Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems

Dirk Zimmer
Institute of Computational Science, ETH Zürich
CH-8092 Zürich, Switzerland
dzimmer@inf.ethz.ch

Abstract

This paper presents a derivative language of Modelica that is called Sol. It has been especially designed for the convenient expression and simulation of variable-structure systems within an object-oriented, equation-based modeling framework. Starting from a formal definition of the grammar and type-system, the paper advances to an explanation of Sol's semantics. Finally the current state of implementation and corresponding processing mechanisms are presented. ***Keywords:*** *language design, variable-structure systems, causalization mechanisms.*

1 Motivation

Many contemporary models contain structural changes at simulation run-time. These systems are typically denoted by the collective term: variable-structure systems. The motivations that lead to the generation of such systems are manifold. Typical cases are represented by ideal switching or breaking processes, variability in the number of entities, dynamic multi-level models or user interaction [5].

Let us focus on the modeling-paradigm that is represented by Modelica: declarative models that are based on differential algebraic equations (DAEs) with hybrid extensions. Within this paradigm, a structural change is reflected by a change in the set of variables and by a change in the set of relations (i.e., equations) between the time-dependent variables. Such replacements may lead to severe changes in the model structure. This concerns the causalization of the equation system, as well as the perturbation index of the DAE system.

Current contributions of this research domain include the development of the language MOSILAB [8] or Hydra [7]. Also some specific techniques, like inline-integrators [4] that can be included in Modelica prove to be helpful in certain situations. However, most of these approaches leave the standard domain of Modelica, since the modeling of variable-structure systems within the current Modelica framework is very limited [10]. This is partly due to a number of technical restrictions that mostly originate from the static treatment of the DAEs. But these technical restrictions are not the only limiting factor. Another major problem is the lack of expressiveness in the Modelica-language itself.

To express structural changes, a corresponding modeling language has to meet certain requirements. For instance, it must be enabled to state relations between variables or sub-models in a conditional form, so that the structure can change depending on time and state. In addition, variables and sub-models should be dynamically declarable, so that the corresponding instances can be created, handled, and deleted at run time. Such requirements partly contradict with fundamental assumptions made in the design-process of Modelica.

Therefore we decided to develop a new language called Sol [11]. It is a language primarily conceived for research purposes that attempts to be of minimal complexity with a high degree of expressiveness. We want to explore the full power of a declarative modeling approach and how it can handle potential, future problem fields. The implementation of Sol will be a small and open project that should enable other researchers to test and validate their ideas with a moderate effort. The longer term goal of our research is to significantly extend Modelica's expressiveness and range of application. Furthermore, the Sol-project gives us a development-platform for technical solutions that concerns the handling of structurally changing equation systems. This includes solutions for dynamic recausalization or the dynamic handling of structural singularities.

Although Sol forms a language of its own, it is designed to be as close to Modelica as reasonably possible. This should drastically ease the understanding for anyone in the Modelica community. It is not our goal to immediately change the Modelica standard or to establish an alternative modeling language. Our scientific work is intended to merely offer suggestions and guidance for Modelica's future development.

Example 1: Model of a simple machine driving a fly-wheel with a fluctuating torque.

```
model SimpleMachine


   define inertia as 1.0;


interface:
   parameter Real meanTorque;
   static Real w;

implementation:
   static Real torque;
   static Real a;
   torque = inertia*z;
   w = int(x=z);
   phi = int(x=w);
   torque = (1+cos(p=phi))*meanTorque;

end SimpleMachine
...
static SimpleMachine M1{ meanT << 10};
cout << SimpleMachine.w;
```

Definition of the main model: “SimpleMachine”

(1) Header:
- Definition of a constant

(2) Interface Section:
- Declaration of parameters
- Declaration of a public member

(3) Implementation part:
- Declaration of private members
- Stating Newton’s law of motion…
- Equation for the fluctuating torque

End of the model-definition

- An example declaration of the machine model.
- Simple Output Generation.

2 The Language: Sol

2.1 Principle Components

Essentially, Sol redefines the fundamental concepts of Modelica on a dynamic basis. Following the spirit of Modelica, it forms a language of strong declarative character and therefore completely abandons any imperative parts. Unlike many other declarative modeling languages, Sol enables the creation, exchange and destruction of components at simulation time. To this end, the modeler describes the system in a constructive way, where the structural changes are expressed by conditionalized declarations. These conditional parts can than get activated and deactivated of during run-time. This constructive approach avoids an explicit description of modes and transitions and yet proves to be fairly powerful and flexible.

