6

Design of Model Libraries

Abstract

This chapter gives some guidelines on structuring object-oriented model libraries. There are no unique solutions to that problem, but the idea is to capture key issues and proven solutions in a collection of *Design Patterns* which can be applied to other modeling problems. Design patterns are an attempt to describe "good practice" in a semi-formal way. Most of the design patterns are applicable to modeling in general, but a few are specific to Modelica or thermo-fluid systems. Examples using Modelica and the ThermoFluid library are used throughout the chapter to illustrate the ideas.

6.1 Introduction

Designing a user-extensible model library is different from building models for one time use. The extensibility is a feature which is not needed to the same degree in different engineering disciplines. When modeling electrical circuits, the models often consist of a large number of components, but each component is described by a few well defined mathematical models, all of which can be made available in an extensive component library. This means that a user typically composes system models from the library, but has no need to alter existing models or write new ones. The situation is different for thermodynamics and process engineering, particularly when the scope of the models is as broad as with the ThermoFluid library. The variety of heat- and mass-transfer equations and physical property data is broad by itself. The number of possible variants grows exponentially if different combinations of these are taken into account as well. Very often, modeling requires a problem-specific simplification which does not hold for other problems. Even a huge component library can not cover all variants that a modeler needs for routine modeling use. A library for thermodynamics has to be open for user defined extensions. Designing

a user-extensible library in such a way that it is powerful, flexible and easy to understand is a difficult challenge. Object-oriented decomposition of engineering system models into subsystem and component models follows the same decomposition as that of the system itself: the elements found on the blueprint or plant flow sheet should be library models. This decomposition has been discussed in [Nilsson, 1993] for process engineering and in [Mühlthaler, 2000] for thermal power plants. Much more code reuse can be achieved when the decomposition is continued to the level of physical phenomena. Library design for reuse at the level of phenomena is the topic of this chapter. Examples refer to thermo-fluid applications and cover the same models as ThermoFluid. The conceptual structure is emphasized instead of the details of the actual implementation[1].

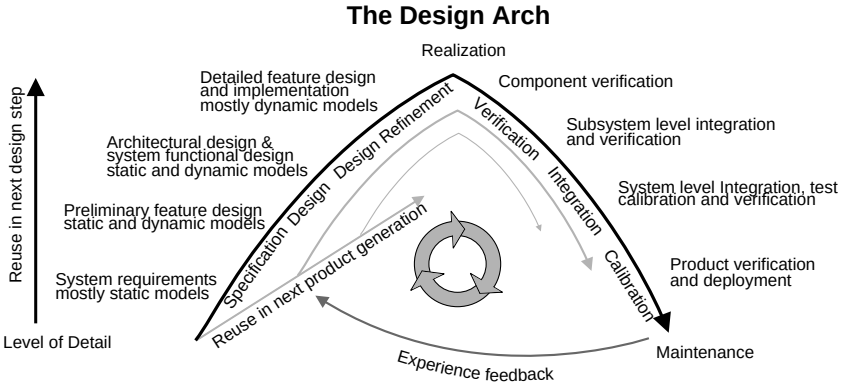

Figure 6.1 Model reuse along the design arch. The development phases are typical for a complex, highly developed product like a car. A similar scheme is sometimes referred to as design-V.

For an engineer, the foremost goal of a system model is to be mathematically correct and to fulfill the requirements in terms of accuracy and simulation speed. Due to the tedious work necessary to verify and validate models, this is a long term process. This model validation is called *calibration* in the automotive industry while the control community refers to the same process as grey-box parameter identification. Some parameters are always uncertain, so for every new system the user has to check again if the simulation gives a "good enough" picture of reality. Parameters have to be adapted to make simulation model output fit to measurement data. Many engineers focus on validating and calibrating their models and tend to neglect structuring and reuse issues. Proper code structuring

[1]A detailed and complete documentation is found at http://www.modelica.org/library.

has proved to increase programming productivity in software engineering. The cyclic nature of modeling in the iterative design of technical products from one generation to the next makes it obvious that reuse will speed up the modeling process substantially. Looking at Figure 6.1 reveals that there are two dimensions of reuse:

1. on the same level of detail for the next product generation and
2. along the path of the "Design Arch" in Figure 6.1, spanning different levels of detail during the same design cycle.

The second dimension of reuse is more difficult to achieve because the mathematical models and the time scales of interest are often different for another level of detail of a system model. The granularity of the system topology can vary along the path of the design process. It is common practice to neglect components with little influence on the system dynamics.

The main incentive for the development of model libraries is the cost savings from code reuse. The estimations for the cost of software development vary widely, in a recent report it was claimed that commercial software goes at a tariff of USD 50 – 200 per line of debugged code. For validated code in a modeling language, the numbers are probably higher. This makes it obvious that validated model libraries of industrial relevance are a valuable resource.

6.2 Means for Library Structuring

Code structuring has been discussed from a language perspective in Chapter 3 and with concrete examples from the ThermoFluid library in Chapter 5. Building on those examples, we will now illustrate how the language tools can be used for model libraries in general. Modeling always offers several ways to solve a problem, but in spite of the many possibilities, similar solutions for the mathematical parts and code structuring are found repeatedly even if the modeling languages are different. Building on power plant library modeling in SMILE and the broader scope of ThermoFluid in Modelica, experiences from designing object-oriented model libraries are summarized.

The advantages of using libraries is to reuse as much well-tested code as possible in models. This minimizes the need for extended testing. Validating and calibrating a model is usually the most time consuming part of the model development process. While there is no way to avoid the so called calibration which consists of adapting the model parameters to plant data via systematic or heuristic methods, testing and internal validation can be substantially reduced when well tested code is reused. For

simulation of standard problems and plants, it is possible to rely exclusively on tested model components. This greatly increases the trust in simulation results.

For the ThermoFluid library, two scenarios of reuse in model development have been considered:

- Use a partial component and complete the model by filling in the missing pieces.
- Start from scratch and build up a model using as many base classes as make sense.

Clearly, using partial components is faster but the partial models may not be available. Building models from base classes is more flexible, but takes longer and needs a thorough understanding of all used classes. When almost complete partial models are used, a developer only needs to know the interfaces and requirements of the missing parts. Readers unfamiliar with object-oriented techniques are recommended to take a look at the glossary in Appendix A in order to get an overview of the vocabulary.

Encapsulation

Information hiding is an important principle to improve the maintainability of programming code. The idea is that all interaction between models occurs via well-defined interfaces. If this principle is neglected, the interdependence between models is likely to increase. That in turn makes it more difficult to change the system model and both flexibility and maintainability are lost. Encapsulation of operations is also a property which makes it easier to debug faulty programs.

