An Application of Sol on Variable-Structure Systems with Higher Index

Dirk Zimmer
Department of Computer Science, ETH Zurich
CH-8092 Zurich, Switzerland
dzimmer@inf.ethz.ch

Abstract

This case study presents the model of an ideal trebuchet. Following the object-oriented modeling paradigm of Modelica, the trebuchet is composed out of ideal elements that belong to a planar mechanical library. The corresponding system of DAEs has index 3. During simulation, the model undergoes also various structural changes that manipulate the number of continuous-time state variables. Furthermore, elastic and inelastic collisions need to be modeled by force impulses. The model is provided in Sol, a derivative language of Modelica, specially designed for research in variables structure systems. *Keywords: Variable-Structure Systems; Index Reduction; Multi-Body Dynamics.*

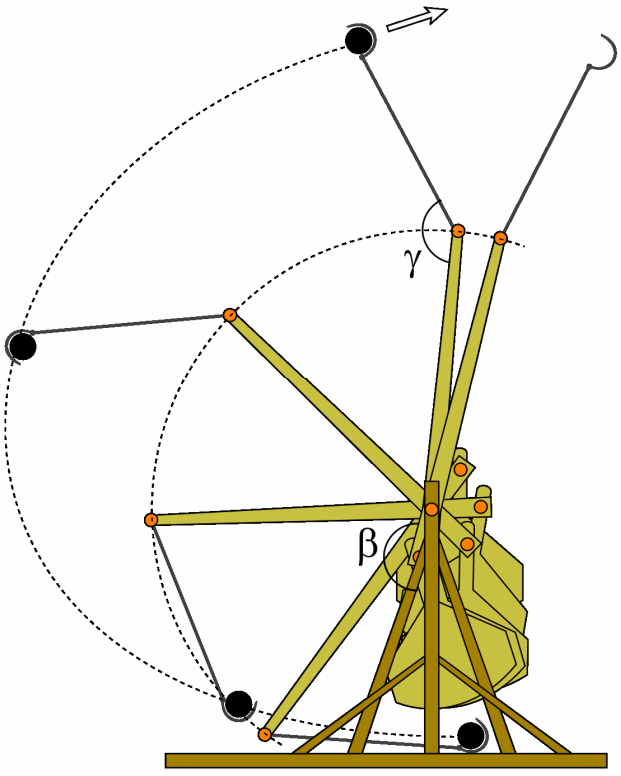
1 The Trebuchet

The Trebuchet is an old catapult weapon developed in the Middle Ages. It is known for its long range and its high precision. Figure 1 depicts a trebuchet and thereby presents its functionality. Technically, it is a double pendulum propelling a projectile in a sling. The rope of the sling is released on a predetermined angle γ when the projectile is about to overtake the lever arm.

Let us state a few assumptions for the model:

- All mechanics are planar. The positional states of any object are therefore restricted to x, y and the orientation angle φ.
- All elements are rigid.
- The sling's rope is ideal and weightless. It exhibits an inelastic impulse when being stretched to maximum length
- The revolute joint of the counterweight is limited to a certain angle β (in order to prevent too heavy back-swinging after the projectile's release). It also exhibits an inelastic impulse when reaching its limit.
- Air resistance or friction is neglected.

Whereas these idealizations simplify the parameterization of the model to a great extent, they pose serious difficulties for a general simulation environment. Such models, although being fairly simple, can neither be modeled nor simulated with Modelica yet. At least not in a truly object-oriented manner. Hence the trebuchet represents a suitable example for the framework of Sol that aims to enable the future handling of variable-structure systems within an object-oriented modeling paradigm.



source: wikimedia commons, modified by author

Mass of projectile: **30kg**	β:**200°**
Mass at lever arm: **100kg**	γ: **150°**
Counterweight: **10t**	

Figure 1: Functionality and specification of a trebuchet

2 Object-oriented Composition

Sol has been introduced at the Modelica Conference 2008 [9, 10]. It is a derivative language of Modelica, specially designed for research purposes in the field of variable-structure systems. Thus, Sol enables the creation and removal of equations or even complete objects anytime during the simulation. 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 runtime. The incentive for this project is to gain knowledge in language design and processing techniques that we think will be essential for Modelica's future development.

A simple planar mechanical library has been developed in Sol. It has been extended by equations for mechanical impulses in order to make discrete velocity changes possible. From this library we need the following components:

- 1x fixation
- 3x fixed translation
- 1x revolute joint
- 1x limited revolute joint
- 2x bodies with mass
- 1x ideal rope with mass

These components are connected as depicted in figure 2. Although the model diagram follows the iconographic of the MultiBody library [4], it serves illustration purposes only, since the modeling in Sol is still purely textual.

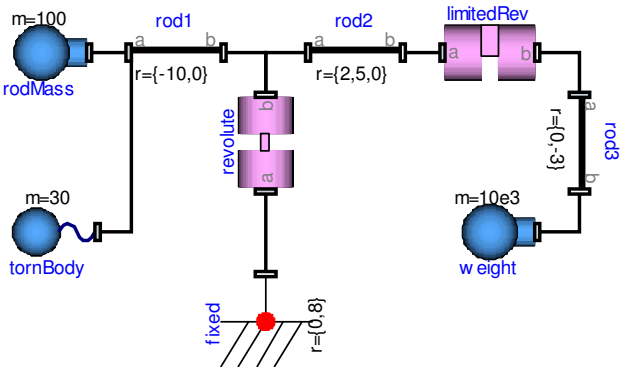


Figure 2: Model diagram of the trebuchet

The total model contains from 246 to 256 variables, depending on the current state of the model. The corresponding systems of DAE have the perturbation index 3. They need to be differentiated twice and there remain linear systems of equation to be solved.

The resulting object-oriented decomposition resembles typical examples from the Modelica domain but it is significantly more demanding since a structural change in any component may affect the total system.

```
model LimitedRevolute
  extends Interfaces.TwoFrames;
interface:
  parameter Real phi_start;
  parameter Real w_start;
  parameter Real l;
implementation:
  static Boolean contact;
  static Boolean fixated;
  static Boolean toFixate;
  static Boolean toRelease;
  static Real phi_a;
  static Real phi;
  static Real Wm;
  static Real We;

  if initial then
    fixated << false;
    toFixate << false;
    toRelease << false;
    phi_a << phi_start;
    We << w_start;
  end;

  when toFixate then
    toRelease << false;
    fixated << true;
  else when toRelease then
    toFixate << false;
    fixated << false;
  end;

  if fixated then
     phi = l;
     Wm = 0;
     contact << false;
     when fb.t < 0 then
        toRelease << true;
        phi_a << l;
     end;
  else then
    contact << (phi > l);
    static Real w;
    static Real Wa;
    w = der(x=phi, start << phi_a);
    when contact then
      w = 0;
      Wm = 0.5*Wa;
      We << w;
      toFixate << true;
    else then
       when fa.contactIn or fb.contactIn then
         w  = 2*Wm - Wa;
         We << w;
       else then
         static Real z;
         z  = der(x=w, start << We);
         Wa << w;
       end;
       fb.M = 0;
    end;
    fb.t = 0;
  end;

  fa.phi + phi = fb.phi;
  fa.t + fb.t = 0;
  fa.Wm + Wm = fb.Wm;
  fa.M + fb.M = 0;

  fa.x = fb.x;          fa.y = fb.y;
  fa.fx + fb.fx = 0;    fa.fy + fb.fy = 0;
  fa.Vmx = fb.Vmx;      fa.Vmy = fb.Vmy;
  fa.Px + fb.Px = 0;    fa.Py + fb.Py = 0;

  fa.contactOut << contact or fb.contactIn;
  fb.contactOut << contact or fa.contactIn;
end LimitedRevolute;
```

Figure 3: The model of a limited revolute joint.

3 Example component

Whereas the top-model can be neatly decomposed into general applicable components, the modeling of these components requires a skilled modeler. To attain a better understanding, let us take a look at the modeling code of one of the components that triggers a structural change: The limited revolute joint. The corresponding code is presented in figure 3.

Since Sol is very similar to Modelica the code shall be roughly understandable without further introduction. Let us go into the details.

An elbow is one possible representation of a limited revolute joint. The model has two major modes: *free* or *fixated*. The mode *free* is equivalent to a normal revolute joint whereas the model equals a fixed orientation in the *fixated* mode. Since the transition between these two states causes a discrete change in velocity, it involves an inelastic impulse on the rigidly connected components. Furthermore impulses from other components (as for instance the ideal rope) need to be handled as well in this component.

The different modes and their transitions are presented in the graph of figure 4, where the continuous-time modes are depicted as round boxes and the rectangular boxes denote discrete intermediate modes. The transitions are represented by arrows and their labels denote the event that triggers the transition. Those without a label are triggered immediately.

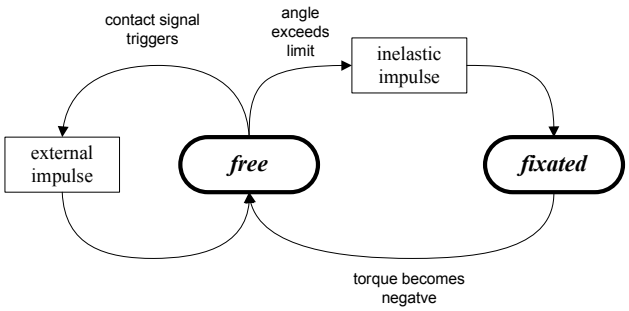


Figure 4: Mode-transition graph of the limited revolute

In the modeling code, the two continuous modes are expressed by the Boolean variable `fixated` and are modeled by an if-statement. We recognize that the first branch represents the fixated mode and does not contain any derivatives whereas the second branch (for the free mode) usually defines two derivatives. Hence, the free mode defines two potential state-variables: the position `phi` and the corresponding velocity `w`. A switch between the two modes is therefore expected to change the number of total state-variables.

The number of continuous-time state variables is also affected by the mechanical impulses. These impulse events are modeled by when-branches that react to a contact signal that may be emitted by other components. In order to understand how the model interacts with other components let us take a look at variables of the connecting interface:

Continuous potential variables:

`x y phi`: the positional states:

Continuous flow variables:

`fx fy t`: forces and torque

Discrete potential variables:

`Vmx Vmy Wm`: mean velocities during impulse.

Discrete flow variables

`Px Py M`: force impulses and angular momentum.

Control signals:

`contactIn`: ingoing contact signal

`contactOut`: outgoing contact signal

This connector design is very similar to the one that has already been applied in the Modelica MultiBondLib [11]. It owns a separate set of variables for the continuous and discrete domain. The Boolean control signals are used to trigger and synchronize the events.

Any component model will have to relate these interface variables. For the limited revolute, the equations that relate the variables of the translational domain are trivial and are placed at the end of the model's main section.

Nevertheless, the equations for the impulse event require further explanation. A force impulse `P`, or angular momentum `M` respectively, causes a discrete change in the corresponding velocity. This change is best described by the mean velocity during the impulse. Let `Wa` be the angular velocity before the impulse and `We` the velocity after the impulse, then `Wm` is defined as the mean `(Wa+We)/2`. Please note that the product of the corresponding interface variables (e.g. `M*Wm`) represents the amount of work that is transmitted during the impulse.

Using these variables, the impulse behavior can be properly described: For any mass element, the equation

```
M = 2*I*(Wm-Wa)
```

holds. An inelastic impulse can be modeled by stating:

```
Wm = 0.5*Wa
```

Mostly and also in this example, these and other impulse equations form a linear system of equations that is distributed over several components. Hence

they need to be activated synchronously. To this end, the Boolean contact signals are required to synchronize the impulse events in different components.

This is illustrated by the event for an external impulse (see figure 5). Before the event the velocity is stored by the auxiliary variable `Wa`. At the event, the differential equation is removed since the velocity is now determined by the impulse equation `w = 2*Wm - Wa`. This new velocity is also stored in the auxiliary variable `We` that is needed after the event when the differential equation gets reestablished and `We` is suggested as (re-)start value for the time integration.

```
when fa.contactIn or fb.contactIn then
  w  = 2*Wm - Wa;
  We << w; [1]
else then
  static Real z;
  z  = der(x=w, start << We);
  Wa << w;
end;
```

Figure 5: Excerpt from figure 3.

In the example of the trebuchet, this event is synchronously triggered with corresponding events from all body components and the other revolute joint. During the contact event, the number of continuous-time states is reduced.

The transition from the free mode to the fixated mode by the inelastic impulse is modeled in a similar manner. The trigger for this transition is a contact signal that becomes true when the angle `phi` exceeds the parameterized limit `l`. The contact signal is transmitted to the connected components in order to synchronize the following event. At this event, the continuous-time equations are replaced by the equations for an inelastic impulse and the variable `toFixate` is set to true. This causes a subsequent event that changes the continuous-time mode.

The reverse transition is modeled in accordance, but here a force impulse is not required. The established fixation is released when the torque acts in the opposite direction: `t < 0`.

4 Further components

Let us put aside the model of the limited revolute. From the remaining 8 components of the trebuchet model, there are two more components that exhibit structural changes. These are the standard revolute joint and the torn body.

The standard revolute joint is significantly simpler than its limited counterpart. It does not own multiple modes for the continuous-time simulation. Just an intermediate mode is required for the impulse handling. This influences the number of continuous-time state variables during the impulse. Typically the angular velocity of the revolute joint represents a state variable but during the impulse it is discretely determined.

The component for the torn body is more interesting. It owns 3 continuous-time modes with different continuous-time state variables:

1. The body is at rest as long as the rope has not been stretched.

   State-Variables: { }

2. The body represents a pendulum as long as the release angle γ has not been reached.

   State-Variables: { φ, ω }

3. The body is free.

   State-Variables: {x, y, φ, v_x, v_y, ω}

Furthermore, the transition between mode 1 and 2 has to be modeled by an inelastic impulse acting in rope direction. Another intermediate mode is required for the handling of external impulses. Figure 6 represents the corresponding transition diagram.

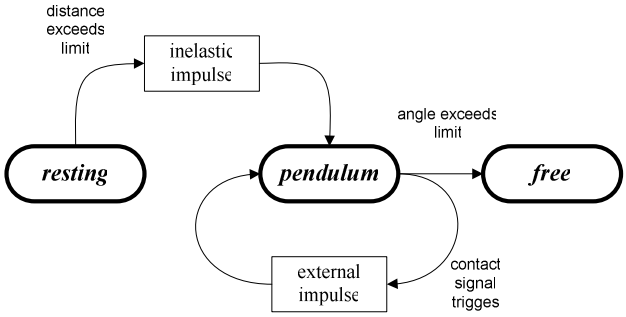


Figure 6: Mode-transitions graph of the torn body

In this way, the modeling of structural changes has been distributed on 3 of the 9 components. The object-oriented paradigm favors such a distribution. The modeling on the local level is not only easier to achieve than a complete description of the system, but also the resulting components represents meaningful entities by themselves and become usable in a generic fashion.

The modes of the total system, the trebuchet, result from the combination of its component's local modes during the simulation of the system. To get a better understanding, let us look at the simulation of the trebuchet.

[1] The symbol << represents a casual transmission - a statement that is similar to an assignment. Once applied, the variable on the left-hand side retains its value and remains determined until it gets re-determined by another causal transmission.

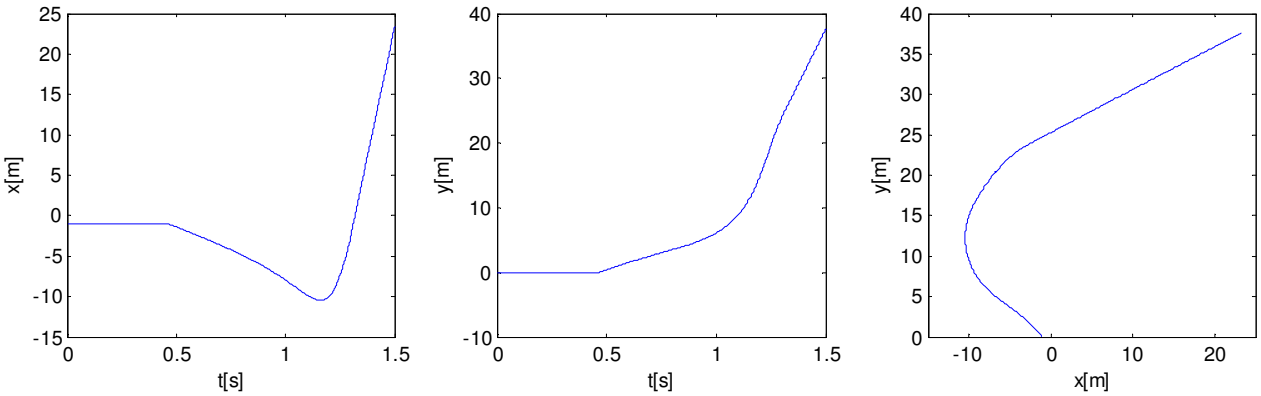


Figure 7: Trajectory of the projectile.

5 Simulation

Figure 7 presents the result of the simulation for the first 1.5 seconds. The model was simulated with Solsim, a console application that represents an interpreter of the Sol language. The main processing loop of the interpreter contains 3 stages:

- Instantiation and flattening
- Dynamic causalization
- Evaluation

In a classic Modelica translator these stages are executed once in sequential order. In Solsim, they form a loop (see also figure 9) and hence all these three stages can be repeated several times. Thus, the interpreter is able to handle almost arbitrary structural changes. All these stages are thereby programmed in way that they try to preserve the existing structure and prevent unnecessary perturbations. Further explanations can be found in [9] and in section 6.

In contrast to the elaborate processing techniques, the numerical algorithms that are included in Solsim are still on a very rudimentary level. Thus, an explicit Euler integration has been applied with a fixed step size of 1ms. In spite of this tiny step size and consequently the high number of iterations, Solsim was still able to parse, to setup and to simulate the whole system roughly within one second on common personal computer.

Let us take a look at the various structural changes that occur during the first two seconds of the simulation. Figure 8 presents an overview for the continuous-time modes and their corresponding state variables. The diagram presents the continuous-time modes for the torn body and the limited revolute with their corresponding state variables.

The transition between these modes may involve force impulses that require an intermediate mode. Such intermediate modes are depicted by a vertical line in the diagram.

The combination of modes of the components forms the modes of the complete system. In total there occur 5 modes where only 2 of them are equivalent. Furthermore, there are 2 intermediate modes for the inelastic impulses.

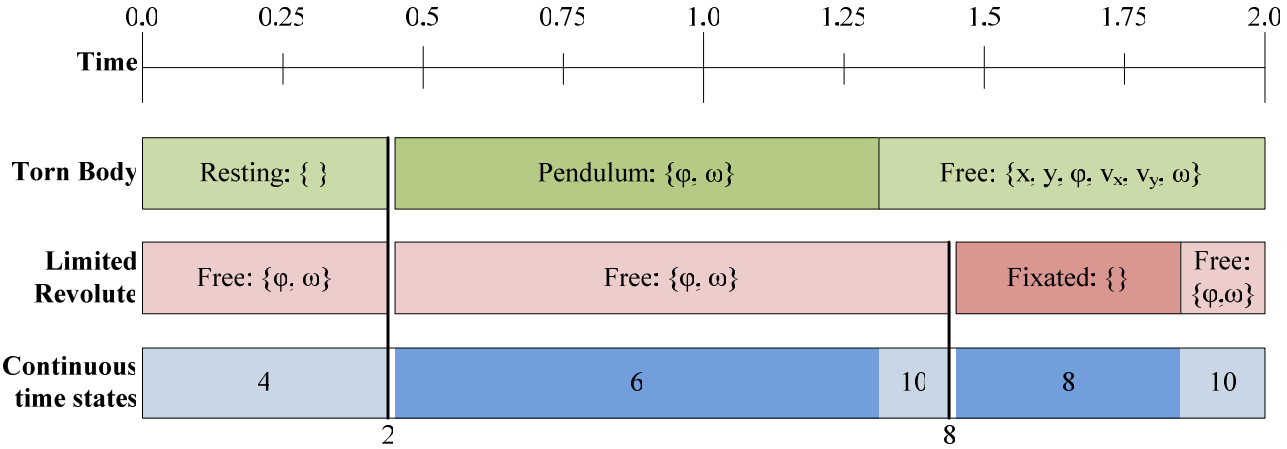


Figure 8: Timetable of the structural changes

In addition to the state variables that are listed in figure 8, there are two more state variables, namely the angle and angular velocity of the non-limited revolute joint. This holds with exception of the intermediate modes. Here, the velocities are disabled as state variables. Hence the number of continuous-time state variables in total varies from 2 to 10.

We recognize that the handling of these structural changes is a demanding problem. It contains a number of sub-tasks that need to be implemented by the simulation environment. Let us therefore review the principal processing steps and how they are affected by the variability in structure. We then continue with the integration of these tasks in the dynamic framework of Sol.

5.1 Event handling

Structural changes represent discrete events. The modeling of mechanical impulses requires that such events can be synchronized. On the other hand, the simulation environment must enable that several discrete events can be scheduled in a sequential order without any time advancement.

For this purpose, Solsim has implemented an event heap that is independent from the time integration. Time can only advance if the event heap is empty for the current time-frame. Preceding valuable contribution in this area are [2] and [5].

5.2 Sate Selection

The selection of feasible state variables is crucial for the time-integration of mechanical systems. Like Modelica, Sol also offers an option to prioritize potential state variables. The modeler can indicate preferred states and thereby support the simulation environment in its selection.

With respect to variable-structure systems, such a mechanism is especially important since a complete a priori analysis of the system might not be available or affordable in a dynamic framework.

5.3 Index reduction

In order to reduce the differential index of the DAE-system, symbolic differentiation has to be applied. Which parts that have to be differentiated depends on the current structure of the system. For instance, some equations of the torn body require differentiation while being in mode 2. After the transition to mode 3, no differentiations for this component are required anymore. Thus, Solsim keeps track of the required derivatives during the simulation.

The standard procedure for index reduction is known as Pantelides [6] algorithm. This algorithm presumes all potential state variables to be known and differentiates the occurring constraint equations.

This procedure is unfortunately inadequate for variable-structure systems. Therefore a different approach is implemented in Solsim: State variables are assumed a priori as unknown and the subsequent state selection is then integrated in the standard causalization procedure.

5.4 Tearing

For computational reasons, a transformation of the system into block-lower-triangular (BLT) form is aspired. The Dulmage-Mendelson permutation [7] is the most well known algorithm for this task, whose central part is the strong component analysis of the Tarjan algorithm [8]. This step identifies the blocks of the BLT. In a subsequent step, tearing variables may be chosen for the blocks that enable the application of iterative solvers.

Such a multi-step algorithm is not suited for a dynamic framework as Sol. Hence Solsim applies the tearing directly on the complete system and identifies the resulting blocks by the corresponding residuals. The block decomposition is therefore not necessarily optimal but mostly still adequate.

Simple heuristics are applied for the selection of tearing variables. