introduces some changes in the model formulation: certain model equations are symbolically differentiated and these new equations are included in the model. The library designer should study the numerical properties of the model obtained after the index reduction, in order to either verify its appropriate numerical properties or reformulate the model.

To this respect, two types of high-index problems can be distinguished: those internal to the class description and those established when connecting the classes. The library designer can study (with the help of the modeling environment) the *high-index problems internal to the class*. As a consequence, this type of high-index problems is useful from the design methodology point of view. On the contrary, the *high-index problems established when connecting the library classes* cannot be studied by the library designers during the design phase (according to the abstraction principle) and, consequently, they should be avoided.

Class connections lead to high-index problems when any of the connection equations only contain: (1) variables that appear differentiated in the internal description of the classes; and, possibly, (2) variables that must be evaluated from other equations (because they are the only unknown variables in these equations). This type of connection equations introduces constraints on variables that appear differentiated. As a consequence, the number of state variables is smaller than the number of differentiated variables. This kind of constraint arises due to a class connection only if all the variables intervening in the connection equation are evaluated in their respective classes (i.e., all the variables are computational outputs in their respective classes). This fact can be avoided proposing rules about the classes' connection, keeping in mind the computational causality of the interface variables. The proposal of the *interface-connection rules* is one of the design phases established by the proposed design methodology.

Although this topic will be discussed in detail later, it is necessary to indicate that frequently the connection rules condition the modeling. One technique to formulate the model so that it fulfills the constraints imposed by its interface is the *constraint relaxation technique* [34]. Instead of forcing the system to evolve instantaneously in the desired way (introducing constraints on their state variables), it is possible to model an arbitrary dynamic in order to make the system evolve "almost instantaneously" toward the desired trajectory. The following example tries to illustrate this point.

Example Consider two control volumes, A and B, each one containing an ideal mixture of semiperfect gases. At any time, a gas-flow between both control volumes is established equaling their pressures instantaneously (for simulation purposes). The system is modeled by means of the following two classes:

1. A class models the molar balance, the energy balance and the state equation of the semiperfect gases in the control volume. The temperature and the number of moles of each component appear differentiated in the balances. The mixture pressure only intervenes in the state equation. Both control volumes are modeled by means of two instantiations of this class.
2. Another class models the gas-flow between the control volumes. It can be modeled by imposing the pressure equality: $p_A = p_B$.

As the pressure intervenes in the gas state equation, together with the temperature and the number of moles, and these appear differentiated in the balances, then the problem has an index greater than one. When carrying out the index reduction, a nonlinear SSE is established. The number of equations of this SSE increases with the number of components of the gas mixture and with the number of control volumes.

As the gas-flow dynamic is very fast compared with the relevant dynamics of the model, an alternative approach consists in modeling arbitrarily the dynamic (instead of considering it instantaneous), but satisfying the condition of being fast enough to be considered "almost instantaneous" for the simulation purpose. The dynamic must not be faster than strictly needed, in order to avoid the system to be unnecessarily stiff. For instance, a solution [34] consists in modeling the gas-flow between the control volumes A and B, $\hat{F}_{A\to B}$, as: $\hat{F}_{A\to B} = k_{prop} \cdot (p_A - p_B)$. In other words, the molar flow is defined (arbitrarily) proportional to the pressure difference. The library user has to assign the smallest possible value to the parameter k_{prop} from the range of the dynamic which is considered instantaneous. In order to assist the user in this selection, a variable is defined with value $\frac{|p_A - p_B|}{0.5 \cdot (p_A + p_B + \varepsilon)}$. It measures the relative error made when considering that the pressure difference is zero. The purpose of the parameter ε is explained in Section 2.4.

2.3. State Variables Selection

Different choices of the state variables may lead to drastically different numerical behavior [30, 31]. The library designer should take into account this fact when

selecting the state variables. Another important criterion is the relative ease in finding appropriate initial values. A model with state variables whose initial values are unknown could be useless. However, in some situations the designer does not know ahead of time which the optimal selection of the state variables is. In these cases, the designer should offer the user the possibility of choosing the state variables of the model: the user makes his selection among a given set of variables selected previously by the designer. This selection is not a model characteristic: it is a characteristic of the experiment, and it can change from one simulation run to another.

Some modeling languages (e.g., Modelica 2.0 [29, 31]) support language constructions in order to include in the model information about the state variables selection. The modeling environments supporting these languages (e.g. Dymola [30] supporting Modelica) allow the user to select the state variables in accordance with the criteria described in the model. These capabilities are useful for model designers and users. When the modeling environment does not support this type of capability, but support index reduction, the designer can use certain techniques to allow the state variables selection. One of these techniques is discussed below [35].

The designer has to add to the original model as many dummy equations and dummy variables as algebraic variables he wants to transform into state variables. Each dummy equation contains the derivative of one of these variables (i.e., the new state variables) and a dummy variable. These equations do not modify the solution of the system: they define the dummy variables. However, these dummy equations make the index of the model to be greater than one. The next step is to reduce the index of the system to one or zero (using the modeling environment). Once the modeling environment has added to the system the derivative of certain equations [16, 17], then the designer or the user may specify which of the differentiated variables have to be considered as state variables and which not. If the designer knows the optimal selection of the state variables, then he can substitute the derivative of the old state variables by algebraic variables (substitutions of the type: $\dot{x}$ by *derx*). Otherwise, he may add the dummy equations to the model and let the user make the index reduction (using the modeling environment) and choose the state variables. Next, an example of the application of this technique is discussed.