At first sight this may not seem like such an important issue. Experience with typical codes for engineering models in industry which evolved over many years shows that these codes tend to mutate to almost unmaintainable spaghetti-code. The main problem in maintainability is undocumented interdependence of different parts of the code, which is difficult to detect. This makes it impossible to find an evolutionary solution to adapt the code to new needs. Many companies depend on the functionality of the code, but when the last of the original developers leaves the company, a complete re-engineering has to be undertaken. Proper encapsulation techniques make it much more likely that an evolutionary solution simplifies the transition to new tools and methods.

Encapsulation in equation based modeling is fundamentally different from encapsulation in object-oriented programs, where interaction is mostly based on message passing between objects. All operations and the data they operate on are encapsulated in objects. In equation based modeling, all components of a system are linked together via a bipartite graph

that connects variables and equations of the differential algebraic equation system. This makes it impossible to speak of encapsulation of operations: an additional equation in one component can be compensated by adding a variable in another component, if the bipartite graph connecting variables and equations allows to match them. As an example, consider a system of a large network of incompressible, internal flow with three boundary conditions defining the interaction with the environment. Two types of boundary conditions can be given: either mass flows or pressures. One of the many possible configurations for boundary conditions is erroneous: when all boundary conditions are mass flows, the pressure inside the network is arbitrary, creating a so called "floating potential" problem.

Thus it is impossible to localize the error to a specific component: any one of the existing boundary conditions could be exchanged against a pressure boundary condition, or an additional pressure boundary condition could be added to remedy the non-physical situation. The equation system glues all components together in a way that the problem could be fixed by providing a pressure anywhere in the system[2].

This does not mean that attempts to encapsulate components or data are useless or impossible. Parameters can be encapsulated in models and access to them can be restricted. Modelica's main tool to achieve encapsulation is to use connectors to couple models. But the main strength of acausal modeling – that the direction of the information flow is not determined in the model, but is derived from the boundary conditions of a complete experiment – makes it difficult to debug models. Variables which are equated in a connection can be calculated in the models on either side of the connection. Therefore it can be useful to impose certain rules to make model debugging and system composition simpler. This has been done in the ThermoFluid library with flow models and control volumes. Flow models never calculate the fluid properties and always compute the flow variables in their connectors, similar rules hold for control volumes.

As in many other programming languages, readable and maintainable program code can not be enforced by the language. Coding style is an important element to achieve safe, maintainable code, as has been pointed out by [Summerville, 2000]. Local parameters and variables in a model can be declared as protected, which means that they can not be accessed by dot-notation from the outside and not be modified, see Chapter 3. This restriction makes debugging easier and prohibits misuse of dot-notation access. Following a design guideline consequently reduces the training time for new users. The Balance-models in ThermoFluid take care of all

[2]Assuming the model has the same number of equations and variables, it has to contain one additional equation for a mass flow as well. It is equally difficult to localize this additional equation.

interaction of a control volume with its environment. It is important that no other functionality is implemented there, this would make it more difficult to get an understanding of the role of each model class. Due to the acausal nature of equations it is impossible to enforce encapsulation of equations in partial components for a library developer who provides partial models. A complement that makes model usage easier is to postulate rules for partial library models and document them extensively.

Inheritance and Aggregation

Inheritance is one of the main tools for achieving reuse both in object-oriented software development and modeling. ▷*Inheritance*[3] ensures that code which is common to several models only appears at one place in the source code, meaning that it only needs to be documented once and maintained once in case of changes. But overuse of inheritance has a few drawbacks. Experience with the design of both software systems and model libraries has shown that the total number of classes in overly structured libraries becomes large. Understanding is difficult and a long learning time is the consequence. Large libraries can not be avoided for complex systems and a broad application scope, but often the large number of classes is caused by too extensive use of inheritance. If all variants of a class of models are derived via inheritance of a base class, many classes are needed. ▷*Aggregation*, on the other hand, makes it possible to add small units of functionality as needed. The difference in complexity becomes obvious when *combinations* of these different units are considered. This will be demonstrated with an example from the ThermoFluid library. The problem is to provide base classes for flow models with one inflow and one outflow e. g., distributed pipe models, but also lumped stirred tank reactors. We consider four optional phenomena which may or may not be required in the model:

- heat transfer interaction,
- dissipative work interaction from a stirrer,
- chemical reactions,
- diffusion through a membrane adjacent to the control volume.

Two alternative designs are considered, one which only uses inheritance and another option which uses both inheritance and aggregation. The class structure of both alternatives is illustrated in Figure 6.2 and Figure 6.3. The usage of the resulting library models is slightly different, so the figures do not cover the same ranges of model behavior.

[3] First occurrences of important terms defined in the glossary are marked with a triangle and typeset in ▷*slanted*.

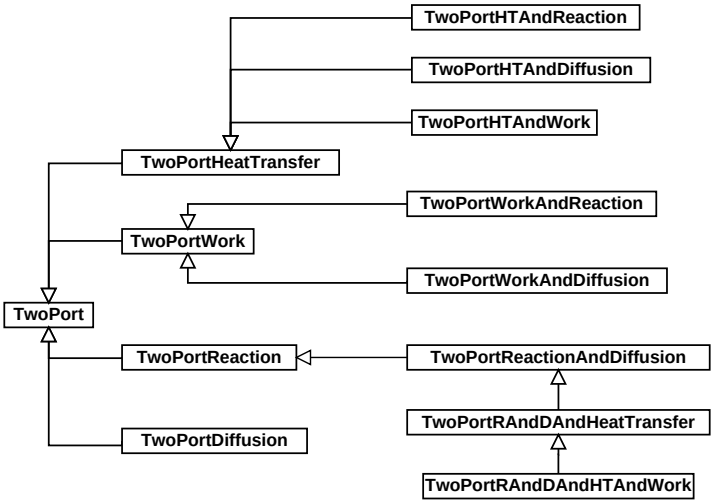

Figure 6.2 Design alternative for TwoPort bases classes using only inheritance. Not all possible combinations are included. The picture makes it obvious that inheritance does not lead to a simple library structure.

Figure 6.2 demonstrates what happens when only single inheritance is used to provide base classes of many model variants which in principle can be combined in an arbitrary fashion. In spite of the large number of classes in the graph, not all possible combinations are present, other combinations are included even if that particular combination is very unlikely to occur in practice. For example, a control volume with both dissipative work and membrane diffusion is very unusual in practice, but from a systematical point of view it should be part of the class structure. It should be kept in mind that none of the classes in Figure 6.2 implements any specific heat transfer or reaction mechanism, they just provide the interfaces.

Figure 6.3 shows the structure of the actual implementation in ThermoFluid, see Section 5.4 for more details. It uses a combination of single inheritance, aggregation and class parameters to achieve a structure which is both powerful and simple. Inheritance is used to specialize a general TwoPort to a TwoPort with heat- and mass transfer interaction. A TwoPort can be used as a base class for models with one inflow and one outflow, but also for more complex subsystems with one inflow and one outflow, e.g., a drum boiler composed of several simple models. TwoPortWithInteraction does not make sense as a base class for a subsystem, but it can be used for control volumes of different types. The instance of a HeatAndMassInteraction model contained in TwoPortWithInteraction

is replaceable, so that simple cases (only heat transfer) don't need to have parts which complicate matters, like reactions. Because membrane diffusion is the least common type of interaction, the diffusion-connector is not present in the default case. The example illuminates different features of

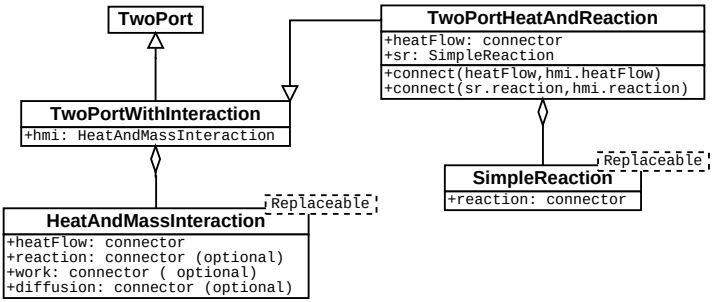

Figure 6.3 Design alternative for TwoPort bases classes using a combination of aggregation, inheritance and class parameters. The graphical notation is explained in Appendix A

.

code reuse using inheritance and aggregation and it also shows that the border between these cases is floating. Generalizing, it can be claimed that

- Inheritance fits very well the paradigm of starting with a very general model which is then specialized step by step.
- Aggregation is useful to cope with optional features which can occur in many combinations.
- Class parameters, discussed in more detail in Section 6.2 can help to keep the simple cases simple while keeping the option for more complex models.

The usage of multiple inheritance (MI) is haunted by the rumor that it adds more complexity than benefits. Multiple inheritance always adds complexity and some possible semantic pitfalls demand more coding discipline. There are however situations when a solution using multiple inheritance is simpler than other alternatives. As many tools for code structuring, multiple inheritance has to be used with care. There are also some fundamental differences between multiple inheritance in programming languages compared to an equation based modeling language. *Repeated* inheritance of the same base class via two inheritance paths, see Figure 6.4 is problematic in some object-oriented programming languages,

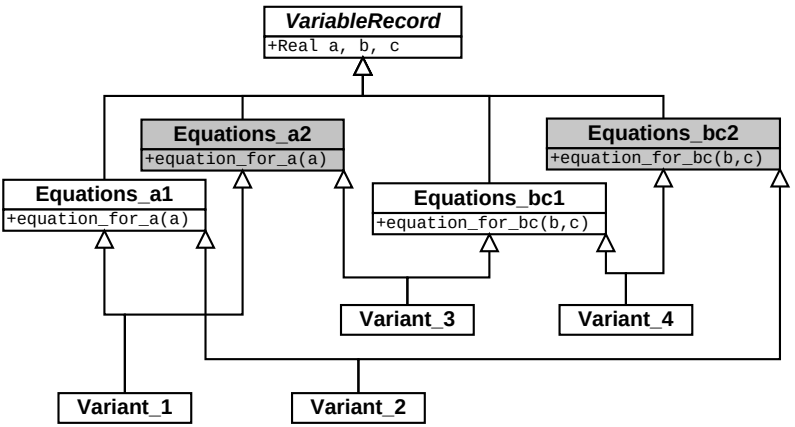

Figure 6.4 Repeated inheritance of the same base class VariableRecord by model variants one through four. Numerical and other modeling issues can be a reason for having different implementations of some equations. When several possible implementations exist and there are good reasons to combine them in a "mix and match" manner, multiple inheritance gives a compact solution.

but does not pose problems in Modelica. Two issues have to be kept in mind with MI in Modelica:

- When declarations with identical type and variable name are found in two base classes, these have to be identical in all components, including modifications. This is a consequence of merging repeated declarations into one without preferring any of them.
- Repeated inheritance works only for classes that do not have any equations (except the definition equations in modifications, which have to be identical and are merged into one equation), because including the same equation twice in a model is *always* a mistake.

Many of the problematic sides of multiple inheritance do not exist in Modelica due to different semantics, others are easy to avoid. In summary the reasons for using multiple inheritance in Modelica are:

- Ease of combination of ▷*polymorphic* implementations. For equation based modeling this means different equation implementations for the same set of variables. These might have different ranges of validity or numerical properties.
- For so called ▷*mixin classes*: behavior which is not always needed can be added by inheriting from one additional base class.

- Separate the graphical representation from the implementation. This can be used to customize graphical plant schematics, recreate the visual appearance of other simulation programs and similar goals without affecting the model behavior.

EXAMPLE 1—MULTIPLE INHERITANCE

As an example we compare the use of MI with other design options which fulfill the same requirements. Alternative designs will be discussed for a situation similar to Figure 6.4, but with more variants. It is assumed that three alternatives each exist for four equation parts which all operate on the same set of twenty variables, giving a total of twelve partial models. The parts implement different physical features. Some of the features are optional, some can be implemented in different ways. Each of the partial models implements a feature with a few equations using a subset of the common variables. For simplicity it is assumed that all possible combinations make sense from a modeling point of view, giving a total of $3^4 = 81$ possible combinations. The following design alternatives are considered:

Multiple inheritance. With multiple inheritance, twelve base classes are needed, giving an inheritance structure similar to Figure 6.4. The more common of the 81 cases can be provided as ready-to-use models, the others can easily be programmed when needed in a "some-assembly-required" fashion.

Single inheritance. Providing 81 classes using only single inheritance results in much redundant code and many classes, so this alternative can be ruled out.

Component aggregation with connectors. An alternative is to model the partial behavior in twelve components. If all information propagation between the components uses connectors, six connector types are needed and overlapping parts of the twenty variables have to be present in each component. In many cases this gives a lot of overhead which hinders readability. All interaction between components is made explicit with connections.

