The Architecture of the Smile/M Simulation Environment

Thilo Ernst, Stefan Jähnichen, Matthias Klose

GMD FIRST Research Inst. for Computer Architecture and Software Technology
Rudower Chaussee 5
D-12489 Berlin, Germany

Technical University of Berlin
Software Engineering Research Group
Franklinstr. 28/29
D-10587 Berlin, Germany

Thilo.Ernst@gmd.de, {jaehn,doko}@cs.tu-berlin.de

Abstract

Smile is an object-oriented, equation-based, hybrid modeling and simulation environment. The focus of this paper is the extension (Smile/M) of the Smile system to Modelica, a new modeling language currently being defined in an international collaboration effort. An overview of this language, which may well become a quasi-standard in the simulation community, is given. Interfacing Smile to Modelica will enhance the interoperability of the system and allow the reuse of models developed in other environments.

1 Introduction

With the advent of highly productive and user-friendly simulation environments, mathematical modeling and simulation becomes more and more an everyday tool in engineering [2] [4] [7] [10]. A considerable number of such environments is available both from industrial vendors and from research organizations. However, many of them originated from specialized application domains, only very few were designed for general applicability. Recent results and trends in computer science were not always taken into account, so openness and other architectural considerations were not usually given appropriate priority.

As a consequence, the majority of available systems lacks satisfying interoperability: Usually, an "import" of models expressed in formalisms different from the native modeling language is impossible or subject to severe restrictions; the integration of simulation modules implemented with other tools (or standard programming languages) is difficult, if at all possible. These difficulties become more and more pressing as the trend in engineering work shifts towards problems requiring multidisciplinary cooperation, and thus also the combination of domain-specific simulation components. Relief could be provided by a common modeling formalism — an exchange medium between the various simulation environments, each of which would then only have to be extended with appropriate import/export features. However, up to now no such common formalism has emerged.

Modelica is an ongoing international effort aimed at designing such a uniform modeling language; to our knowledge there is only one other standardization effort: VHDL-AMS [8] (earlier called VHDL-A [3]), which however lacks full object orientation and suitability for non-causal modeling. Within the Esprit project *Simulation in Europe Basic Research Working Group (SiE-WG)*, in October 1996 an effort was started to design a new, unified language for physical systems modeling. Beyond the central goal of creating a common platform to ease exchange of models and model libraries, this effort is intended to unify and generalize on innovative concepts from several modeling languages.

This paper describes the Smile simulation environment, gives an overview of Modelica and shows how Smile is being adopted to process Modelica.

2 The Smile System

Smile [5] [12] is a simulation environment based on a combination of innovative concepts: Object-oriented and equation-based modeling, separation of model and experiment description, and an open and extensible system architecture. Smile was developed at TU Berlin and GMD FIRST and has an active user community whose main emphasis is currently on energy systems (see e.g. [6]).

The Smile system consists of the Smile *modeling language* to express the mathematical model of a

physical system in, the *experiment description* language to configure an experiment, a set of numerical solvers (discussed in [11]), an interactive runtime support and a developer tool for browsing and exploring the object-oriented model libraries. A characteristic of the Smile system is the strict separation of model and experiment. This separation is reflected in the Smile system by the provision of two distinct languages — the model description language and the experiment description language.

The Modeling Language. The Modeling language Smile was designed as an extension of Objective C, which itself is an object-oriented general-purpose language based on C and Smalltalk. By adding means to express different types of equations, to define a connection interface, and to specify connections, Smile becomes an equation-based object-oriented simulation language.

