Multi-Aspect Modeling in Equation-Based Languages

Dirk Zimmer, Inst. of Computational Science, ETH Zürich, Switzerland, *dzimmer@inf.ethz.ch*

Current equation-based modeling languages are often confronted with tasks that partly diverge from the original intended application area. This results out of an increasing diversity of modeling aspects. This paper briefly describes the needs and the current handling of multi-aspect modeling in different modeling languages with a strong emphasis on Modelica. Furthermore a small number of language constructs is suggested that enable a better integration of multiple aspects into the main-language. An exemplary implementation of these improvements is provided within the framework of Sol, a derivative language of Modelica.

Motivation

Contemporary equation-based modeling languages are mostly embedded in graphical modeling environments and simulators that feature various types of datarepresentation. Let that be for instance a 3D-visualization or a sound module. Consequently the corresponding models are accompanied by a lot of information that describes abundantly more than the actual physical model. This information belongs to other aspects, such as the modeling of the iconographic representation in the schematic editor or the preference of certain numerical simulation techniques. Hence, a contemporary modeler has to cope with many multiple aspects.

In many modeling languages such kind of information is stored outside the actual modeling files, often in proprietary form that is not part of any standard. But in Modelica [6], one of the most important and powerful EOO-languages, the situation has developed in a different way. Although the language has been designed primarily on the basis of equations, the model-files may also contain information that is not directly related to the algebraic part. Within the framework of Modelica, the most important aspects could be categorized as follows:

- *Physical modeling:* The modeling of the physical processes that are based on differential-algebraic equations (DAEs). This modeling-aspect is also denoted as the primary aspect.
- *System hints:* The supply of hints or information for the simulation-system. This concerns for example hints for the selection of state-variables or start values for the initialization problem.
- *3D Visualization:* Description of corresponding 3Dentities that enable a visualization of the models
- *GUI-Representation:* Description of an iconographic representation for the graphical user-interface (GUI) of the modeling environment.
- *Documentation:* Additional documentation that addresses to potential users or developers.

We will use this classification for further analysis, since it covers most of the typical applications fairly well. Nevertheless, this classification of modeling aspects is of course arbitrary, like any other would be.

Let us analyze the distribution of these aspects with respect to the amount of code that is needed for them. Figure 1 presents the corresponding pie-charts of three exemplary models of the Modelica standard library. These are the “FixedTranslation” component for the MultiBodylibrary, the PMOS model of the electrical package and the “Pump and Valve” model in the Thermal library. The first two of them represent single components; the latter one is a closed example system.

In the first step of data-retrieval, all unnecessary formatting has been removed from the textual model-files. For each of these models, the remaining content has then been manually categorized according to the classification presented above. The ratio of each aspect is determined by counting the number of characters that have been used to model the corresponding aspect.

The results reveal that the weight of the primary aspect cannot be stated to be generally predominant. The distribution varies drastically from model to model. It varies from only 14% to 53% for these examples.

Yet one shall be careful by doing an interpretation of the pie-charts in figure 1. The weight of an aspect just expresses the amount of modeling code with respect to the complete model. This does not necessarily

correlate with the invested effort of the modeler and even less it does correlate with the overall importance of an aspect. It needs to be considered that code for the GUIrepresentation is mostly computer-generated code that naturally tends to be lengthy. On the other hand side, the code that belongs to the primary aspect of equation-based modeling is often surprisingly short. This is due to the fact that this represents the primary strength of Modelica. The language is optimized to those concerns and enables convenient and precise formulations. Unfortunately, this can hardly be said about the other aspects in our classification.

The discussion about the Modelica and other EOOlanguage is often constrained to its primary aspect of physical modeling. But in typical models of the Modelica standard-library this primary aspect often covers less than 25% of the complete modeling code. Any meaningful interpretation of figure 1 reveals that the disregard on other modeling aspects is most likely inappropriate especially when we are concerned with language design. For any modeling language that owns the ambition to offer a comprehensive modeling-tool, the ability to cope with multiple aspects has become a definite prerequisite.