Component aggregation and modifications. Modifications are used in Modelica to propagate variables from a main model into its component models. Interaction between the container model and the components is achieved via using the propagated variables in equations. Compared to multiple inheritance, the modification code is additional overhead.

□

The actual design of a similar case in ThermoFluid are the control volume models which make use of a mix of all four structuring alternatives, taking advantage of their respective strengths and weaknesses. A few guidelines can be deduced from the experiences with ThermoFluid:

- It is practical to have optional parts as components because they can be added later on at any time.
- Multiple inheritance is advantageous for parts with a variety of implementations which can be mixed and matched in many combinations. This means also that multiple inheritance is only used to split the implementation of complex physical phenomena inside a single piece of equipment into more manageable parts, but not on the level of system composition.
- Mix-in behavior is a good case for multiple inheritance. In ThermoFluid, the initialization can be regarded as mix-in behavior and is added to the main model with multiple inheritance.
- System composition is always done by aggregation of engineering components using connections for the information exchange.
- Model parts which should be encapsulated can be put into a component. The component can be part of a model which is then used in multiple inheritance.
- Single inheritance and specialization of the child class should be used to finalize a partial model. A control volume class is complete, but some important high level parameters have to be specified for the final model: the type of fluid, the geometry, the heat transfer equation and similar details are defined in a child class using modifications.

Splitting up the implementation of the equations into different submodels does neither contradict nor enhance encapsulation, because the graph structure of the equation system is largely independent from the component structure anyway.

A known problem of multiple inheritance, name clashes and unintentional merging of variables with equal names, is easier to avoid in Modelica than in traditional programming languages. The Modelica typesystem combined with coding discipline make such errors unlikely: two variables typed as `SIunits.Pressure` and `Real` but both named p will cause an error. When both variables are of type `Real` this results in an unwanted merge of the definitions. When all physical variables make use of Modelica's fine-grained typing, such errors are very unlikely to occur.

From a structuring viewpoint, multiple inheritance is closer to aggregation than to single inheritance because it makes it possible to treat

parts of the model behavior as optional. The parts can then be assembled as needed. In object-oriented programming this use of multiple inheritance is called "mixin" class. A detailed example of the use of multiple inheritance and aggregation in ThermoFluid is found in Section 5.4. Similar structural designs can in principle be achieved with aggregation from components and multiple inheritance. The difference is the way the parts interact:

- When model parts are assembled using multiple inheritance, all interaction is implicit in the equations. Interaction is hidden in the bipartite graph that connects variables and equations. Some of the variables have to be present in more than one base class.
- For aggregation, there are three options of interaction:
  - equations in the container model that access variables in component models using dot-notation,
  - propagation of variables from the surrounding model to the components using modifications.
  - use of connectors and connections, either between components or from the surrounding model to a component.

  The last option is the most explicit way of interaction. Connectors result in a lot of overhead for small components with only one or two equations. Components with dot-notation can make equations difficult to read.

In Modelica multiple inheritance often increases the readability of the models because it results in compact code. As [Abelson *et al.*, 1985] put it: "programs must be written for people to read, and only incidentally for machines to execute". This holds equally for modeling languages. A disadvantage shared by both methods of aggregation and inheritance is that it can be difficult to get an overview over the complete set of equations that form the model. An editor that has the possibility to show the "flattened" code and merges all declarations and equations from base classes and components would overcome this drawback.

Class Parameters

Mathematical models evolve partially before and partially in parallel to building prototypes of the real system. This parallelism requires models which are flexible to quickly answer questions that come up during the design process. The most important feature to adapt models to changing needs is flexibility of the model development process. The responsibility

for achieving this flexibility is shared between the modeling tool, the modeling language and the libraries[4]. A Modelica feature that promotes flexibility is the concept of generic classes, usually called ▷*type parameters* or ▷*class parameters*. Using type parameters, models become *polymorphic*, meaning that they can represent different behavior depending on the value of the type parameter.

Type parameters are different from aggregation and inheritance because they do not only provide flexibility during the model development, but they also keep a model flexible all the way to the model user. A complete model ready for ▷*instantiation* can represent vastly differing behavior depending on the chosen type parameters. This illustrates the close connection between language issues and tool issues with respect to model flexibility: a type parameter can select a linear model instead of a nonlinear one, but a tool can equally well automate that process. From a user perspective it may not make a difference whether the linearized model is generated by the tool or built into the library.

For model library design, the first task is to identify the model parts or subsystems which should be kept exchangeable. In the ThermoFluid library, three types of submodels are kept as replaceable models:

- the fluid property calculation in the `Medium` type parameter,
- equations for heat transfer and
- friction pressure drop equations

Replaceable functions are a special case of generic classes. They are approximately equivalent to virtual methods in object-oriented programming languages. Replaceable functions are useful to keep the implementation of functional computations with given input-output relations exchangeable. A good example for a replaceable function is the computation of the isentropic change of enthalpy for turbines, valves, pumps and compressors. No matter in what equipment it is used, it always takes the inflow specific entropy and the outflow pressure as input arguments and it returns the corresponding specific enthalpy.

Type parameters are used in component modifications for propagating a type into hierarchical submodels in the same way as ordinary parameters are propagated. This is a very powerful feature that makes complete system models polymorphic. It is also the safest way to make sure that a type change is introduced consistently at all places where it has to be introduced. An example is a refrigeration system which can be used with different types of refrigerant, e. g., R134a and R22. A user can change the refrigerant type at the system level and the changes are propagated into all components and subcomponents, as illustrated in Figure 6.5.

[4] A more detailed look at modeling tools is outside the scope of this thesis.

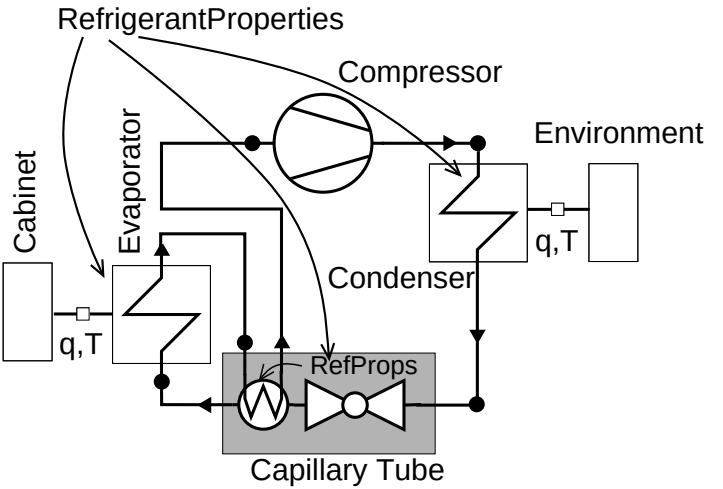

Figure 6.5 A refrigeration system is a prototype case where type parameter propagation makes the model very general. The type parameter for the refrigerant type is propagated down to all levels of the component hierarchy where a fluid property model is needed.

A requirement for building such systems is that the components or types which are going to be redeclared have been declared as replaceable to begin with.

6.3 Design Patterns for Modeling

Software design borrowed the notion of *design patterns* from architecture: there it has been in use for a long time to transfer knowledge and proven solutions to new generations of architects. Design patterns in mathematical modeling address recurring modeling situations by using a library design or modeling language idiom that helps to solve that modeling problem efficiently. The idea is to capture a structuring concept in a catch-phrase that is easy to remember. A design pattern should be sufficiently abstract to apply to many different situations, yet concrete enough to make its application to a particular problem situation obvious. Design patterns for software have been characterized into three categories: Creational Patterns, Structural Patterns and Behavioral Patterns. Mathematical models have rich dynamics, but the run-time code structure of dynamic models is completely static. Creational patterns are not yet implemented for this type of engineering modeling. Some simulation environments with a focus

on discrete event systems permit to create and destroy objects with continuous states during simulation runtime. Usually these are very simple models with few states which are integrated with their own instance of an explicit Euler or Runge-Kutta integrators, e. g., a car on a highway section.

Assuming that the scope and equations of the mathematical models in the library are clear, the task of library design is to divide the models into building blocks with well-defined interfaces, similar to Figure 6.6. Coding guidelines for structuring system models can be classified into two categories:

- **Structural Patterns** for code reuse. These can be classified as physical patterns that abstract physical behavior and topology patterns reflecting system structure.
- **Numerical Patterns** that make sure that solution methods can deal with the models as effectively as possible.

The possibilities for design patterns are closely tied to the features of the modeling language. The existence of equations as independent entities in the language can be seen as a pattern for modeling. The `flow`-prefix in Modelica is a kind of design pattern, derived from a generalization of Kirchhoff's law for electrical circuits to all modeling domains where flows of force, torque, mass etc. follow the same semantics. Consequently, some of the following patterns are specific to Modelica, but others are completely independent of the modeling language and apply equally to FORTRAN subroutines used in a legacy simulator. This holds mostly for the numerical patterns.

6.4 Structural Design Patterns

In mathematical modeling of systems there are two structures that design patterns can refer to: the inheritance based class structure and the mathematical structure of the equation system. The class structure is responsible for achieving code reuse and the mathematical structure is important for computational performance. Most people with experience in mathematical modeling do not at the same time have a background in software engineering. The software design motivated design patterns should therefore be suitable for non-programmers and straightforward to use. The simplicity for the model user is not so much a question of the complexity of the underlying implementation but of how well the simulation tool wraps the concept into an intuitive user interface. Some of the following simple patterns are well known since years and used by many

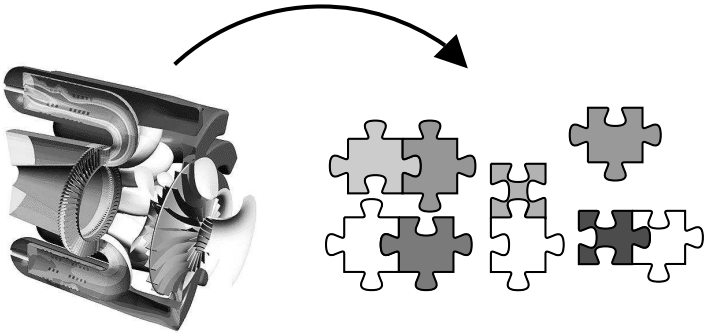

Figure 6.6 Design patterns: Finding abstractions in a class of technical products which are useful jigsaw pieces in many contexts.

modelers, but usually patterns which are well known in one engineering domain are ignored in other domains where they are equally useful.

Unfortunately, many of the design patterns depend both on the expressiveness of the modeling language and the capabilities of the modeling tool. The following design pattern assume Modelica 2.0 as the modeling language and Dymola as the simulation tool. The patterns are extracted from the experiences of developing the ThermoFluid library and not meant to be complete for other engineering domains. Especially the numerical patterns represent the most common pitfalls for non-experts in simulation. Our experience is that 80 % of the questions and problems arising from the use of ThermoFluid would have been avoided if all users understood these numerical pitfalls. The remaining 20 of support requests were caused by "chattering" of discrete modes, an as of yet unsolved problem of combined continuous and discrete simulation, see Section 2.1.

Physical Patterns

Many attempts have been made to make modeling a more systematic activity. All of these attempts emphasize the importance of identifying the *driving forces* or potentials and *flows*. If models are split into submodels and the connections between these submodels are abstracted to have zero volume, then the driving forces on both sides of the connection are equal and the flows are equal in magnitude but opposite in sign. This *flow semantics* is found in all areas of physical modeling, they have among others been used in Bond Graphs and the many extensions to Bond Graphs that try to extend Bond Graphs beyond energy flow, see e. g., [Cellier, 1991] and [Gawthrop and Smith, 1995]. A recent attempt to develop systematic rules for physical modeling which can be seen as physical design patterns

is elaborated in [Weiss and Preisig, 2000].

Flow semantics are not common in other thermo-hydraulic simulation tools, but using flow semantics makes a significant difference for model reuse. For the ThermoFluid library it simply means that a 1-to-5 flow splitter can be realized by attaching 5 flow models to a control volume model. No extra model is needed and due to the flow semantics the mass and energy balances are fulfilled.

DESIGN PATTERN 6.1—FLOWCONNECTOR
Use Modelica flow semantics for transport of conserved quantities for all physical connectors between subsystems. △

EXAMPLE 2—FLOW SEMANTICS
The ThermoFluid library uses flow semantics for three flow types: vector of component masses, flow of enthalpy and flow of momentum. The usage of flow semantics makes it in most cases unnecessary to have models for flow junctions. Splitting a flow into many smaller ones is simply done by using a one-to-many connection. □

Using flow semantics for mass and energy flows avoids errors and reduces the number of classes needed. It also works for momentum flows for simple cases, but due to the simplification of the three-dimensional momentum vector to a scalar, connections with an angle different from 180°C have to be modeled with a detailed model instead of just connecting the flow channels.