In contrast to Modelica, the grammar of Sol (cf. appendix) is significantly stricter. In its aim for simplicity, it prohibits any ambiguous ordering of its major sections. Also any grammar elements that one would typically denote by the term syntactic sugar are largely omitted. Whereas the strict section ordering definitely leads to a good modeling style, the lack of syntactic shorthand notations may sometimes result in clumsy formulations.

On the top-level, the Sol language features only a single language component that represents the definition of a model in a very generic way. Such one-component approaches are frequent for experimental languages (e.g. [1]), since they typically result in a uniform structure that eases further processing. In addition, they yield to a clear and simple grammar.

The example above gives a first glance at Sol and enables us to take a closer look at the structure of a model-definition. A model-definition consists of three parts, where each of them is optional:

- The **header** section is essentially composed out of further definitions. These may be constants or further models. Definitions of the header-part can be publicly accessed and belong the model definition itself and not to one of its instances. In addition, the header enables you to state an extension of an existing definition.
- The **interface** section enables the modeler to declare the members of a model that can be publicly accessed. The members can either be basic variables or sub-models. Any of these members can be marked as a parameter that is passed at the model’s instantiation and remains constant for the object’s lifetime. Hence extra means for class-parameterization as in Modelica become obsolete.
- The **implementation** part contains then the actual relations between the variables and describes the dynamics of the system.

This general structure of a model-definition enables it to be used also for degenerated tasks like the definition of packages or connectors. A model in Sol represents a uniform approach that is similar to the class concept of Modelica. On the other hand side, the term "model" is almost overstressed and became so general that it lost some of its actual meaning. To regain expressiveness, Sol offers you different model-specifiers that enable the explicit denotation of certain sub-kinds. The usage of these specifiers involves consequently a number of restrictions. However, the syntax and semantics still remain uniform.

2.2 Object-Oriented Organization

The object-oriented and hierarchic organization of modeling code is substantially supported by two model-specifiers:

- **package**: Packages are used to collect models in a meaningful entity. A package-definition is reduced to the header part. It features neither an interface nor an implementation.
- **connector**: A connector typically collects members that are intended to be related by a connection-statement. A connector consists essentially of an interface. There's no implementation part.

The creation of an object-oriented hierarchy is illustrated in example 2 where the machine-model is split up into its principle components: An engine, a flywheel and additionally a simple gear model. These models use a uniform connector model and are based upon partial models that have been collected in an extra template package. Example 2 makes frequent use of the keyword `extends` that demonstrates the appliance of type-generation.

Example 2: Package structure in Sol

```
package MechTemplate

  package Interfaces

    connector Flange
    interface:
      static potential Real phi;
      static flow Real t;
    end Flange;

    partial model OneFlange
    interface:
      static Flange f;
    end OneFlange;

    partial model TwoFlanges
    interface:
      static Flange f1;
      static Flange f2;
    end TwoFlanges;

  end Interfaces;

end MechTemplate;


package Mechanics extends MechTemplate;

  model Engine1
    extends Interfaces.OneFlange;
  interface:
    parameter Real meanT;
  implementation:
      f.t = meanT;
  end Engine1;

  model Engine2
    extends Interfaces.OneFlange;
  interface:
    parameter Real meanT;
  implementation:
    static Real transm;
    transm = 1+cos(x = f.phi);
    f.t = meanT*transm;
  end Engine2;

  model FlyWheel
    extends Interfaces.OneFlange;
  interface:
    parameter Real inertia;
    static Real w;
  implementation:
    static Real z;
    f.phi = int(x=w);
    w = int(x=z);
    -f.t = z*inertia;
  end FlyWheel;

  model Gear
    extends Interfaces.TwoFlanges;
  interface:
    parameter Real ratio;
  implementation:
    ratio*f1.phi=f2.phi;
    -f1.t=ratio*f2.t;
  end Gear;

end Mechanics;
```

Sol offers three simple but effective mechanisms for type-generation. The most important of them is the type-extension better known as inheritance. Any model can extend any other model as long as there are no circular or recursive dependencies. Since packages represent models as well, inheritance can be applied to complete packages as well. The remaining two mechanisms consist in the redeclaration of members and the redefinition of models. Also these mechanisms can be applied to all feasible elements. In contrast to Modelica the redeclaration is used for type-generation only and not for class-parameterization.

Figure 1 depicts the resulting package structure of our example. The solid lines denote the memberships whereas the dotted arrows represent inheritance.

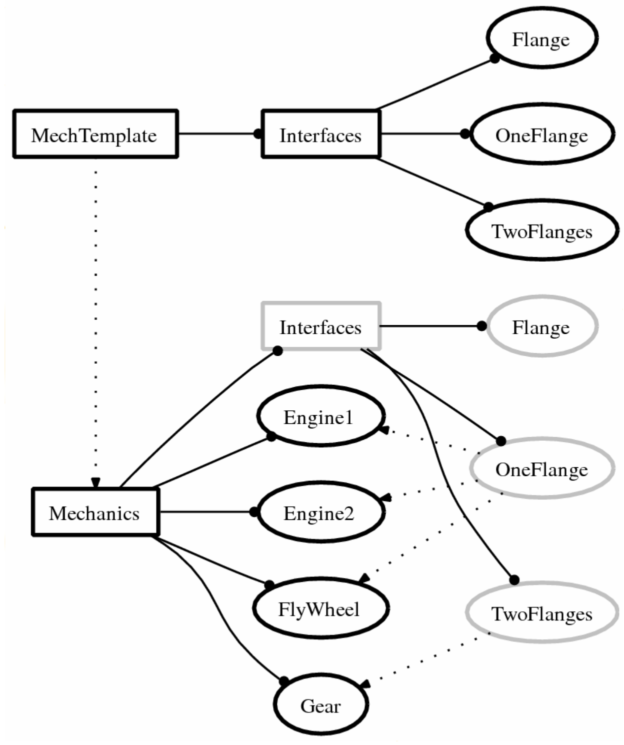


Figure 1: Exemplary package-hierarchy in Sol

Whereas the example has been over-elaborated for the purpose of demonstration, the combined usage of type-generation mechanisms forms a powerful tool for certain application domains like fluid-dynamics [3]. There, a package for a certain material may serve as a potential template. A modeler can then quickly adapt to other materials by a package-extension and a redefinition of the basic material model.

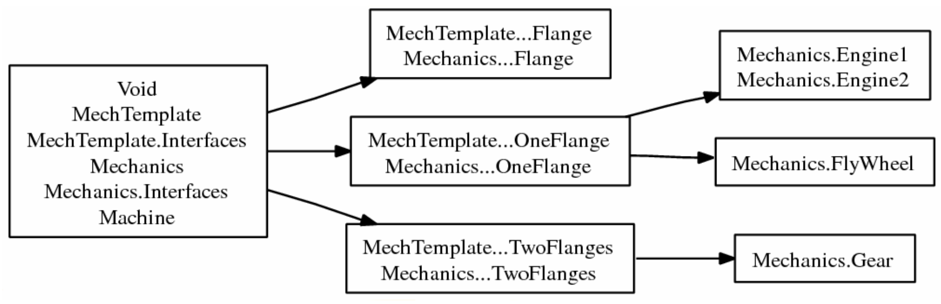


Figure 2: Exemplary type-hierarchy in Sol

2.3 Type-System

Like Modelica, Sol features a structural type-system [2]. It is solely based on the model's interfaces. The development of implementations and interfaces can therefore be separated and disjoined lines of implementation may yield into compatible types. The provided mechanisms of inheritance and redeclaration enable a satisfactory degree of polymorphism.

The type of a model is composed out of its members in the interface section. Any type-extension will yield to the creation of a sub-type of the inherited model. Also redeclarations and redefinition are limited to be only possible by sub-types of their original representation. Figure 2 illustrates the resulting type-structure of Example 2.

A proper and user-evident type-system becomes increasingly important in a dynamic framework like Sol. In situation where assignments are applied on complete sub-models to perform a model-exchange the corresponding assignments should be guarded by the type-rules.

2.4 Implementation part

The implementation part represents a block. A block may contain declarations of private members, relations (e.g. equations) or further nested conditional blocks in any arbitrary order. Let us analyze each component in more detail.

Private Declarations:

Declarations of private members do hardly differ from their counterparts in the header sections. Only the parameter attribute and the access-specifiers are now meaningless and therefore disabled.

The declaration of a member links a model instance to a given identifier. This linking is either static or dynamic. This selection has to be stated before the actual declaration. In contrast to a static linking, a dynamic linking enables to modeler to (re-)assign a new instance to the corresponding identifier.

Conditional Blocks:

Sol features if-else-branches and when-else-branches. The condition of an if-branch is immediately applied. It forms a safe condition that can be assumed to hold for its content. Hence the condition must be independent on any of its branches' content. When-statements are used to catch an event. The events are triggered during the update-procedure and are scheduled for the next one. Thus, when-conditions are not safe. Unlike Modelica, there are no syntactical restrictions on the content of the branches, but all branches shall finally lead to correct system of equations.

Statements:

Three fundamental operators are provided for setting up relations between members:

- The operator `=` states an equation between two expressions of type real.
- The causal copy-transmission `<<` is setting up causal relationships between real variables and can be used to link a copy of a model-instance to an identifier.
- The causal move-transmission `<-` is used to link a model-instance to a new identifier and to remove the former linking.

Member-access in statements:

To access the public members of your sub-models, three options are provided:

- As in Modelica the `.` operator is the most straightforward way of access, but not always convenient.
- The `connection(…)` statement exist also in Sol and has practically the same meaning as its counterpart in Modelica.
- The `( )` operator enables a function-like notation. It is especially suited for anonymously declared members.

Whereas the . operator represents a universal form of member access, the other two forms serve convenience and their proper appliance is determined by specifiers at the corresponding member-declarations. The connection statement only refers to variables that have been marked by the specifiers **`flow`** or **`potential`**. The specifiers **`in`** and **`out`** determine the applicability of the access by round-brackets.

3 Example Model

The presented language elements are sufficient for the formulation of highly variable systems. However, given the brief introduction above, it may not be evident how objects can be dynamically created, exchanged and deleted as there appears to be no explicit tool for these purposes. Let us therefore look at an example.

We reassemble the machine-model from example 1 that consists of an engine that drives a fly-wheel. This time we use the components of the `Mechanics` package in example 2. Furthermore we add a simple gear to our model. We recognize that the package provides two models for an engine: The first model `Engine1` applies a constant torque on the flange. In the second model `Engine2`, the torque is dependent on the positional state, roughly emulating a piston-engine. Both models share the same type (see figure 2). 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.

Example 3: Machine with a structural change

```
model Machine
implementation:
  static Mechanics.FlyWheel F{inertia<<1};
  static Mechanics.Gear G{ratio << 1.8};
  connection(G.f2,F.f);

  static Boolean fast;
  if fast then
    static Mechanics.Engine1 E{meanT<<10};
    connection(E.f,G.f1);
   else then
    static Mechanics.Engine2 E{meanT<<10};
    connection(E.f,G.f1);
  end;

  if initial then fast << false; end;
  when F.w > 40 then fast << true; end;
end Machine;
```

The resulting model is presented above. It includes two conditional branches, one for each mode. The current mode is stored in the Boolean variable `fast`. The corresponding transition is modeled by the when-statement.

3.1 Simulation Result

Using an interpreter program, the system was simulated for 10 seconds by the excessive number of 10'000 integration steps with the forward Euler method. The computational effort sums up to a total of 0.2 seconds on a standard CPU, where the effort for parsing and preprocessing is almost completely negligible. Figure 3 displays a plot of the angular velocity. The structural change reveals more clearly in the magnification. The actual change in the structure of equations is presented by the two causality-graphs of figure 5 and 6. Their closer examination is part of section 4.

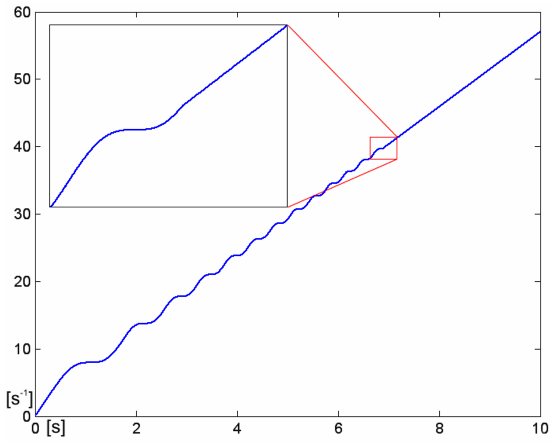


Figure 3: Angular velocity of the flywheel.