It is the aim of this paper to improve modeling languages with respect to these concerns. To this end, we will suggest certain language constructs that we have implemented in our own modeling language: Sol. The application of these constructs will be demonstrated by a small set of examples. But first of all, let us take a look at the current language constructs in Modelica and other modeling languages.

1 Current handling of multiple aspects

1.1 Situation in VHDL-AMS, Spice, gPROMS, Chi

The need for multiple aspects originates primarily from industrial applications. Hence this topic is often not concerned for languages that have a strong academic appeal. One example for such a language is Chi [3]. For the sake of simplicity and clarity, this language is very formal and maintains its focus on the primary modeling aspect.

In contrast, languages like SPICE3 [9] or VHDL-AMS [1,10] and Verilog-AMS[12] are widely used in industry. Unlike Modelica, these languages do typically not integrate graphical information into their models. The associated information that describes the schematic diagram and the model icons is often separately stored, often in a proprietary format. For instance, the commercial product Simplorer [11] generates its own proprietary files for the model-icons. The corresponding VHDL-code does not relate to these files.

However, different solutions are possible: both AMSlanguages contain a syntax-definition for attributes. These can be used to store arbitrary information that relate to certain model-items. Since there is only a small-number of predefined attributes (as unit descriptors, for instance), most of the attributes will have to be specified by the corresponding processing tools.

Furthermore these two languages and SPICE3 own an extensive set of predefined keywords. This way it is possible to define output variables or to configure simulation parameters. The situation is similar in ABACUSS II [5], which is the predecessor to gPROMS [2]. This language offers a set of predefined sections that address certain aspects of typical simulation run like initialization or output.

1.2 Multiple aspects in Modelica

The Modelica language definition contains also a number of keywords that enable the modeler to describe certain aspects of his model. For instance, the attributes `stateSelect` or `fixed` represent system-hints for the simulator. In contrast to other modeling languages, Modelica introduced the concept of annotations. These items are placed within the definitions of models or the declarations of members and contain content that directly relates on them. Annotations are widely used within the framework of Modelica. The example below presents an annotation that describes the position, size and orientation of the capacitor icon in a graphic diagram window.

```
1 Capacitor C1(C=c1) "Main capacitor"
2    annotation (extent  =[50, -30; 70, -10],
3                rotation=270);
```

Listing 1. Use of an annotation in Modelica

Since annotations are placed alongside the main modeling code, they inflate the textual description and tend to spoil the overall clarity and beauty. A lot of annotations contain also computer-generated code that hardly will be interesting for a human reader. Thus, typical Modelica editors mostly hide annotations and make them only visible at specific demand of the user. However, this selection of code-visibility comes with a price. First it reduces the convenience of textual editing, since cut, copy and paste opera-

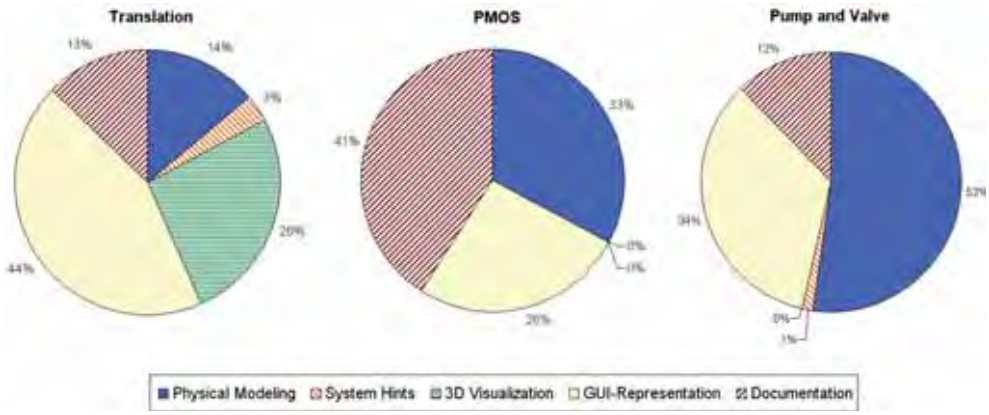

Figure 1. Code distribution of aspects in Modelica models.

tions may involve hidden annotations. Second, the selection of visibility happens on a syntactical level not on a semantic level.

Storing data for GUI-representation or other specific hints and information has been initially a minor topic in the design process of Modelica. Still, there was a compelling need for it. To meet these urgent requirements, the Modelica community decided to introduce the concept of annotations into the modeling language. Already the first language definition of Modelica contained the concept of annotations and also presented some applications for GUI-representation and documentation. The corresponding annotations have been used as a quasi-standard despite the fact that they only have been weakly documented. Annotations served also as an official back-door entrance to non-official, proprietary functionalities. Since it happens frequently in software engineering that certain things just grow unexpectedly, many further annotations have been introduced meanwhile. Nowadays, annotations contain a lot of crucial content that revealed to be almost indispensable for the generation of effective portable code. Therefore it is no surprise that just recently a large set of annotations had to be officially included in version 3 of the Modelica language definition [8]. This way, what started out as a small, local and semi-proprietary solution, became now a large part in the official Modelica standard.

To store the information that belongs to certain aspects, different approaches are used in Modelica and often more than one language-tool is involved. The following list provides a brief overview on the current mixture of data representation:

- The physics of a model is described by DAEs and is naturally placed in the main Modelica model.
- Hints or information for the simulation-system are mostly also part of the main Modelica language but some of them have to be included in special annotations.
- Information that is used by the GUI is mostly included in annotations. But the GUI uses also uses information from textual descriptions that are part of the main-language.
- The description of 3D-visualization is done by dummy-models within main-Modelica code.
- Documentation may be extracted from the textual descriptions that accompany declarations and definitions, but further documentation shall be provided by integrating HTML-code as a text-string into a special annotation. Other annotations store information about the author and the library version.

1.3 Downfalls of the current situation

Obviously, this fuzzy mixture of writings and language constructs reveals the lack of a clear, conceptual approach. As nice as the idea of annotations appears in the first moment, it also incorporates a number of problematic insufficiencies.

The major drawback is that only pre-thought functionalities are applicable. The modeler has no means to define annotation by its own or to adapt given constructs to his personal demands. Furthermore, syntax and semantics of each annotation needs to be defined in the language definition. Since there is always a demand for new functionalities, the number of annotations will continue to increase. This leads to a foreseeable inflation of the Modelica language definition.

1.4 Lack of expressiveness

These downfalls originate from a lack of expressiveness in the original Modelica language. Whenever one is concerned with language design [7], it is important to repetitively ask some fundamental questions. How can it be that a language so powerful to state highly complicated DAE-systems is unable to describe a rectangle belonging to an iconographic representation? Why do we need annotations at all?

These questions are clearly justified and point to the fact that the development scope of the Modelica language might have been too narrowly focused on the equation based part. Therefore, extension that would have been of great help in other domains, have been left out:

- There is no suitable language construct that enables the declaration of an interface to an environment that corresponds to a certain aspect.
- Instances of objects cannot be declared anonymously within a model.
- The language provides no tool for the user that enables him or her to group statements into semantic entities.
- The language offers no means to refer on other (named) objects, neither statically nor dynamically.

By removing these four lacks, we will demonstrate that the use of annotations can be completely avoided and that the declarative modeling of multiple aspects can be handled in a conceptually clear and concise manner. The following section will discuss this in more detail and provide corresponding examples.