The modeling language combines advantages of two concepts, and thus provides the user with means to conveniently model physical systems: On the one hand, the characteristics and behaviour of a real-world object can be described in the form of differential and algebraic (discrete and continuous) equations, and on the other hand, the object-orientation of the language means that it also provides the user with the means to structure the system in a natural way. The description of the characteristics and behaviour of a real-world object is encapsulated in a class definition. Further structuring is possible by allowing object-valued variables as components of a class. This part-of relation allows the system to be decomposed into sub- and subsubsystems. Thus, large and complex systems can be structured and modeled in a hierarchical manner. In addition to the well-known advantages of object-orientation, such as the reusability, extensibility and easy specialization of models, this also yields a further advantage, which is especially useful to the simulation engineer: subsystems can be validated independently from the compound model they will be part of.

```
@interface HeatStore : Model {
  @export
    double T  [eq, doc: "Temperature", unit: "K"];
  @intern
    double m  [doc: "Mass", unit: "kg"];
    double cp [doc: "Specific heat capacity",
               unit: "kj/kg/K"];
    double dQ [doc: "Chng of Energy", unit: "kJ/s"];
} @end
```

```
@implementation HeatStore

@eq diff T
{
    return dQ / (m * cp);
}

@end
```

In the example above the definition of the `HeatStore` is divided into two parts. In the interface, the model variables are declared with a visibility specification: exported variables become part of the connection interface. Attributes related to variables specify, for example, a short textual description or its unit. In the implementation part relationships among variables can be expressed with equations. Specialized or extended models can be derived from base models by using the inheritance relation. The `Plate` model used in the example below is derived by using this relation. The `Stove` model shown below is built-up of instances of the `Plate` model (part-of relation). The components of a model are initialized in the context of the `Stove` model by the `@component` construct (arrays of components are conveniently initialized by a single construct). Connections between variables belonging to the interface part of models are connected by the `@connect` construct.

```
@interface Stove : Model {
  @protected
     Plate *myplate[4];
     double Power [unit:"kJ/s",
         doc:"Power supply of stove"];
  @export
     double OutP[4] [eq, unit:"kJ/s",
         doc:"Power->Plate"];
     ...
} @end
```

```
@implementation Stove
  @component myplate[i] {
    // initialization of myplate[i]
    // for the context of the stove
  }
  @connect {
    myplate[0..3].InP = OutP[0..3];
  }
  @eq discrete OutP[i:0::3]  { ... }
@end
```

The Runtime Environment. The modeling language supports the user in the modeling phase of the simulation process. The next step in the process is the actual description and execution of an experiment. This is done by an *experiment description* which specifies a particular Smile model that must be simulated and instantiates both the specific parameters of this model (e.g. start values) and

certain global simulation parameters (e.g. simulation time, selection of output, type of the numerical solver to be used). This information is then used to link the compiled Smile model to the runtime environment, and possibly further external code, producing an efficient, executable simulation program, which can also be accessed interactively by the user.

3 The Modelica Language

The Modelica language integrates concepts from many simulation languages, e.g. ASCEND, Dymola, gPROMS, MOSES, NMF, ObjectMath, SIDOPS+, Smile, and ULM. More specific information can be found on the Modelica website [9].

The first phase of the Modelica design effort focused on continuous systems modeling since for this field there is a generally accepted mathematical framework — differential-algebraic equation (DAE) systems — and a large body of experience. Discrete features also were included from the beginning to allow handling of discontinuities and sampled systems. A primary design goal is extensibility so the language can be successively generalized to a multi-domain and general-purpose modeling formalism. In the following, selected language features are described.

Object-Oriented, Equation-Based Physical Systems Modeling — the Class Concept. By the success of general-purpose programming languages such as Java, Eiffel and C++, object-orientation has proven to be a very powerful concept. It improves the developer's productivity by supporting reuse and extensibility, and provides a high level of abstraction for the modeling of complex systems. However, despite the fact that this concept indeed originated in the realm of simulation (Simula67 language), it is not yet widely applied in modeling and simulation languages.