Example Consider the following model:

$$\dot{x} = u_1(t) - u_2(t) \cdot (y_1 - u_3(t)) \tag{1}$$

$$x = y_1 \cdot y_2 \tag{2}$$

$$y_2 = f(y_1) \tag{3}$$

The time-dependent functions $u_1(t)$, $u_2(t)$ and $u_3(t)$ are known. The unknown variables are x, y_1 and y_2. As x appears differentiated, the partition algorithm considers that it acts as a state variable [5]. Consequently, its derivative, $\dot{x}$, is replaced by an auxiliary

variable, *derx*, and x is considered to be calculated by integration of *derx*. The result of the computational causality assignation is:

1. The variables y_1 and y_2 are calculated from the nonlinear SSE that form the Equations (2) and (3).
2. *derx* is calculated from the Equation (1).

However, lets suppose that it is preferable y_1 to be the state variable of the system instead of x. In order to get it, a dummy equation must be added to the system. It has to contain the derivative of y_1 (i.e., $\dot{y}_1$) and it should not modify the system solution. A possible selection is, for instance, $\dot{y}_1 = \alpha$ [35]. The dummy variable α only appears in this dummy equation, so the equation is used to evaluate α. The system, extended with the dummy equation, has an index greater than one: it contains one degree of freedom and two differentiated variables, x and y_1. Once the index reduction has been carried out [16, 17], the modeling environment allows the user to choose which variable (x or y_1) must be considered as state variable. Choosing y_1 as state variable, the following formulation of the system is obtained:

$$derx = u_1(t) - u_2(t) \cdot (y_1 - u_3(t)) \tag{4}$$

$$x = y_1 \cdot y_2 \tag{5}$$

$$y_2 = f(y_1) \tag{6}$$

$$\dot{y}_1 = \alpha \tag{7}$$

$$derx = \dot{y}_1 \cdot y_2 + y_1 \cdot dery_2 \tag{8}$$

$$dery_2 = \frac{\partial f}{\partial y_1} \cdot \dot{y}_1 \tag{9}$$

2.4. Extension of the Intensive Variables Definition

Mass, energy and linear momentum balances can be proposed in terms either of the extensive properties (mass, moles, internal energy and linear momentum) or the intensive properties (density, temperature and velocity). There is one intrinsic difficulty associated with the intensive formulation: the intensive variables have no meaning when the amount of matter is zero. This is a potential source of run-time numerical problems.

A way of avoiding this problem is extending the intensive variables definition to the ''zero-mass'' case. The usual relationship between the intensive (y) and the extensive (Y) variables, $y = \frac{Y}{m}$, should be substituted by $y = \frac{Y}{m+\varepsilon}$, where ε is a positive constant with a very small value. As a consequence, the $\frac{0}{0}$ singularity is transformed into $\frac{0}{0+\varepsilon} = 0$. As the mass is a non-negative magnitude, there is not risk of dividing by zero. The value chosen for ε should be small enough to modify the intensive variable definition

only for values close to zero. However, the chosen value of ε should not be smaller than strictly needed. The very small values of ε do not avoid the numerical errors. This method is applicable whenever the variable in the denominator takes only either non-negative or non-positive values.

2.5. Small Oscillations Around an Event Activation Condition

Depending on its trigger condition, events can be classified into *time-events* and *state-events*. The procedure to detect the event trigger instant depends on its type. In the case of the *time-events*, the detection procedure is trivial. Once the simulation algorithm detects that the closest time-event is produced in the next integration step, the step size is reduced so that the evaluation instant and the time-event are the same [19].

State-events are triggered when the system satisfies certain conditions. Their trigger instants are unknown in advance and they have to be calculated by iteration (the problems associated with their calculation by step size reduction are discussed in [18–20]). In order to understand how the integration algorithm calculates the state-event trigger instant, consider a function defined by means of different branches. Each branch is valid within a given domain of the state space. State-events indicate the end of the validity domain of a branch and the start of the validity domain of another branch. Boolean *invariant functions* are defined to detect the events, signaling the state trajectory crossing from one domain to another. These invariant functions split the state space into two parts: the set of admissible states, where all are true, and the remaining states, where at least one of them is false. When the system crosses the boundary defined by these invariant functions, going out of the admissible state domain, an event is triggered.

Event conditions are defined as the Boolean complements of the invariant functions [12]. Event conditions that depend on at least one continuous variable are called continuous conditions. Those that only depend on discrete variables are called discrete conditions. Discrete variables are those that only change their value in certain instants of time. *Crossing functions* are used for detecting state events of continuous conditions. The event condition is formulated in the following way: $z > 0$. For example, the procedure employed by Dymola [18] consist in defining a small interval $(-\varepsilon ps, \varepsilon ps)$ around zero (the value of εps is typically 10^{-10}). The event trigger condition is: either z crosses $-\varepsilon ps$ with a negative slope or it crosses εps with a positive slope. When, due to the initial conditions, the crossing function remains within the interval $(-\varepsilon ps, \varepsilon ps)$, the crossing function value is considered zero and the corresponding branch of the *if-then-else* expression is used.