2 Multi-aspect modeling in Sol

Sol is a language primarily conceived for research purposes. It owns a relatively simple grammar (see appendix) that is similar to Modelica. Its major aim is to enable the future handling of variable-structure systems. To this end, a number of fundamental concepts had to be revised and new tools had to be introduced into the language. The methods that finally have become available suit also a better modeling of multiple aspects. These methods and their application shall now be presented.

2.1 Starting from an example

In prior publications on Sol [13,14] the "Machine" model has been introduced as standard example. It contains a simple structural change and consists of an engine that drives a flywheel. In the middle there is a simple gear box. Two versions of an engine are available: The first model `Engine1` applies a constant torque. In the second model `Engine2`, the torque is dependent on the positional state, roughly emulating a piston-engine. Our intention is to use the latter, more detailed model at the machine's start and to switch to the simpler, former model as soon as the wheel's inertia starts to flatten out the fluctuation of the torque. This exchange of the engine model represents a simple structural change on run-time.

```
1  model Machine
2  implementation:
3     static Mechanics.FlyWheel F{inertia<<1};
4     static Mechanics.Gear G{ratio<<1.8};
5     dynamic Mechanics.Engine2 E {meanT<<10};
6
7     connection c1(a << G.f2, b << F.f);
8     connection c2(a << E.f, b << G.f1);
9     when F.w > 40 then
10      E <- Mechanics.Engine1{meanT << 10};
11    end;
12 end Machine;
```

Listing 2. Simple machine model in Sol.

The first three lines of the implementation declare the three components of the machine: fly-wheel, gear-box and the engine. The code for the corresponding connections immediately follows. The third component that represents the engine is declared dynamically. This means that the binding of the corresponding identifier to its instance is not fixed and a new instance can be assigned at an event. This is exactly what happens in the following declaration of the when-clause. A new engine of compatible type is declared and transmitted to the identifier E. The old engine-model is thereby implicitly removed and the corresponding equations are automatically updated.

This model contains the physics part only. We now want to add other aspects to the model. We would like to add a small documentation and to specify the simulation parameters. Furthermore we want to add information about model's graphical representation in a potential, graphical user-interface. The following subsections will present the necessary means and their step by step application.

2.2 Environment packages and models

Many modeling aspects refer to an external environment that is supposed to process the exposed information. This environment may be the GUI of the modeling environment or a simulator program. Therefore it needs to be specified how a model can address a

potential environment. To this end, Sol features environment packages and models that enable to define an appropriate interface. Let's take a look at an example:

```
1  environment package Documentation
2     model Author
3       interface:
4          parameter string name;
5     end Author;
6     model Version
7       interface:
8          parameter string v;
9     end Version;
10    model ExternalDoc
11      interface:
12         parameter string fname;
13    end ExternalDoc;
14 end Documentation
```

Listing 3. Environment package.

This example consists in a package that contains models which can be used to store relevant information for the documentation of arbitrary models. The keyword `environment` does specify that the models of the corresponding package address the environment and are therefore not self-contained. They merely offer an interface instead. The actual implementation and semantics of the package remains to be specified by the environment itself.

It is important to see that stipulating the semantics would be a misleading and even futile approach. Different environments will inevitable have to feature different interpretations of the data. For instance, a pure simulator will complete ignore the "Documention" models whereas a modeling editor may choose to generate an HTML-code out of it. Nevertheless it is very meaningful to specify a uniform interface within the language. This provides the modeler with an overview of the available functionalities. Furthermore the modeler may choose to customize the interface for its personal demands using the available object-oriented means of the Sol-language.

2.3 Anonymous declaration

The language Sol enables the modeler to anonymously declare models anywhere in the implementation. The parameters can be accessed by curly brackets whereas certain variable members of the model's interface are accessible by round brackets. This way, the modeler can address its environment in a convenient way just by declaring anonymous models of the corresponding package. An application of this methodology is presented below in listing 4 for the Machine model.