As, however, *object-oriented modeling* already demonstrated its merits in a few modeling and simulation languages (e.g. ObjectMath, Omola, and Smile), it was decided to base Modelica on this paradigm. Consequently, the *class* is the central unit of modularisation in Modelica, and the main building block of model descriptions. The class concept has strong similarities to that of general-purpose OO-languages, but also differences.

```
class LowPassFilter {
  parameter Real: T=1;
  Real: u, y(Start=1);

  equation
    T*der(y) + y = u;
}
```

Each class has a name and bundles a set of *components* (which are *quantity variables* or *parameters*) with *equations* relating to these quantities and parameters. Each component has a type (which is a class). The example shown right demonstrates some more features:

- The first component declaration contains a `parameter` specifier which asserts that the value of this component will not change during a simulation run. (A more restrictive `constant` specifier exists which asserts that the corresponding value will *never* change.) Parameters can be assigned values in the declaration (`T=1`).
- The second declaration demonstrates how the `Start` attribute (component) (which each `Real` variable has) receives a value by the *modifier* `(Start=1)`.
- The equation section has a single equation which relates the quantities u and y (input and output of the filter). `der(y)` denotes the derivative dy/dt.

Having modeled the `LowPassFilter`, we can demonstrate its use in the model description displayed on the right side. The `class` keyword in Modelica also has a set of specializing synonyms (`block`, `connector`, `model`, `record`, `type`) each of which asserts that this particular class fulfills special restrictions. This enables additional consistency checks and improves

```
class FiltersInSeries {
  // two instances, different parameters
  LowPassFilter: F1(T=2), F2(T=3);

  equation
    // TIME is the independent variable
    F1.u = sin(TIME);
    // connect first to second filter
    F2.u = F1.y;
}
```

readability, but semantically, in a valid model, all of the synonyms can be replaced by `class` without changing the model's behaviour.

Inheritance. Class definitions can use inheritance to reuse the components of existing classes (by inclusion). Multiple inheritance is available (in case of repeated inheritance conflicts are forbidden).

```
class LowPassFilter5 {
  extends LowPassFilter(T=5);
  // ... more extensions
} //     for new class ...
```

In the `extends` clause (which expresses inheritance in Modelica), component modifers are allowed so as to enable specialization of the class being inherited from "on the fly" as shown above.

For simple cases where inheritance as shown above is employed only to make available for reuse a version with minor modifications, there is also an equivalent shorthand syntax as shown to the right.

```
class LowPassFilter5
    = LowPassFilter(T=5);
```

Unlike with general-purpose object-oriented programming languages, the class hierarchy of Modelica is not consulted in type compatibility checks; only structural equivalence is important, not the "inheritance history". (The type system of Modelica is built on ideas from [1].)

Genericity, Redeclarations, Partial Models. Often it is possible — and desirable w.r.t. the goal of reusability — to capture the behaviour of a whole set of models in a single description containing "loose ends", i.e. places where information is missing or represented just by a placeholder. Modelica offers three mechanisms to express that: *type parameters* (called *virtual classes*), a *component redeclaration* feature, and *partial* (incomplete) classes.

Class definitions can contain declarations of the form: `virtualclass PName=DName;`
This introduces `PName` as the name of a type parameter — it can be used as a type name in subsequent component declarations, all of which will be affected when a *type parameter substitution* `substitute class PName=TName;` occurs later on. `DName` is the name of a *default type* for the type parameter; the default type will be substituted as long as no explicit type parameter substitution is active; it also constrains the possible actual types to be substituted.

For the simple case where it is sufficient to change the type of individual components in a descendant class to express the specialization of a model, the `substitute` specifier can also be used to indicate a redeclaration, e.g.: `substitute MyLowPassFilter: F2;`

A class definition can be declared `partial` to indicate that it is incomplete. Such a class cannot be used in component declarations as it is; first it has to be completed by defining a descendant class which fixes the loose ends (missing equations or type parameters without a default type).