Modeling languages provide clauses of the kind $\langle expression \rangle = if\ \langle condition \rangle\ then\ \langle expression1 \rangle\ else\ \langle expression2 \rangle$ in order to describe functions with several branches [6, 7, 11, 12, 27]. The modeling environment uses invariant functions to represent the Boolean conditions of such clauses, so that a discrete event is triggered when the

Boolean value of a condition changes. If it is a state-event of continuous condition, when the trigger condition is detected (by means of its crossing function), the integration is suspended and the iteration to find the event instant starts.

However, sometimes the exact determination of the trigger instant is not crucial for the purpose of the simulation. In these cases, it may be preferable to sacrifice the precision in the trigger instant determination in order to increase the simulation speed. In particular, this may be the case when the system describes small oscillations across the event condition. An option is to consider that the event instant coincides (approximately) with the instant in which the invariant function change is detected. Some modeling languages (e.g., Modelica [11]) allow distinguishing between those *if-then-else* functions in which it is necessary to iterate in order to find the event instant and those in which it is preferable not to do so.

If the modeling language does not allow making this distinction and if the designer does not want to make an event-based treatment of certain functions with several branches, then the designer has to formulate the model to such an end. This situation can arise in the case of state events of continuous conditions. A general procedure is described below in order to represent a function with two branches without using *if-then-else* clauses. Applying the proposed formulation, the transition between the branches takes place when the value of the associated invariant function would have changed if an *if-then-else* clause has been used.

Consider the following general function of two branches:

$$y = \text{if } z > 0 \text{ then } x_1 \text{ else } x_2 \tag{10}$$

An equivalent definition, except for the method used for the event-instant detection, is:

$$c = \frac{z}{\text{abs}(z)} \tag{11}$$

$$y = \frac{1}{2} \cdot (1 + c) \cdot x_1 + \frac{1}{2} \cdot (1 - c) \cdot x_2 \tag{12}$$

In order to avoid an indetermination $\frac{0}{0}$ when z is zero, the technique described in Section 2.4 can be applied. It is obtained:

$$c = \frac{z}{\text{abs}(z) + \varepsilon} \tag{13}$$

$$y = \frac{1}{2} \cdot (1 + c) \cdot x_1 + \frac{1}{2} \cdot (1 - c) \cdot x_2 \tag{14}$$

The extension of the method to functions with more than two branches is trivial. Any function of several branches can be written as a set of functions with two branches. For example, the following function of three branches:

$$y_1 = \text{if } z_1 > 0 \text{ then } x_1 \text{ else if } z_2 > 0 \text{ then } x_2 \text{ else } x_3 \tag{15}$$

It can be written as follows:

$$y_1 = \text{if } z_1 > 0 \text{ then } x_1 \text{ else } y_2 \tag{16}$$

$$y_2 = \text{if } z_2 > 0 \text{ then } x_2 \text{ else } x_3 \tag{17}$$

2.6. Branches Extension

The iterative method used to find the event instant requests the function branches to be extended beyond their validity domain. This point is discussed by means of the following example.

Example Consider the steady-state macroscopic balance of linear momentum applied to the flow of liquid into a straight pipe. The liquid velocity satisfies that the friction force plus the pressure force plus the gravity force equals zero. f represents the sum of the components in the movement direction of the pressure and gravity forces. F represents the flow of liquid. A way of writing the balance model consists in defining a Boolean variable that represents the flow direction, $f > 0$, and expressing the liquid flow as a function with two branches as follows:

$$F = \text{if } f > 0 \text{ then } k \cdot \sqrt{f} \text{ else } -k \cdot \sqrt{-f} \tag{18}$$

Expressing the momentum balance in this way produces a numeric run-time error when the sign of f changes. The simulation algorithm detects the event trigger (in this case, the change of $f > 0$ from true to false or vice versa) once all the algebraic variables have been calculated in the evaluation instant. In other words, once the calculation attempt of the square root of a negative number has been made (producing the consequent error). For this expression to work, the simulation algorithm should check if the event $f > 0$ has been triggered after calculating f and before calculating F. Obviously, the simulation algorithm does not work in this way. It is advisable, in order to avoid this kind of problem, to allow the function branches to be evaluated beyond their definition domain. In this example, this can be achieved by means of the absolute value function, abs( ), provided by the modeling language:

$$F = \text{if } f > 0 \text{ then } k \cdot \sqrt{\text{abs}(f)} \text{ else } -k \cdot \sqrt{\text{abs}(f)} \tag{19}$$

Writing the balance in this way indicates to the modeling environment that every time f sign changes, the integration must be suspended and it must start the iterative process to find the event instant. The iterative process uses the ''old'' branch of the function. Once the event instant is found (with the required precision), the model variables are recalculated using the ''new'' branch of the function. Next, the integration algorithm is restarted (in case no more events are triggered).