Anonymous declarations are an important element of Sol, since they enable the modeler to create new instances on the fly, for example at the execution of an event. This is very helpful for variable-structure systems. However, within the context of multi-aspect modeling, anonymous declarations serve primarily convenience. It is of course possible to assign names to each of the documentation items. They can be declared with an identifier like any other model, but this is mostly superfluous and would lead to bulky formulations.

```
1  model Machine
2    implementation:
3       […]
4       when F.w > 40 then
5          E <- Mechanics.Engine1{meanT << 10 };
6       end;
7       Documentation.Author{name<<"DirkZimmer"};
8       Documentation.Version{v << "1.0");
9       Documentation.ExternalDoc
                     {fname<<"MachineDoc.html"};
10 end Machine;
```

Listing 4. Use of anonymous declarations.

2.4 Model sections

Sol has been extended by the option for the modeler to define sections using an arbitrary package name. Sections incorporate three advantages: One, code can be structured into semantic entities. Two, sections add convenience, since the sub-models of the corresponding package can now be directly accessed. Three, section enable an intuitive control of visibility. A modern text editor may now hide uninteresting sections. The user may then be enabled to toggle the visibility according to its current interests. This way, the visibility is controlled by semantic criteria and not by syntactical or technical terms.

```
1  model Machine
2    implementation:
3       […]
4       when F.w > 40 then
5          E <- Mechanics.Engine1{meanT << 10 };
6       end;
7       section Documentation:
8          Author{name << "Dirk Zimmer"};
9          Version{v << "1.0"};
10         ExternalDoc{fname<<"MachineDoc.html"};
11      end;
12      section Simulator:
13         IntegrationTime{t << 10.0};
14         IntegrationMethod{method<<"euler",
15         step << "fixed", value << 0.01};
16      end;
17 end Machine;
```

Listing 5. Sections

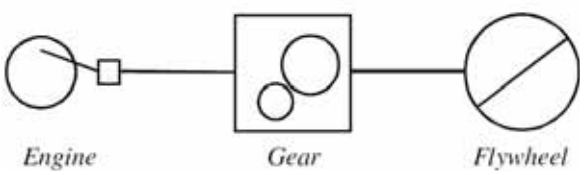

Figure 2. Diagram representation

The documentation part of the machine model has now been wrapped within a section. A second section addresses another environment called "Simulator" and shows an exemplary specification of some simulation parameters. Both sections could be hidden by an editor if the user has no interest in their content.

2.5 Referencing of model instances

The provided methods so far, are fully sufficient for simple application cases. The proper implementation of a GUI-representation is yet a more complex task that demands a more elaborate solution. In the classic GUI-framework for object-oriented modeling, each model owns an icon and has a diagram window that depicts its composition from sub-models. Figure 2 displays the aspired diagram of the exemplary machine-model that contains the icons of its three sub-models. The connections are represented by single lines. The following paragraphs outline one possible solution in Sol.

The problem is that many models will own GUI information but only the information of certain model instances shall be acquired. This originates in the need for language constructs that enable hierarchical or even mutual referencing between model-instances. Sol meets these requirements by giving model-instances a first-class status [4]. This means that model-instances cannot only be declared anonymously but also these instances can be transmitted to other members or even to parameters.

This capability had already been applied in listing 2 to model the structural change of the engine. The statement

```
E <- Mechanics.Engine1(meanT << 10)
```

declares anonymously an instance of the model "Engine1" and then transmits this instance to the dynamic member E. Hence the binding of the identifier to its instance gets re-determined which causes a structural change.

A similar pattern will occur in our solution for the GUI-design. Let us take a look at the corresponding environment-package.

- environment package Graphics
  - model Line
  - model Rectangle
  - model Ellipse
  - model Canvas
    - model Line
    - model Rectangle
    - model Ellipse
  - model GraphicModel

Figure 3. Structure of the Graphics package.