3.2 Alternative modeling approach

In the prior example, model-instances have been implicitly created and removed by the if-statement. Using local engine-models in the two branches is a very natural modeling approach, but often leads to redundant formulations (e.g. the connection statement) and therefore not all structural changes can be formulated in such a way. Thus, Sol enables the dynamic linking of an identifier to its instance. This offers a more convenient and general approach.

Let us model the machine for a second time, this time using a dynamic engine-model `E` that is initially linked to an `Engine2` model. At the transition-event, the `Engine1` model is dynamically created by an anonymous declaration. Since it is linked to the member `E` by a move-transmission, its lifetime exceeds the event and the newly created model replaces the former one. The replacement is valid because the types of the two engine models are equivalent.

Example 4: Alternative version of the machine-model

```
model Machine
implementation:
  static FlyWheel F{inertia<<1};
  static Mechanics.Gear G{ratio << 1.8};
  dynamic Engine2 E{meanT << 10};
  connection(E.f,G.f1);
  connection(G.f2,F.f);
  when F.w > 40 then
    E <- Engine1{meanT << 10};
  end;
end Machine;
```

The deletion of a model-instance is mostly done implicit by replacing the linking to an instance (as above) or by the removal of the corresponding identifier. However, example 5 presents the predefined `trash` object that is of type void and can be used for the explicit deletion of any object.

Example 5: Explicit deletion of a model-instance

```
trash <- E;
```

This mechanism for the dynamic linking of a model-instance represents a pointer-free modeling approach. The linking obeys clear ownership principles and therefore the simulation system can assure a memory-safe execution. Furthermore, the modeler is freed from the tedious and error-prone task of memory-management.

4 Processing Schemes

Sol is currently processed by an interpreter. The interpreter was named Solsim and represents a command-line program running under Linux or Windows. The input-file can be written in a standard text-editor. The simulation is performed and its output can be written into a file readable by the programs Matlab™ or Gnuplot. In addition to its main task, the interpreter provides also tools for the analysis of the model-hierarchy, type-structure and causalization mechanisms

Whereas the pair of a compiler and a simulator is the preferred choice for high-end simulation tasks, an interpreter is an appropriate tool (cf. [6]) for research work on language design. The development process becomes easier, faster and more flexible. Hence the development of the interpreter can proceed in parallel with a further refinement of the language. Furthermore, new debugging techniques become crucial in a more dynamic framework. This can be easier provided by an interpreter, since all necessary meta-information is available. Figure 4 displays a simplified overview of the main processing scheme that is composed out of six blocks. The following sections discuss these parts in more detail.

4.1 Parsing and Lexing

The Lexer processes the elementary elements of the language and discards all comments and formatting. Since the remaining part of the language forms an L1-Grammar, the actual parsing forms a rather trivial task. The parser is handwritten and features an automatic error-generation.

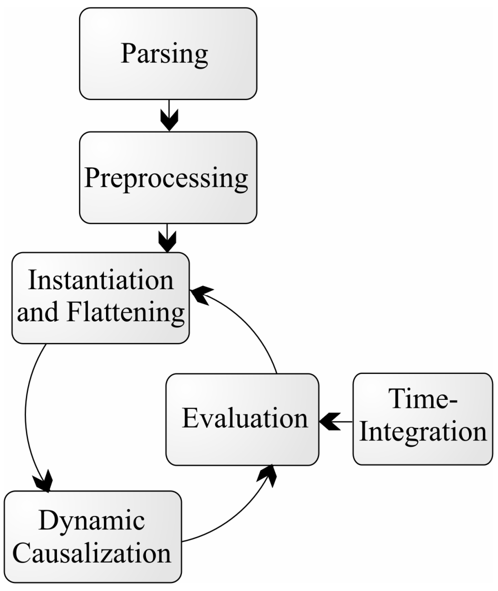


Figure 4: Processing scheme of Sol

4.2 Preprocessing

In the next stage, the mechanisms for type-generation are applied. This concerns primarily the resolving of type-identifiers and the appliance of the type-extensions. However these two processes cannot be implemented in a linear fashion. They usually have to be processed in several, interleaved steps.