This approach is time-consuming if the value of f describes small oscillations around zero. If the accurate knowledge of event instants is not relevant for the

simulation purpose, this is not an adequate formulation of the momentum balance. A more adequate formulation is obtained rewriting the expression as indicated in the previous section:

$$c = \frac{f}{\text{abs}(f) + \varepsilon} \tag{20}$$

$$F = c \cdot k \cdot \sqrt{\text{abs}(f)} \tag{21}$$

2.7. Computational Causality of the Interface

According to the proposed design methodology, the design of an interface does not only consist on the definition of: (1) the variables of which it is composed; and (2) the type of their connection equations. In addition, the *allowed computational causalities of the interface* should be specified. In order to clarify the meaning of ''allowed computational causalities of the interface variables,'' consider the following example.

Example An interface is composed of three variables: x, y and z. It has $2^3 = 8$ *possible* computational causalities. However, the design-specifications of the interface dictate that it is going to be used in contexts that require one of the two following situations: (1) x and y inputs, z output; and (2) x and z inputs, y output. As a consequence, the interface has two *allowed* computational causalities. Any other interface causality is not allowed.

The library designer should formulate the system internal description so that it has appropriate numerical properties for all the allowed computational causalities of its interface. Therefore, the allowed causalities of its interface decisively condition the formulation of the model internal description. For instance, when any of the interface computational causalities leads to the establishment of a SSE internal to class, the designer should include the tearing information in the class code. Responding to this necessity, some modeling languages (e.g., EcosimPro [7]) allow different formulations for each expression of the continuous part of the model, depending on the variable to be evaluated. The formulation used by the modeling environment in each instantiation of the class depends on the variable to be evaluated of the expression in each case.

In addition, the designer should propose rules in order to guarantee that the models are not used in contexts requiring non-allowed computational causalities of their interfaces. This can be achieved by means of the interface connection rules. The following example tries to illustrate how the interface conditions the internal formulation of the system.

Example Continuing the example posed in Section 2.6, suppose that the linear momentum balance has to be formulated in a way that it facilitates: (1) the calculation of f from F; and (2) the calculation of F from f. In this case, the formulation by means of the Equations (20) and (21) is not appropriate. Since the inverse of the absolute

value function is not monovaluated, this model has been written implicitly assuming that f is known and then c is calculated of Equation (20) and F from the Equation (21). Next, a model modification is shown in order to allow both computational causalities: (1) F unknown, f known; and (2) f unknown, F known.

$$c = \frac{f}{absf + \varepsilon} \tag{22}$$

$$F = c \cdot k \cdot \sqrt{absf} \tag{23}$$

$$absf = \text{abs}(f) \tag{24}$$

The calculus sequence of the two possible causalities is the following:

1. F is unknown and f is known. *absf* is computed using Equation (24). c is calculated from Equation (22). F is derived from Equation (23).
2. f is unknown and F is known. The Equations (22)–(24) form a system of three simultaneous equations with three unknown variables. The purpose of introducing the variable *absf* is to limit the appearance of the absolute value function to only one equation. This makes the tearing of the SSE easier. The tearing has to satisfy that the loop iteration does not require the evaluation of the inverse of the absolute value function. In the Equation (24), the variable *absf* is monovaluated, while the variable f is bivaluated. So, the loop has to be iterated in a way that the Equation (24) is used to evaluate *absf*. Two possible ways of making the tearing satisfy this condition are: (1) tearing variable: f, residue equation: Equation (22); and (2) tearing variable: *absf*, residue equation: Equation (24).

2.8. Infinite Propagation of the Events

A problem the designer has to face when modeling hybrid systems is the infinite propagation of events. This effect occurs when every execution of event actions triggers one or more new events, so that the event iteration does not terminate. There is not a general systematic procedure to detect this kind of problem during the design phase. Consequently, its detection is completely based in the designer's knowledge of the model. Next, an example of infinite events propagation is discussed.

Example A simple model of the macroscopic momentum balance applied to a pipe completely full of liquid is shown in Table 1. The linear momentum acts as a state variable. The fluid density, ρ, and its viscosity, μ, are considered constants. The force due to the pressure difference between the pipe ends, f_p, is a known function of time. The variable to calculate for each equation (assuming that P acts as state variable) is signaled between brackets.

The Fanning friction factor is formulated as a function with two branches of the Reynolds number. The condition $\text{Re} < 2100$ determines the end of the validity

Table 1. Macroscopic linear momentum balance applied to a pipe completely full of liquid.