Figure 3 gives a structural overview of the environment package `Graphics`. This package provides rudimentary tools for the design of model-icons and diagrams. These are represented by models for rectangles, ellipses and lines. The package contains also a `Canvas` model that enables drawings on a local canvas. Furthermore the package contains a partial model `GraphicModel` that serves as template for all models that support a graphical GUI-representation. It defines two sub-models: one for the icon-representation and one for the diagram representation. Models that own a graphical representation are then supposed to inherit this template model. Please note that the icon has a canvas model as parameter.

```
1 model GraphicModel
2 interface:
3    model Icon
4    interface:
5       parameter Canvas c;
6    end Icon;
7    model Diagram
8    end Diagram;
9 end GraphicModel;
```

Listing 6. A template for graphical models.

A graphical modeling environment may now elect to instantiate one of these sub-models. This will cause further instantiations of models belonging to the "Graphics"-package that provide the graphical environment with the necessary information. Below we present an exemplary icon model for our engine that corresponds to the icon in Figure 2.

```
10 model Engine2 extends Interfaces.OneFlange;
11                          // that extends GraphicalModel
12 interface:
13   parameter Real meanT;
14   redefine model Icon
15   implementation:
16      c.Ellipse(sx<<0.0, sy<<0.2,
                  dx<<0.6, dy<<0.8);
17      c.Rectangle(sx<<0.9, sy<<0.45,
                    dx<<1.0, dy<<0.55);
18      c.Line(sx<<0.3, sy<<0.3,
               dx<<0.9, dy<<0.5);
```

```
19     end Icon;
20  implementation:
21     […]
22  end Engine2;
```

Listing 7. An implementation of an icon

The icon of listing 7 "paints" on a local canvas that is specified by the corresponding parameter c. The transmission of this parameter is demonstrated in Listing 8 that represents the whole diagram of figure 2. This model declares the icons of its sub-models and creates a local canvas for each of them by an anonymous declaration. The two connections c1 and c2 also own a Line-model for their graphical representation.

```
1  model Machine extends Graphics.GraphicalModel;
2  interface:
3     redefine model Diagram
4     implementation:
5        section Graphics:
6           F.Icon{c<<Canvas{x<<10, y<<10,
                            w<<10, h<<10}};
7           G.Icon{c<<Canvas{x<<30, y<<10,
                            w<<10, h<<10}};
8           E.Icon{c<<Canvas{x<<50, y<<10,
                            w<<10, h<<10}};
9           c1.Line(sx<<20, sy<<15,
                    dx<<30, dy<<15);
10          c2.Line(sx<<40, sy<<15,
                    dx<<50, dy<<15);
11          c.Rectangle(0,0,70,30);
12       end;
13    end Diagram;
14 implementation:
15    […]
16    section Documentation:
17    […]
18    section Simulator:
19    […]
20 end Machine;
```

Listing 8. An implementation of a diagram

The "GraphicalModel" involves another key-concept of Sol. The language enables the modeler to define models also as member-models in the interface section. When instantiated, these models belong to their corresponding instance and are therefore not independent. This means that the Diagram or Icon model always refer to their corresponding super-instance. Consequently, they also have access to all the relevant parameters and can adapt.

Please note that the resulting GUI-models are potentially much more powerful than their annotation-based counterparts in Modelica. All the modeling power of Sol is now also available for the graphical models. For instance, only a minimal effort is needed to make the look of an icon adapt to the values of a model-parameter. No further language construct would be required. A model could even feature "active" icons that display the current system-state and hence enable a partial animation of the system within the diagram-window. Even the structural change of the machine-model could be made visible in the diagram during the simulation. Such extensions (if desired or not) become now feasible and demonstrate the flexibility of this approach.

However, the provided examples are merely a suggestion and represent just one possible and convenient solution within the framework of Sol. There are also many other language constructs that would lead to feasible or even more general solutions. Many of them could easily be integrated into equation-based languages. Some of them are featured in Sol. With respect to Modelica, this is unfortunately not the case yet.