Built-in Types and Basic Types. In Modelica, all type names are class names, even the atomic built-in types (`RealType`, `BooleanType` etc.). The *basic types* normally employed by the modeller (`Real`, `Boolean` etc.) are descendant classes of these built-in types. Their definitions are known (and expressed in Modelica), each mainly adding a set of attributes to the corresponding built-in type, e.g.:

```
type Real {
  extends RealType, VariableAttributes;
  parameter StringType: Quantity = "";
  parameter StringType: Unit = "";
  parameter Alternative(Equal, Sum):
                    Connection = Equal;
  parameter RealType:  Min=-Inf,
                       Max=+Inf;
  // Initial and restart value
  parameter RealType:  Start = 0;
  // ...
}
```

Connections. Connections between submodels can be expressed by the special `CONNECT` operator which is used in the equation section. This operator has the effect of generating equations according to the `Connection` attributes of its arguments (i.e. components), which can take the values `Equal`, `Sum`. For `Equal` variables, the arguments of `CONNECT` are simply set equal; for `Sum` variables, a zero-sum equation involving the arguments of the `CONNECT` is generated. This corresponds to the notions of "accross" and "through" variables found in several modeling and simulation systems and languages. `CONNECT` can be applied to structured components, which are automatically decomposed.

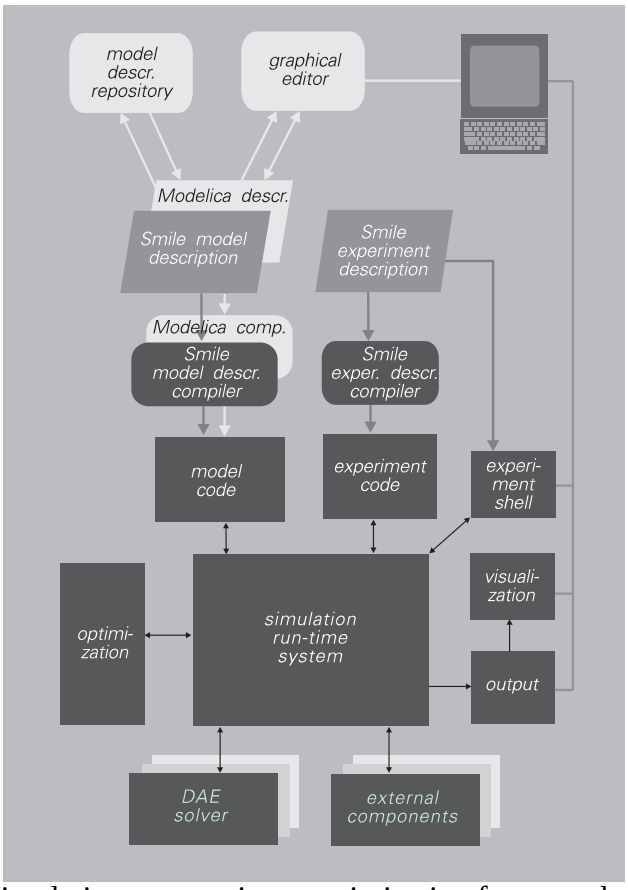
`connector` classes can then be used as convenient encapsulations of complex interfaces between submodels bundling several components (which in turn can be structured as well).

Units and Quantities. It is often considered desirable to associate units of measurement (e.g. from the SI system) and quantity category names ("length", "pressure", etc.) to the quantity variables found in a model description, so many simulation languages or systems offer features for that purpose. That way, more "physical" information is retained in the model description, allowing e.g. unit-based consistency checks of equations.

As, however, not all modeling languages which Modelica is intended to unify can handle this kind of information, Modelica does not "hard-wire" any details (such as a concrete system of physical units and quantity names) into the language. This information is completely optional, and Modelica only offers standardized "hooks" ensuring that it can be associated to the quantity variables and retrieved in a unified manner. Even that is not provided in the language itself, but in the (standardized) definition of base types: e.g. the `Real` class above contains string attributes `Quantity` and `Unit` which are reserved for the purposes described above. The official definition of Modelica will specify the use of these attributes in more detail (e.g. refer to a standardized syntax for expressing compound SI units by ASCII character strings).