$P = S \cdot \rho \cdot [v] \cdot L$	Relationship among the fluid linear momentum, the pipe section, the fluid density, its velocity and the pipe length.
$[\mathrm{Re}] = \frac{D \cdot v \cdot \rho}{\mu}$	Reynolds adimensional number for a circular-section pipe.
$[\kappa_{Fanning}] = \begin{cases} \frac{16}{\mathrm{Re}}, & \text{if } \mathrm{Re} < 2100 \quad \text{(laminar)} \\ \frac{0.0791}{\mathrm{Re}^{1/4}}, & \text{if } 2100 < \mathrm{Re} < 10^5 \quad \text{(turbulent)} \end{cases}$	Fanning friction factor for laminar and turbulent flow in a long, smooth and circular pipe (Blasius formula).
$[f_F] = \begin{cases} -S_w \cdot \frac{1}{2} \cdot \rho \cdot v^2 \cdot \kappa_{Fanning}, & \text{if } P \geq 0 \\ S_w \cdot \frac{1}{2} \cdot \rho \cdot v^2 \cdot \kappa_{Fanning}, & \text{if } P < 0 \end{cases}$	Friction force exerted by the pipe wall. S_w represents the internal surface of the pipe wetted by the liquid.
$\varepsilon_P \cdot \left[\frac{dP}{dt}\right] = f_F + f_p$	Steady-state momentum balance. The value of ε_P has to be small enough in order to produce a fast enough dynamic to be considered ''instantaneous'' for the simulation purpose.

domain of a branch and the start of the validity domain of another. The problem generated by this formulation of the Fanning friction factor is that finding a restart state after the branch commutation event may be impossible. Suppose the flow is laminar and the velocity increases, in such a way that the Reynolds number becomes larger than its critical value, $\mathrm{Re}_c = 2100$. The Fanning factor value at Re_c is bigger for turbulent flow, $\frac{0.0791}{2100^{1/4}}$, than for laminar one, $\frac{16}{2100}$. Consequently, the friction force is stronger for the turbulent flow than for the laminar. The friction increment, due to the transition from laminar to turbulent regime, produces a reduction of the Reynolds number. If the inertia of the fluid in the pipe is small, this reduction may be large

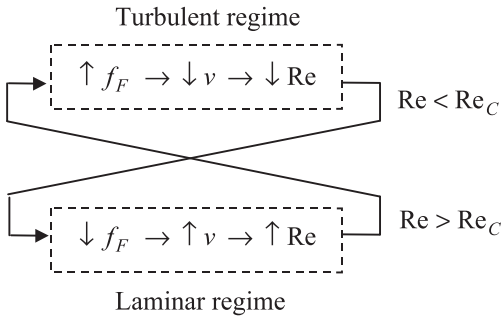

Fig. 1. Cycle established at regime change.

enough to make the Reynolds number smaller than its critical value (the system goes back to the laminar regime). This change to laminar regime makes the friction force decrease, increasing the fluid velocity and the Reynolds number. This increment in the Reynolds number makes the system change again to the turbulent regime, and so on. The cycle previously described is represented in Figure 1. A possible solution to this problem is modifying the Fanning factor definition. For instance, a transition zone may be defined between the laminar and the turbulent regime, so that the Fanning factor is a continuous function of the Reynolds number [34].

3. DESIGN METHODOLOGY

The proposed design methodology is based on the object-oriented modeling principles. A distinction is made between library designers/programmers and library users. One of the main goals of the proposed methodology is to guarantee that the users are able to use the model libraries without having to know their internal details. In particular, the users should not be confronted with numerical problems. By applying the proposed design methodology, the following topics are answered:

1. What the full set of requirements for each class is.
2. How a class can be used and what classes are interchangeable.
3. How the libraries can be extended with new classes.

Next, the five steps composing the design methodology are discussed. The key concept of the methodology is defined: the *design rules of the libraries*. Finally, the role of the designers and their relationship with the library users are defined.

3.1. First Step – Phenomena and Hypotheses

The first step of the design of a set of mathematical model libraries consists in:

1. Decide the phenomena to model.
2. Definition of the general modeling hypotheses.

Frequently, the same set of phenomena has to be modeled with different degrees of detail and using different sets of hypotheses. The designer should document the different sets of hypotheses as well as stating which corresponds to each model. He has to decide whether the exchange between the different models of a given phenomenon has to be made in the simulation run or, on the contrary, the modeling hypotheses do not change during the simulation run. In this last case, it makes sense to group different-hypothesis models in different libraries. Once the pairs (*phenomenon to model – modeling hypothesis*) have been identified, these should be grouped into libraries.

3.2. Second Step – Phenomena Interaction: Causal Connectors

The second step of the design consists of the basic elements definition needed for describing the interaction between the different phenomena to be modeled (i.e., the interface basic components). According to the proposed design methodology, these basic elements are the *causal connectors*. They are an extension of the *connector* concept of the modeling languages (also called cut, port, etc.). A *connector* is a class type intended to facilitate the description of the model connections.