3 Conclusion

Handling complexity in a convenient manner and organizing modeling knowledge in a proper form have always been primary motivations for the design of modeling languages. The introduction of object-oriented mechanism has yield to a remarkable success and drastically simplified the modeling of complex systems. Object-orientation essentially enabled the modeler to break models into different levels of abstraction. Hence, the knowledge could be organized with respect to depth.

However, certain models combine many different aspects that have to be linked together at a top level. Here the knowledge needs to be organized with respect to breadth. For those tasks, current mechanisms in EOOlanguages are underdeveloped.

This paper focuses on *four conceptual language constructs* for EOO-languages that in combination drastically increase the ability to deal with multiple aspects. These are:

1. *Environment-packages* that enable the aspectspecific declaration of interfaces.
2. *Anonymous declarations* of model instances.
3. *Sections* can be used to form semantic entities and control visibility.
4. *Referencing mechanisms* between model-instances. (In Sol, these mechanisms are pro-

vided by giving model-instances a first class status and enabling socalled member-models.)

The proposed constructs have been implemented in our experimental language Sol and their application is demonstrated by a set of corresponding examples. The resulting advantages of this approach are manifold:

- The methods how to address a potential environment are made available within the language. The modeler may browse through the provided functionalities like she or he is used to do it for standard libraries.
- The existing object-oriented mechanisms can be applied on these environment-models. Hence the modeler can customize the interface for its personal demands and is not constrained to a predefined solution.
- Anonymous declarations enable a convenient usage of these models, anywhere in the implementation. The resulting statements are naturally readable and integrate nicely into the primary, equation-based part.
- User-defined sections help to organize the model and offer an excellent way to filter for certain modeling aspects. Uninteresting information may consequently be hidden without hindering the editing of the code. The filtering criteria are not based on syntax anymore, there are based on semantic entities that have been formed by the modelers themselves. Furthermore sections enable a clear separation of computer generated modeling code.
- The embedment into an existing object-oriented framework enables a uniform approach for a wider range of modeling aspects. Furthermore, it increases the interoperability between these aspects.

However, the most important conclusion is that the ability of the language to help and to extend itself by its own means has been drastically improved with respect to other languages like Modelica. Further development is now possible within the language does not require a constant update and growth of the language definition.

4 Appendix

The following listing of rules in extended Backus-Naur form (EBNF) presents an updated version of the core grammar for the Sol modeling language. The rules are ordered in a top-down manner listing the high-level constructs first and breaking them down into simpler ones. Non-terminal symbols start with a capital letter and are written in bold. Terminal symbols are written in small letters. Special terminal operator signs are marked by quotation-marks. Rules may wrap over several lines.

The inserted modifications concern solely the modeling of multiple aspects. With respect to a prior version of the grammar [13], the changes are minor and concern only 3 rules: `ModelSpec`, `Statement` and `Section`.

```
Model       = ModelSpec Id Header
              [Interface] [Implemen] end Id ";"
ModelSpec   = [redefine | partial | environment]
              (model | package | connector | record)

Header      = {Extension} {Define} {Model}
Extension   = extends Designator ";"
Define      = define (Const | Designator) as Id ";"
Interface   = interface ":" {(IDecl | ParDecl) ";"} {Model}
ParDecl     = parameter Decl
IDecl       = [redelcare] LinkSpec [IOSpec] [CSpec] Decl
ConSpec     = potential | flow
IOSpec      = in | out

Implemen    = implementation ":" StmtList
StmtList    = [Statement {";" Statement }]
Statement   = [ Section | Condition | Event |
              Declaration | Relation ]

Section     = section Designator ":" StmtList end [section]
Condition   = if Expression then StmtList ElseCond
ElseCond    = (else Condition)|([else then StmtList]
              end [if])
Event       = when Expression then StmtList ElseEvent
ElseEvent   = (else Event)|([else then StmtList]
              end [when]
Declaration = [redeclare] LinkSpec Decl
LinkSpec    = static | dynamic
Decl        = Designator Id [ParList]

Relation    = Expression Rhs
Rhs         = ("=" | "<<" | "<-") Expression

ParList     = "{" [Designator Rhs {"," Designator Rhs }] "}"
InList      = "(" [Designator Rhs {"," Designator Rhs }] ")"

Expression  = Comparis {(and|or) Comparis }
Comparis    = Term [("<"|"<="|"=="|"<>"|">="|">")Term]
Term        = Product {( "+" | "-" ) Product }
Product     = Power { ("*" | "/") Power }
Power       = SElement {"^" SElement }
SElement    = [ "+" | "-" | not ] Element
Element     = Const | Designator [InList] [ParList]
              | "(" Expression ")"
Designator  = Id {"." Id }
Id          = Letter {Digit | Letter}
Const       = Number | Text | true | false
Number      = ["+"|"-"] Digit { Digit }
              ["." {Digit }] [e ["+"|"-"] Digit { Digit }]
Text        = "\"" {any character} "\""
Letter      = "a" | ... | "z" | "A" | ... | "Z" | "_"
Digit       = "0" | ... | "9"
```