Topology Patterns

DESIGN PATTERN 6.2—CONNECTORSET
Provide models with typical connector configurations as base classes. Implement the physics inside them in derived classes. △

This design pattern is very basic and has been used in all engineering Modelica libraries, for example:

- `TwoPin` is a base class for electrical models such as resistors, capacitances, diodes and many other models,
- `TwoFlanges` are base classes to all rotational, one dimensional mechanical models with two flanges,
- `TwoPorts` are the base classes in ThermoFluid with two flow connectors like pipes, valves or pumps.

Similar base classes exist also for other libraries. These base classes are reused using single or multiple inheritance in the base classes.

DESIGN PATTERN 6.3—TYPERECORD
Collect the set of variables and record-components that define type compatibility in a record class. Classes which belong to this type compatible set shall inherit from this class. △

TypeRecord is a simple means to ensure *type compatibility* among a group of models where a basic, simple model is designed to be replaceable by a more complex one if needed. The TypeRecord is used as the *constraining class* in the declaration of the replaceable component or class.

EXAMPLE 3—TYPE RECORDS
Many classes in the package CommonRecords are designed as TypeRecords: collections of variables that characterize a group of models. The class StateVariables_ph defines type compatibility for all fluid property models using pressure p and specific enthalpy h as inputs to the medium property calculations. The TypeRecord makes it easy for other developers to write fluid property models which can replace an existing property model in ThermoFluid without any adapters or interface code. TypeRecord is a fundamental pattern for polymorphic model implementations in Modelica. □

DESIGN PATTERN 6.4—CONSISTENCYMODEL
Collect sets of data, functions and equations that have to stay together for consistency reasons in one replaceable model. △

This pattern is similar to class design in object-oriented programming. In mathematical modeling there are also cases where several functions operate on the same set of data. If these functions should be replaceable, they should by designed in such a way that it is not possible to replace parts that render the whole model inconsistent.

EXAMPLE 4—CONSISTENT REDECLARATION
High accuracy property functions consist of a large number of parameters describing a non-linear thermodynamic surface. Many functions make use of this data: if e. g., a function for the speed of sound and one for the specific heat capacity are needed within the same model, the unit of redeclaration should comprise both the functions and the parameters. □

DESIGN PATTERN 6.5—PARAMETERLIFTING
This pattern ensures that consistency constraints between parameters are enforced when propagating parameters into submodels. △

Geometrical parameters at the interface between two components obviously have to be consistent in both models. This can be achieved with

parameters in connectors – an assertion is generated to make sure that both parameters are identical on both sides of the connect. However, this would lead to a large number of connectors. A better solution is to make the container model responsible for consistency of the parameters. This is best demonstrated with an example.

EXAMPLE 5—PARAMETER PROPAGATION
A heat exchanger in ThermoFluid is composed of a discretized control volume on the hot side, one on the cold side and a solid wall separating them. We assume a tube-and-shell configuration with a cylindrical shell and ten straight tubes. For simplicity, the heat capacity of the outer cylinder is neglected. The given data is:

component	dimension	symbol
tube bundle	number of tubes	N
tube bundle	inner diameter	d_i
tube bundle	outer diameter	d_o
tube bundle	length	l
tube bundle	density	ρ
tube bundle	specific heat capacity	c_p
shell cylinder	length	$L = l$
shell cylinder	inner diameter	D_i

The heat capacity of the simplified single tube wall is computed as

$$C = N\ l\ 0.25\pi(d_o^2 - d_i^2)\rho\ c_p.$$

The volume of the shell side of the heat exchanger is computed as

$$V_{shell} = L(\pi D_i^2 - N\ \pi d_o^2)/4$$

similar expressions hold for other parameters. This re-parameterization has to make sure that:

- All low-level parameters are assigned in modifications to avoid possible sources of errors when setting such values by hand.
- Parameters shared by several components are consistent.
- The top level parameters that a user has to specify are easily accessible from component blueprints.

This may seem to be a trivial issue. However, it is very easy to overlook such parameter dependencies and it is a common source of errors in component based modeling. □

6.5 Numerical Design Patterns

Numerical design patterns address typical numerical pitfalls when textbook equations are used in modeling code. There are many books on numerical mathematics, e.g., [Press *et al.*, 1986] and [Hairer and Wanner, 1996], but they usually do not address the numerical problems in system simulation. The task of a model library designer is to identify typical pitfalls in a domain and provide library models that avoid them. For non-obvious pitfalls the library designer should go as far as to discourage the user to run into them when building models.

In order to be numerically robust, library models should not contain code that may cause problems. In practice, this can not be achieved completely, because many problems are not manifested before a complete model is assembled. Many of the following problems could automatically be detected by tools and warnings could be issued

DESIGN PATTERN 6.6—SINGULARITYCHECK
Make sure that functions with singular points or singular derivatives which are non-physical due to simplifications are regularized properly.
△

Many empirical models return physically meaningless values or have singularities outside their region of validity. For simple system models it is often not justified to model the regions in detail, but it increases the robustness and usefulness of the model if the results are qualitatively correct and the numerical singularities are taken care of. This is sometimes a hack for physical models, but often an unavoidable compromise to keep system models simple. Physical correctness outside the region of validity of the model is sacrificed as long as the qualitative behavior is still correct and larger robustness of the model is obtained.

EXAMPLE 6—SQUARE ROOT FUNCTIONS
A notorious problem in modeling of flow resistances is the use of empirical or semi-empirical flow resistance formulas involving the square root function. An example is a formula derived for turbulent flow with high Reynolds numbers, for which the behavior is often extrapolated to low flow speeds when the exact behavior at low speeds is not irrelevant. They trigger a standard problem when Newton-Raphson algorithms are used for solving non-linear equations. In its simplest form, pressure loss formulas can be written in the following form:

$$f(\Delta p) = \dot{m} - k\ sign(\Delta p)\sqrt{\rho\ abs(\Delta p)} = 0 \quad \Delta p = p_1 - p_2$$

Assume that the Δp has to be calculated from the above equation, e.g., in a zero-volume T-junction, see Section 5.8. Successive approximations

to the solution of $f(\Delta p)$ are obtained from the following iteration:

$$\Delta p^{j+1} = \Delta p^j + \frac{f(\Delta p^j)}{\frac{\partial f(\Delta p^j)}{\partial \Delta p^j}} \approx \Delta p^j + \frac{f(\Delta p^j)}{\frac{\Delta f(\Delta p^j)}{\Delta(\Delta p^j)}}$$

The step sizes of the Newton method depend on the approximated or analytically computed derivative of $f(\Delta p)$ with respect to Δp. For Δp close to zero the derivative goes to infinity and the step size goes to zero. This means that the iteration progresses very slowly. This phenomenon is sometimes called *inflection* because it occurs at inflection points of a curve with an infinite derivative at that point.