Since a type-extension can be applied even on a complete package, the extension itself may generate new type-identifiers that may have to be resolved elsewhere. Thus, the algorithm has to "crawl" through the dependencies. Circular or recursive extensions lead to an inevitable downfall of this process and are therefore detected.

Furthermore the mechanisms for model-redefinition and member-redeclaration are processed. All methods for type-generation undergo a validation process, where consistency of the type-structure is checked. The resulting tree-structure of the package-hierarchy and of the type-system can be displayed by the interpreter. Please note that figure 1 and 2 represent graphs that have been automatically generated.

4.3 Instantiation and flattening

At the beginning, the top model is instantiated. The instantiation of a model evokes the following steps: First, all members (i.e. variables or sub-models) are instantiated recursively. Second all the statements in the implementation are processed.

The process of instantiation is aligned with the flattening of the system. Hence common statements like transmissions or equations are collected in a global set. The processing of an if-statements leads preliminary just to the instantiation of its corresponding condition. The actual content is instanced at a latter evaluation cycle.

In the dynamic framework of Sol the instantiation of models isn't restricted to the initial build up phase. Later instantiations will most likely occur. Consequently also their removal has to be managed. This is done in the exact reverse way.

4.4 Dynamic Causalization

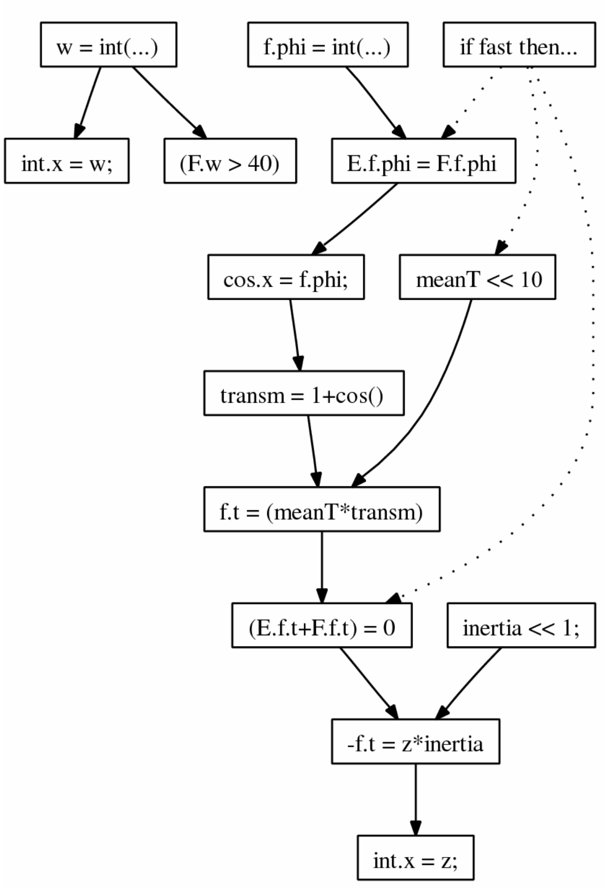
The result of the previous stage is a flattened model represented by a global set of equations and transmissions. The dynamic causalization analyzes this set of equations generates a data-structure that is suited for later evaluation cycles. The final target of this processing stage is depicted by the causality-graph in figure 5 and 6. There, the actual change in structure is revealed.

The resulting graph sketches the dependencies between the equations and transmissions. It includes also logical dependencies (dotted-lines) that result out of the conditional branches. This graph can then be further simplified by removing alias-variables or constant parts.

Any change in the set of equations will yield to an update of the causality-graph. The new equations need to be causalized and integrated into the graph. Furthermore the causality of previously causalized equations may now change. To handle all these cases in an efficient manner, the algorithm for the dynamic causalization is strongly optimistic. This means that it preserves existing structures, as long as possible, even if they temporarily loose their causal roots. Hence we can ensure that a small local change will not cause a global change unless the structure of the equation system makes this inevitable. For instance, the exchange of the engine model will not affect the causality of the fly-wheel or the gear model. Therefore the update considers only a sub-graph and can be treated locally. The details of this algorithm remain to be published.

4.5 Update and Evaluation

Based on the causality-graph, the system can be evaluated. This may consider the whole system or only a small subpart. Arbitrary updates can be triggered. If several updates are triggered at once, they are evaluated synchronously. The update procedure evaluates all dependent relations and successfully avoids any multiple evaluations of relations where separate update-paths meet.