Definition A *causal connector* class is defined by means of the four following pieces of information:

1. *The variables composing the connector.* They must be classified into *across* variables and *through* variables (Dymola [28] terminology is adopted). *Across* variables must be equaled, at a connection point. *Through* variables (also called *flow* variables) must sum to zero, at the connection point. This interface variables classification into across and through is not an intrinsic property of the variables, but it is determined by the modeling strategy [36]. The following example tries to illustrate this point.
   Example When modeling electrical circuits, voltage can be selected as an across variable and current as a through variable. This selection leads to the node equations of the circuit: (1) a voltage is assigned to each node; and (2) the sum of the currents going into each node is zero. The dual model consists in considering the voltage as a through variable and the current as an across variable. This approach leads to the net equations of the circuit: (1) a current is associated with each net; and (2) the sum of the voltage drops around each net is zero.
2. *The allowed computational causalities of the connector class.* An allowed computational causality is defined classifying the causal connector variables into computational inputs and outputs. The design methodology allows a causal connector to have multiple allowed computational causalities, but the computational causality of each causal connector instantiation should not change during the simulation run. This restriction is imposed in order to avoid the establishment of SSE when connecting the classes.
3. *The way of breaking the computational causality loops.* In this context, a variable is said to break a computational causality loop when it is calculated exclusively as a function of time and/or state variables.
4. *A graphic icon*, in order to facilitate its identification by the user.

Modeling languages support the description of the connector variables, their type (i.e., across or through), and the graphical icon. The allowed computational causalities of the causal connector and the information about what connector variables must break

the computational causality loops are part of the class documentation. Causal connectors should be grouped into libraries defined only to this purpose.

Example Two causal connector classes are defined for modeling the flow of a liquid mixture [34]: *liquidFlow_R* and *liquidFlow_C*. They have five variables: (1) *Across variables*: Mass (vector), Temperature and Pressure; and (2) *Through variables*: MassFlow (vector) and EnergyFlow. Each connector class has 5 variables and consequently it has 32 *possible* computational causalities. The proposed design methodology allows a connector class to have several *allowed* computational causalities. However, in this case, the library designer decides that these connector classes have only one allowed computational causality and the result is the following:

1. Connector class: *liquidFlow_C*. The across variables are computational outputs and they are only a function of state variables and/or time (i.e., they break the computational causality loops). The through variables are computational inputs.
2. Connector class: *liquidFlow_R*. The across variables are computational inputs. The through variables are computational outputs.

This information is included in the documentation of the connector classes. In addition, the library designer should propose *connection rules* in order to guarantee that only the allowed computational causalities are required.

3.3. Third Step – Connection Rules of the Causal Connectors

The third step of the design consists in defining the rules for the causal connectors connection. The connection rules state the nature and number of the allowed connections. Based on the casual connectors definition, the connection rules must guarantee that:

1. The causal connector causality required by the connection is one of its allowed causalities.
2. The complete system is not overdetermined or underdetermined.
3. Causal connectors connections do not lead to the establishment of SSE.
4. Causal connector connections do not lead to high-index problems.

The formulation of the connection rules is different depending on whether the connection is made between *same-hierarchy causal connectors* or between *different-hierarchy causal connectors*. In this context, two classes (and, consequently their causal connectors) are said to be at different hierarchical levels when one is a composed-class and it contains the other class as a submodel. In order to facilitate the fulfillment of the connection rules, they should be described in terms of the graphic attributes of the causal connectors.

Example Connection rules states the nature and number of the connections among *liquidFlow_R* and *liquidFlow_C* connectors [34]:

- Connection between causal connectors of the same hierarchy:
  1. A *liquidFlow_C* connector can be left unconnected or it can be connected to an arbitrary number of *liquidFlow_R* connectors.
  2. A *liquidFlow_R* connector must be connected to one, and only to one, *liquidFlow_C* connector.
- Connection between causal connectors of different hierarchy:
  1. A *liquidFlow_C* connector of the molecular class must be connected to one, and only to one, *liquidFlow_C* connector of the submodels. It also can be connected to an arbitrary number (zero is allowed) of *liquidFlow_R* connectors of the submodels.
  2. A *liquidFlow_R* connector of the molecular class must be connected to at least one *liquidFlow_R* connector of the submodels. It cannot be connected to any *liquidFlow_C* connector of the submodels.

3.4. Fourth Step – Definition of the Interfaces

The fourth step of the design consists in defining the interface classes. Interface classes are gathered in specific *interface libraries*. The design methodology establishes how the interface classes must be.

Definition Interface classes must be only composed of (an arbitrary number of) causal connectors. Interface classes own all the attributes of its causal connectors.

Properties Three important properties arise of the interface classes definition:

1. It allows the library user to define new classes of interface adapted to his particular necessities.
2. Two models with the same interface can be exchanged in any context.
3. The connection rules define completely the way each class can be used.

Constraints should not be introduced among the causal connector variables at the interface definition. Otherwise, the applicability of the interface for the modeling of composed classes could be restricted unnecessarily. The next example tries to illustrate this point.

Example Consider a plane wall of thickness L, such that one of its surfaces is at temperature T_A and the other one is at temperature T_B. The heat flow through the wall in steady state can be modeled assuming that it is proportional to the temperature difference, $T_A - T_B = R_{th} \cdot Q_{A \to B}$. The proportionally factor, R_{th}, is the thermal

Fig. 2. Thermal resistance interface.

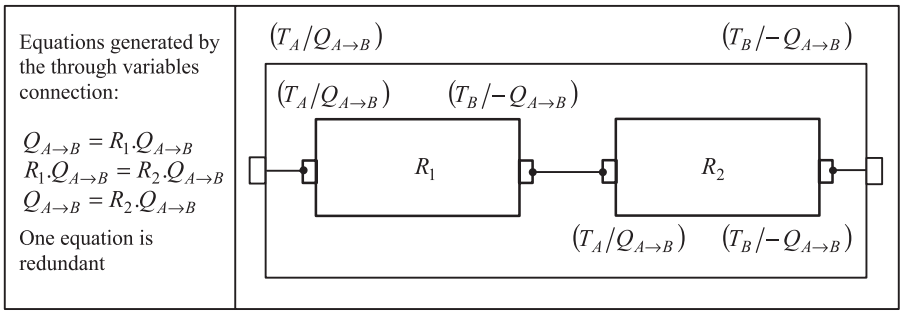

Fig. 3. Wall modeled as the connection of two thermal resistances.

resistance of the wall. A possible definition of the interface of the thermal resistance model is shown in Figure 2. It explicitly contains the condition that the heat flow entering by a connector is equal, at any time, to the heat flow exiting by the other connector. The connector variables are represented in the following way: (across variables/through variables).

In order to model in more detail the heat flow through the wall, this can be divided into N elements of the same thickness, L/N. In this way, the wall is represented by means of a connection in series of N thermal resistances. These N thermal resistances are grouped into a class that models the wall (the case $N = 2$ is represented in Fig. 3). However, if the interface of this molecular class and the interface of the thermal resistance class are the same (it is the most logical option), then there is a redundant equation, because the input and output heat flow is supposed to be the same in both. For this reason, it is advisable not to establish constraints among the interface variables in the interface definition. It is preferable to establish these constraints in the interface subclasses (i.e., in the internal description of the phenomena).

3.5. Fifth Step – Internal Description of the Phenomena

The fifth step of the design consists in programming the internal description of the phenomena. Phenomena classes inherit the interface classes. The interface causal

connectors impose a set of conditions that the phenomena modeling formulation must satisfy. The way of facing these conditionings depends on each particular case. However, some general techniques exist that could be adapted for solving the particular problems posed [34]. Some of them are discussed in the second part of this paper, ''A Case Study''.

3.6. Design Rules of the Libraries

Definition The *design rules* for the model libraries are composed by:

1. The definition of the causal connector classes.
2. The rules for connecting the causal connectors.

The purpose of the design rules is:

1. Facilitate the efficient modeling and simulation of large systems.
2. Establish the complete set of conditions that each library class must satisfy. This facilitates the test, debugging and maintenance of the library classes.
3. Define the way of using the classes. Define which classes can be exchanged.
4. Defining how the libraries can be extended with new classes.

3.7. The Role of the Library Designer

The design methodology defines the relationship between the library designers and users as follows. The only two pieces of information that the library designers have to provide to the library users are:

1. The modeling hypothesis which each class relies on.
2. The design rules of the libraries.

4. CONCLUSIONS

An object-oriented methodology for the design of reusable model libraries has been proposed. It takes into consideration all the phases of the library life cycle: design, programming, maintenance, modifications, extension, etc. The methodology completely defines the role of the library designers and their relationship with the library users. The library designers should guarantee the numerical efficiency of the models based on the library predefined-models. In addition, users should not be confronted with numerical problems. As a consequence, the numerical problems associated with the use of the classes should be identified and solved in the design phase. Some recommendations are given about the numerical aspects of the model libraries design. The key concept of the proposed methodology is the *libraries design rules*.

They constitute the only piece of information needed, from the numerical point of view, for using, modifying and enlarging the model libraries.

ACKNOWLEDGEMENTS

The authors wish to thank the referees of this paper for their constructive comments.

REFERENCES

1. Astrom, K.J., Elmqvist, H. and Mattsson, S.E.: Evolution of Continuous-Time Modeling and Simulation. In: *Proceedings of the 12th European Simulation Multiconference, ESM'98*, Manchester, UK, 1998. http://www.modelica.org
2. Steward, D.V.: *System Analysis and Management: Structure, Strategy and Design*. Petrocelli Books, Inc., 1981.
3. ABACUSS II: http://yoric.mit.edu/abacuss2/abacuss2.html
4. Piela, P.C.: ASCEND: *An Object-Oriented Environment for Modeling and Analysis*. Ph.D. Thesis, EDRC 02-09-89. Engineering Design Research Center, Carnegie Mellon University, Pittsburg, PA, USA, 1989.
5. Elmqvist, H.: *A Structured Model Language for Large Continuous Systems*. Ph.D. Thesis, TFRT-1015. Lund Institute of Technology, Sweden, 1978.
6. Elmqvist, H., Brück, D. and Otter, M.: *Dymola. Dynamic Modeling Laboratory. User's Manual*. Version 4.1b. Dynasim AB, Lund, Sweden, 2001. http://www.Dynasim.se
7. EA International and ESA: EcosimPro ver. 3.0. Five volumes: *Getting Started*, *User's Manual*, *Modeling Language (EL)*, *Modeling and Simulation Guide*, and *Mathematical Algorithms*. 1 December 1999.
8. Barton, P.I.: *The Modelling and Simulation of Combined Discrete/Continuous Processes*. Ph.D. Thesis. Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, London, May 1992.
9. MODEL.LA: http://www.mit.edu/afs/athena/org/m/modella/
10. Stephanopoulos, G., Henning, G. and Leone, H.: MODEL.LA. A Modeling Language for Process Engineering. Part I. The Formal Framework. Part II. Multi-Facetted Modeling of Processing Systems. *Comput. Chem. Eng*. 14 (1990), pp. 813–869.
11. Modelica, T.M.: *A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification. Version 2.0*. July 10, 2002. http://www.modelica.org
12. Andersson, M.: *Object-Oriented Modeling and Simulation of Hybrid Systems*. Ph.D. Thesis, ISRN LUTFD2/TFRT-1043-SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.
13. Cellier, F.E.: Integrated Continuous-System Modeling and Simulation Environments. In: D. Linkens (ed.), *CAD for Control Systems*. Marcel Dekker, New York, 1993, pp. 1–29.
14. Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H. and Kurstrom, A.: The Open Source Modelica Project. In: *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002. http://www.ida.liu.se/∼pelab/modelica/
15. Brenan, K.E., Campbell, S.L. and Petzold, L.R.: *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, PA, 1996.