The singularity of the pressure drop function near the origin has no physical significance. Therefore it is perfectly reasonable to replace the singular formula with an approximation that does not cause numerical problems. The approximation should of course be correct qualitatively and it should not influence the system behavior more than necessary. In the ThermoFluid library, third order polynomials are used in a neighborhood around zero flow. The polynomial coefficients are chosen such that the overall function is continuous with continuous derivatives. □

EXAMPLE 7—THE LOG-MEAN TEMPERATURE

Another example where a careful implementation is needed is the *log-mean temperature difference*, ΔT_{lm} which has a statically correct behavior for heat transfer in heat exchangers. This means that heat transfer is calculated based on:

$$\Delta T_{lm} = \frac{\Delta T_1 - \Delta T_2}{ln(\Delta T_1 / \Delta T_2)}$$

where ΔT_1 is the temperature difference at one end of the heat exchanger and ΔT_2 is the temperature difference at the other end of the heat exchanger. Dynamically and under start-up conditions, the temperature gradients can be reversed for short times. It does not make sense to use the log-mean temperature difference when the signs of ΔT_1 and ΔT_2 are different. A numerically robust implementation has to take care of this case, too. Singularities or numerical ill-conditioning occur when:

- $\Delta T_1 \approx \Delta T_2$ and
- either $\Delta T_1 \approx 0$ or $\Delta T_2 \approx 0$.

The singularities near zero can be treated in the same way as the flow singularity above, the case of $\Delta T_1 \approx \Delta T_2$ can according to [Mattsson, 1997] be treated as follows. When $|\Delta T_1 - \Delta T_2| < max(|\Delta T_1|, |\Delta T_2|)$ it is better

to use the Taylor expansion

$$\Delta T_{lm} = 0.5(\Delta T_1 + \Delta T_2) \times \left(1 - \frac{1}{12}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\left[1 - \frac{1}{2}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\right]\right)$$

□

Scaling and normalization are important techniques which traditionally are used to improve numerical calculations. It is often advantageous to use dimension free variables and parameters. Many design patterns for model derivation are based on scaling and normalization. Many books on modeling elaborate on scaling and dimension-free quantities as fundamental modeling techniques, see e. g., [Lin and Segel, 1988]. Different engineering domains have different traditions regarding these techniques: in some domains it is common to always normalize models, in others unscaled values are used. Normalization is not common in thermo-fluid systems and process engineering and therefore ThermoFluid uses non-normalized quantities.

Some types of numerically motivated scaling have to be done by the model developer, for example the following:

DESIGN PATTERN 6.7—SCALING
Scale extremely nonlinear functions to improve numerics. △

EXAMPLE 8—SCALING OF EXPONENTIAL FUNCTIONS
Chemical equilibrium reactions often contain exponentials of temperature functions as the equilibrium constant. The dissociation of hydrogen at high temperatures, $\frac{1}{2}\ H_2 \leftrightarrow H$, can be described by the following equations:

$$k = 2.6727 - \frac{11.247}{T} - 0.0743\ T + 0.43170\ log(T) + 0.002407\ T^2$$

$$x_H = \frac{\sqrt{x_{H_2}}}{\sqrt{p}}\ e^k$$

where x_H is the mole fraction of atomic hydrogen, x_{H_2} is the mole fraction of molecular hydrogen, p is the pressure in atmospheres and T is the temperature in Kelvin. At low temperatures, the mole fraction of atomic hydrogen is extremely small. The second equation is scaled by taking logarithms. This can be achieved by introducing scaled variables, e. g., $logx_H = log(x_H)$. At 1 atmosphere and 280 K, the ratio of the left- and right hand side of the equations is 1.3×10^{75} in the non-scaled variables. This ratio reduces to 172 when logarithmic scales are used. □

Scaling of variables and equations can also be done by the tool, but the tool needs to have information about the ranges and the nominal values of the variables.

DESIGN PATTERN 6.8—VARIABLERANGES
Set tight minimum and maximum and accurate nominal values for all physical variables. △

Accurate minimum, maximum and nominal values can help numerical routines to find the solution and improve the numerical conditioning. Making sure that ranges are set as accurately as possible is thus a part of careful modeling. Many models have mathematically correct solutions which are physically meaningless. Equations for chemical equilibrium always permit solutions with negative concentrations which do not make sense physically. Limiting the search for solutions of nonlinear solvers to the physically meaningful ranges reduces the number of failures and the need for users to provide good initial guess values.

Nominal values are for example important to determine how perturbations should be chosen for numerical linearization of a model.

DESIGN PATTERN 6.9—SMOOTHING
Piece-wise and discontinuous function approximations which should be continuous for physical reasons shall be smoothened. △

In physical modeling it is common to have empirical or semi-empirical formulae that approximate measured data. Non-dimensional numbers are used to describe a given problem with a few parameters. In fluid flow there are many relations for turbulent or laminar flow with considerable uncertainty in the transition.

EXAMPLE 9—HEAT TRANSFER EQUATIONS
Convective heat transfer with outer flow perpendicular to a cylinder is characterized by the following empirical equations:

$$Nu_{lam} = 0.664 Re^{1/2} Pr^{1/3}$$
$$Nu_{turb} = \frac{0.037 Re^{0.8} Pr}{1 + 2.443 Re^{-0.1}(Pr^{2/3} - 1)}$$

These formulas are combined as

$$Nu = 0.3 + \sqrt{Nu_{lam}^2 + Nu_{turb}^2}$$
$$10 < Re < 10^7, \quad 0.6 < Pr < 1000.0$$

with an offset for low Reynolds numbers. The weighted mean is continuous and continously differentiable except at the origin. If this formula is to be used for plant startup, $Re = 0$, design pattern 6.6 SingularityCheck

should be used to make the formula robust at zero flow speed. Note that in this case smoothing acts like a weighted summation because the total Nusselt number is always larger than any of the parts. □

The following two design patterns deal with the *stiffness* of the resulting equation system, see Chapter 2. Stiffness is often the result of composing a system from subsystems. Therefore it is not possible to avoid all stiffness problems with library models. Choosing a solver that can deal with stiff equations may not necessarily be the best solution. Making the equation non-stiff speeds up the solution considerably and broadens the range of applicable solvers.

DESIGN PATTERN 6.10—TIMECONSTANTSELECTION
Make a problem non-stiff by using a steady-state assumption (singular perturbation technique) when the fast dynamics of the system are not of interest. Make a problem non-stiff by approximating very slow dynamics with constants. △

Typical examples of this technique are given in Chapter 4, the assumption of chemical equilibrium for fast reactions in Section 4.9 and using the quasi steady-state assumption for the momentum balance of fluids in Section 4.3. The assumption of taking the volume of a control volume to the limit of zero, used in Section 5.8 for T-junctions, is based on the same considerations. This example demonstrates that the decision to use a control volume model with or without dynamics has to be done on the *system* level by the model user. A related technique is to replace very slow dynamics like fouling with a constant. Modelers should be aware of the fact that it only makes sense to look at a limited range of system time constants at a time. This is reflected in the name to this design pattern.

DESIGN PATTERN 6.11—EASINGSTIFFNESS
Render a problem less stiff by making the time constants of the fast dynamics slower. △

This design pattern has been used for modeling of turbines in power plants, see [Thumm, 1989]. This particular way of dealing with stiffness is better than removing the fast states is explained in Section 5.8. The disadvantage of a singular value perturbation in the steam turbine example is a non-linear equation system involving all tap-off mass flows and pressures in the turbine. This is a purely numerical motivation for knowingly altering the dynamics of the real system. Under certain circumstances, this is a reasonable solution:

- Removing the fast states by a singular perturbation leads to numerical difficulties, typically non-linear equations which can be difficult to solve, especially at initialization time.

- The dynamics after changing the time constants are still much faster than the dynamics of interest.
- The change of the time constants makes the equation system solvable by solvers for stiff differential equations.

This technique can also be used to decrease model stiffness in order to use explicit solvers for hardware-in-the-loop simulation. In the case of the steam turbine discussed in Section 4.7, the ratio of the largest to smallest eigenvalue was changed from 10^6 to 100 without a noticeable influence on the dynamics of interest. From a control perspective this means that the model is rendered numerically tractable by changing it outside the frequency range of interest.

The design patterns EasingStiffness and TimeConstantSelection are addressing the same problem but suggesting different solutions. This shows that experience is needed to choose the best solution. In many cases both of the above methods will work well with similar performance, in other cases one of the methods is clearly superior to the other.

DESIGN PATTERN 6.12—FULLYSYMBOLICCODE
Make sure that symbolic derivatives are available for all model parts, including external functions. This improves the solution of non-linear equation systems and enables automatic index reduction. △

This design pattern is specific to tools which support symbolic manipulation of the model code like automatic index reduction and automatic differentiation, but require that all model equations and functions are differentiable. This is the case in Dymola, MathModelica, ABACUSS II and to some extent gPROMS. High index problems, compare Section 2.1, arise when algebraic equations constrain states to a manifold defined by an algebraic equation. Object-oriented model composition goes hand-in-hand with the need for automatic index reduction, because the constraint equations are the equations generated from a connect statement. An algorithmic way to reformulate the model to an equivalent problem with index one is based on the symbolic differentiation of the algebraic equation. Due to the complexity of object-oriented modeling, a large part of the algebraic equations or functions of a model can become a constraint. Consequently, modelers should provide derivatives for all model functions. In Modelica, this is done with the help of derivative annotations. Compared to using equations, this is additional work for the modeler. However, it can not be avoided for external functions or when functions have other significant advantages over equations.

DESIGN PATTERN 6.13—DISCONTINUITIES
Avoid discontinuities in user defined functions whenever possible. △

A Modelica compiler can easily detect discontinuities in equations, but this is not the case with functions. A function returning discontinuous outputs for continuous inputs will cause serious trouble for numerical integrators. Instead of a function with one discontinuity, two functions should be provided. An event is generated automatically when the functions are called in the branches of an if-expression.

It may be impossible to avoid discontinuities when reusing external functions. The next design pattern applies to external functions with discontinuities or discontinuous derivatives.

DESIGN PATTERN 6.14—EVENTDETECTION
Provide explicit crossing functions for non-smooth external functions. △

The Modelica language and Modelica implementations handle the numerical requirements of hybrid models automatically. Crossing functions are needed by numerical integrators for models with discontinuities in the right-hand side of the differential equations, as outlined in Section 2.1. The automatic generation of the crossing functions does not work for external functions, where the discontinuities are hidden from the Modelica translator. The only way to obtain a numerically robust treatment of such cases is to add a crossing function to the model with the discontinuous external function. The function has the following properties:

- The function needs to have one output which changes its sign at the same location as the discontinuity of the original function. This usually means that it has the same inputs as the original function.
- The crossing function has to be used in such a way in the Modelica code that it triggers an event, see the following listing.

```
model ExternalCrossing "a model using a discontinuous external function"
  ... // other model parts omitted
  function externalCF "Modelica declaration of external function"
    input Real a,b,c "sample input";
    output Real zero_xing "value changes sign at discontinuity";
    external "C" myCCode(a,b,c,zero_xing); // name of the external function
  end externalCF
  Real zero_xing "value changes sign at discontinuity";
  Boolean externalEvent(start=true) "a boolean variable";
equation
  zero_xing = externalCF(a,b,c);
  // the next line causes an event when zero_xing changes its sign.
  externalEvent = if zero_xing > 0 then true else false;
  ... // further equations and functions omitted
end ExternalCrossing;
```

Listing 6.1 Usage of an external crossing function.

This type of problem is typical for interfacing Modelica to traditional C– or FORTRAN codes. A non-trivial example which required to develop the external crossing function and the Modelica interface for multi-phase property calculations is described in [Tummescheit and Eborn, 2002]. Crossing functions for external functions could be avoided with similar techniques as derivatives for external functions. Automatic differentiation techniques, described in more detail in Chapter 7, can not only analyze code to compute the derivatives of that code, they can also be used to detect discontinuities. This has been demonstrated in [Tolsma and Barton, 2002].

6.6 Conclusions

This chapter discussed two issues in development of object-oriented model libraries which are independent of the application domain: code structuring for reuse and numerics. Both issues are important to obtain a flexible and robust library. The numerical design patterns and examples in this chapter are proposals for avoiding difficulties but they do not cover all problems that users of ThermoFluid have experienced. The structural design patterns are also only a few patterns for structuring the model code, but they summarize patterns that were successfully used in ThermoFluid. Experiences from model libraries in other domains indicate that there are many similarities between different domains.

Object-oriented modeling or software techniques are not like a silver bullet that ensures well structured, well documented and flexible code. Discipline in documenting code and adequate use of the object-oriented features are necessary in order to take full advantage of the benefits of object orientation.