Other Features. In this paper, only a subset of important features of the Modelica language were presented. Besides those parts of the language that strongly resemble existing general-purpose programming languages (e.g. the basic expression syntax, user-defined functions), there are special features to support regular model structures, hybrid (continuous/discrete) models, conditional and structure-changing models, the mapping of model descriptions to graphic symbols, matrix operations, and many others, some of which are still subject to ongoing development.

4 The Smile/M Architecture



The Smile system [5] [12] is being extended by a *Modelica compiler*. Rather than generating Smile model descriptions as an intermediate form, the Modelica compiler will directly generate the model code suitable for cooperating with the other components of the Smile system. As was the original Smile compiler, this compiler is being developed using advanced compiler generation tools. The runtime system is extended to support features currently not found in the Smile modeling language, but which are required to support the Modelica language., e.g. support for conditional and structure-changing models.
The separation of model and experiment description in the original Smile system has proven valuable and will be maintained in the new system. Since, however, the experiment description compiler turned out to be somewhat unflexible, an interpretative *experiment shell* is being added based on an existing "scripting language". In addition to the simulation configuration in the experiment description, it allows to customize the flow of control of a simulation run and to embed a simulation run e.g. in an optimization framework.

Further new components, such as a generic *graphical model editor* supporting several graphical modeling styles, the application of an object-based repository for storing model and experiment descrip-

tions, and improvements to the existing *visualization* and *optimization components* are being considered, too.

5 Conclusions

This paper presented the extension of Smile, a modeling and simulation environment developed at TU Berlin and GMD FIRST, to Modelica, an emerging (quasi-)standard language for object-oriented, equation based, hybrid modeling and simulation. Modelica generalizes on concepts found in Smile and opens up new dimensions of interoperability. An overview of this language and its development was given.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, Berlin, 1996.

[2] M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. Department of Automatic Control, Lund Institute of Technology, Lund, 1994.

[3] J. Barby. The need for a unified modeling language and VHDL-A. In *Proceedings of 1996 IEEE International Symposium on Computer-Aided Control System Design*, pages 258–263, 1996.

[4] P.I. Barton and C.C. Pantelides. Modeling of combined discrete/continuous processes. *AIChE J.*, 40:966–979, 1994.

[5] M. Biersack, V. Friesen, S. Jähnichen, M. Klose, and M. Simons. Towards an architecture for simulation environments. In T. I. Vren and L. G. Birta, editors, *Proceedings of the Summer Computer Simulation Conference (SCSC'95)*, pages 205–212. The Society for Computer Simulation, 1995.

[6] G. Bartsch et al. Entwicklung rechnergestützter Simulationshilfen zur Beschreibung des Energieverhaltens komplexer energiewandelnder Systeme. Abschlußbericht des universitären Forschungsschwerpunktes 4 der Technischen Universität Berlin. Teil 1. Technical report, TU Berlin, 1997.

[7] P. Fritzson et al. High-level mathematical modeling and programming. *IEEE Software*, 12, July 1995.

[8] IEEE DASC 1076.1 Working group. Analog and mixed signal extensions to VHDL. http://vhdl.org/vi/analog, 1997.

[9] The Modelica Design Group. Modelica – a unified object-oriented language for physical systems modeling (draft). http://www.dynasim.se/Modelica/, 1997.

[10] D. Brück H. Elmqvist and M. Otter. Dymola — user's manual. Technical report, Dynasim AB, Research Park Ideon, Lund, Sweden, 1996.

[11] C. Klein-Robbenhaar. Numerical methods for dynamic simulation of thermal energy systems: a case study. In this volume.

[12] M. Kloas, V. Friesen, and M. Simons. Smile — A simulation environment for energy systems. In A. Sydow, editor, *Proceedings of the 5th International IMACS-Symposium on Systems Analysis and Simulation (SAS'95)*, volume 18–19 of *Systems Analysis Modelling Simulation*, pages 503–506. Gordon and Breach Publishers, 1995.

[13] H. Tummescheit and R. Pitz-Paal. Simulation of a solar thermal central receiver power plant. In this volume.