16. Mattsson, S.E. and Söderlind, G.: A New Technique for Solving High-Index Differential Equations Using Dummy Derivatives. In: *Proceedings of the IEEE Symposium on Computer-Aided Control System Design*, California, USA, 1992.
17. Pantelides, C.C.: The Consistent Initialization of Differential-algebraic Systems. *SIAM J. Sci. Stat. Comput*. 9 (2), 1988, pp. 213–231.
18. Elmqvist, H., Cellier, F.E. and Otter, M.: Object-Oriented Modeling of Hybrid Systems. In: *Proceedings of the ESS'93, European Simulation Symposium*, Delft, The Netherlands, 1993.
19. Cellier, F.E.: *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. Ph.D. Dissertation, Diss. ETH 6483, 1979.
20. Cellier, F.E., Elmqvist, H., Otter, M. and Taylor, J.H.: Guidelines for Modeling and Simulation of Hybrid Systems. In: *Proceedings of the IFAC World Congress*, Sydney, Australia, 1993.
21. Elmqvist, H., Cellier, F.E. and Otter, M.: Object-Oriented Modeling of Power-Electronic Circuits Using Dymola. In: *Proceedings of the CISS – First Joint Conference of International Simulation Societies*, ETH, Zurich, Switzerland, 1994.
22. Elmqvist, H. and Otter, M.: Methods for Tearing Systems of Equations in Object-Oriented Modeling. In: *Proceedings of the ESM'94, European Simulation Multiconference*, Barcelona, Spain, 1994.
23. Elmqvist, H., Otter, M. and Cellier, F.E.: Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems, *Proceedings ESM'95*, European Simulation Multiconference, Prague, Czech Republic, 5–8 June, 1995, pp. xxiii–xxxiv.
24. Nilsson, B.: *Structured Modelling of Chemical Processes – An Object-Oriented Approach*. Department of Automatic Control, Lund Institute of Technology, 1989.
25. Nilsson, B.: *Object-Oriented Modeling of Chemical Processes*. Thesis. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1993.
26. Marquardt, W.: Dynamic Process Simulation – Recent Progress and Future Challenges. In: W.H. Ray and Y. Arkun (eds.): *Chemical Process Control*, CPC-IV. CACHE, 1991, pp. 131–180.
27. Cellier, F.E.: *Continuous System Modeling*. Springer, Berlin, 1991.
28. Elmqvist, H., Brück, D. and Otter, M.: *Additional Documentation for Dymola 3.1*. Dynasim AB, Lund, Sweden, 1998-08-11.
29. Mattsson, S.E., Elmqvist, H., Otter, M. and Olsson, H.: Initialization of Hybrid Differential-Algebraic Equations in Modelica 2. In: *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002. http://www.modelica.org
30. Mattsson, S.E., Olsson, H. and Elmqvist, H.: Dynamic Selection of States in Dymola. In: *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002. http://www.modelica.org
31. Otter, M. and Olsson, H.: New Features in Modelica 2.0. In: *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, 2002. http://www.modelica.org
32. Cellier, F., Otter, M. and Elmqvist, H.: Bond Graph Modeling of Variable Structure Systems, *Proc. ICBGM'95, 2nd Int. Conf. on Bond Graph Modeling Simulation*, Las Vegas, NV, pp. 49-55.
33. Elmqvist, H.: Object-Oriented Modeling and Automatic Formula Manipulation in Dymola. In: *Proceedings of the SIMS'93*. Scandinavian Simulation Society, Kongsberg, Norway, 1993.
34. Urquía, A.: *Modelado orientado a objetos y simulación de sistemas híbridos en el ámbito del control de procesos químicos*. Ph.D. Thesis. Departamento de Informática y Automática, Facultad de Ciencias, UNED, Spain, July 2000.
35. Elmqvist, H.: Personal Communication, 1995.
36. Karnopp, D.C., Margolis, D.L. and Rosenberg, R.C.: *System Dynamics: A Unified Approach*, 2nd edition. Wiley, 1990.