Listing 9. EBNF grammar of Sol

Acknowledgements

I would like to thank Prof. Dr. François E. Cellier for his helpful advice and support. This research project is sponsored by the Swiss National Science Foundation (SNF Project No. 200021-117619/1).

References

[1] P.J. Ashenden, G.D. Peterson, D.A. Teegarden. *The System Designer's Guide to VHDL-AMS* Morgan Kaufmann Publishers. 2002.

[2] P.I. Barton and C.C. Pantelides. *Modeling of Combined Discrete/Continuous Processes*. American Institute of Chemical Engineers Journal. 40, pp.966-979, 1994.

[3] D. A. van Beek, J.E. Rooda. *Languages and Applications in Hybrid Modelling and Simulation: Positioning of Chi*. Control Engineering Practice, 8(1), pp.81-91, 2000

[4] R. Burstall. *Christopher Strachey – Understanding Programming Languages*. Higher-Order and Symbolic Computation 13:52, 2000.

[5] J.A. Clabaugh, *ABACUSS II Syntax Manual, Technical Report*. Massachusetts Institute of Technology. http://yoric.mit.edu/abacuss2/syntax.html. 2001.

[6] P. Fritzson. *Principles of Object-oriented Modeling and Simulation with Modelica 2.1*, John Wiley & Sons, 897p. 2004.

[7] C.A.R. Hoare. *Hints on Programming Language Design and Implementation*. Stanford Artificial Intelligence Memo, Stanford, California, AIM-224, 1973.

[8] *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling* Language Specification Version 3.0. Available at www.modelica.org .

[9] T.L. Quarles. *Analysis of Performance and Convergence Issues for Circuit Simulation*. PhDDissertation. EECS Department University of California, Berkeley Technical Report No. UCB/ERL M89/42, 1989.

[10] P. Schwarz, C. Clauß, J. Haase, A. Schneider. *VHDL-AMS und Modelica - ein Vergleich zweier Modellierungssprachen*. Symposium Simulationstechnik ASIM2001, Paderborn 85-94, 2001.

[11] Ansoft Corporation: *Simplorer* Avaiable at: http://www.simplorer.com .

[12] *Verilog-AMS Language Reference Manual Version 2.2* Available at http://www.designers-guide.org/VerilogAMS/.

[13] D. Zimmer. *Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems*. Proc. 6th International Modelica Conference, Bielefeld, Germany, Vol.1 47-56, 2008.

[14] D. Zimmer. *Enhancing Modelica towards variable structure systems*. Proc. 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, Berlin, Germany, 61-70, 2007.

Corresponding author: Dirk Zimmer
Institute of Computational Science
ETH Zürich, Switzerland,
dzimmer@inf.ethz.ch

Accepted EOOLT 2008, June 2008
Received: July 30, 2008
Accepted: August 10, 2007