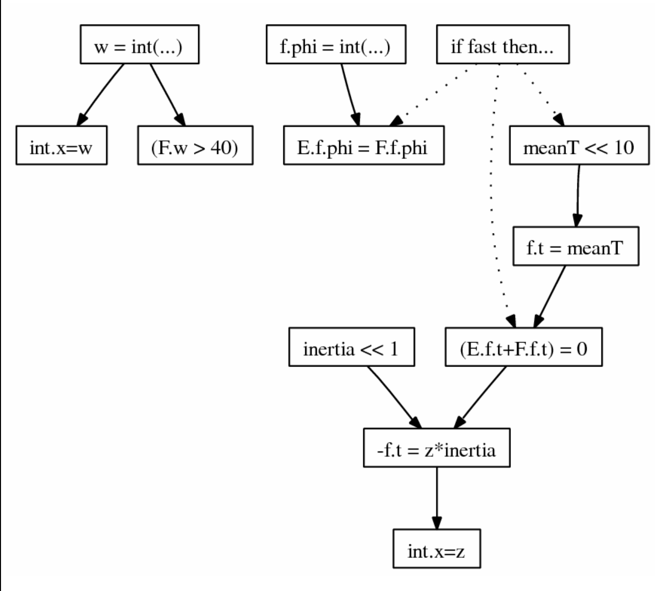


Figure 5 (left): Causality-graph of the machine at time 1 having "Engine2" as submodel.

Figure 6 (right): Causality-graph of the machine at time 8 using the simpler model "Engine1" as submodel.

Both graphs originate from an automatically generated version, where the gear-model has been omitted. The graphs have been slightly simplified to increase clarity and readability.

Logic-dependencies in the causality graph form silent dependencies. This means that an update of the corresponding Boolean expression does not directly trigger updates on its logical dependent equations. Silent dependencies are purposed only to ensure a correct update flow.

Furthermore, the causality graph contains also relations that own side-effects. Those relations may typically trigger an instantiation or removal of equations. The condition of an if-branch represents a prime example for this.

4.6 Time-Integration

The evaluation of the system (or a part of it) is triggered by two major sources. One is the insertion of new relations through instantiation. The other one, and much more frequent, is the time-integration of the corresponding state-variables. Currently, only simple explicit methods for integration are available. Since the system may reconfigure during an integration step, most integration algorithms with multiple steps cannot be implemented in a straightforward manner. It should be ensured that only the final step may trigger structural changes. Also certain methods for step-size control need to be adapted for the new framework.

5 Limitations and Efficiency

5.1 Current limitations

The current version of Solsim provides a framework for a more dynamic handling in equation-based modeling. The language itself enables the statement of drastic structural changes in a general way. Thus, the causalization of several equations may change in dependence of the structure. Also various submodels may be instantiated or removed on run-time leading to a variable number of instances.

However, there are severe restrictions that consider the type of equation systems that are currently supported. Solsim is yet unable to treat any equation system that contains algebraic loops. Also there is no index-reduction mechanism. And therefore the differential equations are temporarily formulated by the explicit statement of an integrator.

These restrictions reduce severely the applicability of the current system. In most practical situations, structural changes hardly lead to an isolated reconfiguration like a simple causality change. Often a complete set of tasks has to be accomplished at once [5]. This concerns, for instance, the dynamic handling of algebraic loops, a dynamic state-selection and mechanisms for index-reduction or robust, redundant re-initialization. In mechanics, the problem of multiple contact points with ideal-friction even yields to a complicated optimization task [9].

5.2 Efficiency

Whereas it is too early to give serious benchmark results, this section may at least give an impression about the current speed of our interpreter on a standard CPU. In general, we can state that the number of equations that can be evaluated per second is in the order of magnitude from 10^5 to 10^6. The mechanisms for instantiation, flattening and causalization manage altogether to handle between 10^4 and 10^5 equations per second.

Most important is that the efficiency is high enough to let us exceed the complexity of trivial models. Of course, the interpreter, like any other interpreter suffers from a certain computational overhead that will prevent its usage for highly demanding simulation applications.

Please note that the outlined processing scheme is not an exclusive solution. It is a very general approach and consequently represents overkill for many specific applications. However, a declarative language as Sol is very well suited to enable various optimization techniques, since the semantics do not directly stipulate the processing scheme. A number of optimizations may therefore be developed. For instance, a potential optimization is a run-time compiler. One might also try to include certain parts of the causalization, simplification and flattening into the preprocessing stage. Another interesting topic is the automatic identification and pre-compilation of situations where the system can be described by a finite set of sub-modes.

5.3 Future Tasks

Our primary target is to enhance the general applicability of our approach with respect to the set of DAE-systems that can be properly handled. Therefore we have a strong incentive to develop algorithms for the tearing of algebraic loops and index-reduction that are flexible and can be well integrated into our dynamic framework.

Furthermore the presentation of the core language omits a number of language elements that have still remained in the state of design. This concerns, for example, a general solution for collections of models (e.g. arrays).

6 Conclusions

The Sol language is built upon declarative principles and is strongly influenced by Modelica. It incorporates a general modeling methodology for variable-structure systems. The Sol research project offers a dynamic framework that enables the convenient acquaintance of knowledge in language design and processing techniques that we think will be essential for Modelica's future development.

Such a methodology benefits prevalent application areas and is likely to enlarge application field for equation-based modeling. To this end, future developments that concern primarily language design and processing techniques are required.

Appendix

The following listing of rules in extended Backus-Naur form (EBNF) presents the core grammar of the Sol modeling language. The rules are listed 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 quotes. Rules may wrap over several lines.

Common fundamental expressions like the model for the mathematical function **sin()** or given global variables as **time** or **initial** form predefined elements within the language itself and are therefore not part of the grammar. The same holds for the fundamental types in Modelica. These are: **Real**, **Integer**, **Boolean**, **String** and **Void**.

Listing 1: EBNF-Grammar of Sol

Model	=	**ModelSpec Id Header**
		[**Interface**] [**Implemen**] end **Id** ";"
ModelSpec	=	[redefine] [partial]
		(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   =  [Condition | Event | Declaration | Relation]

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"
```

Acknowledgments

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] Bläser, L.: A Component Language for Structured Parallel Programming. In: *Joint Modular Languages Conference,* Oxford, UK (2006) 230-250.

[2] Broman, D., Fritzson, P., Furic, S.: Types in the Modelica Language. In: Proceedings of the Fifth International Modelica Conference, Vienna, Austria (2006) Vol. 1, 303-315.

[3] Casella, F., et al.: The Modelica Fluid and Media Library […]. In: *Proc.eedings of the Fifth International Modelica Conference*, Vienna, Austria (2006) Vol. 2, 631-640.

[4] Cellier, F.E., Krebs, M.: Analysis and Simulation of Variable Structure Systems Using Bond Graphs and Inline Integration. In: *Proc. ICBGM'07, 8th SCS Intl. Conf. on Bond Graph Modeling and Simulation*, San Diego, CA (2007) 29-34.

[5] Enge, O.: *Analyse und Synthese elektromechanischer Systeme,* Ph.D. Dissertation, TU Chemnitz, Germany (2006).

[6] Mosterman, P.J.: HYBRSIM - A Modeling and Simulation Environment for Hybrid Bond Graphs, In: *J. Systems and Control Engineering*, 216, Part I (2002) 35-46.

[7] Nilsson, H., Peterson, J., Hudak, P.: Functional Hybrid Modeling from an Object-Oriented Perspective In: *Proc. of the 1st Intern. Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany (2007) 71-87.

[8] Nytsch-Geusen, C., et al.: Advanced modeling and simulation techniques in MOSILAB: A system development case study. In: *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria (2006) Vol. 1, 63-71.

[9] Pfeiffer, F., Glocker, C.: *Multibody Dynamics with Unilateral Contacts*. John Wiley & Sons, New York (1996).

[10] Zauner, G., Leitner, D., Breitenecker, F.: Modeling Structural-Dynamics Systems in Modelica […] Mosilab and AnyLogic. In: *Proc. of the 1st Intern. Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany (2007) 71-87.

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

Biography

Dirk Zimmer received his MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He gained additional experience in Modelica and in the field of modeling mechanical systems during an internship at the German Aerospace Center DLR 2005. Dirk Zimmer is currently pursuing a PhD degree with a dissertation related to computer simulation and modeling under the guidance of Profs. François E. Cellier and Walter Gander. His current research interests focus on the simulation and modeling of physical systems with a dynamically changing structure.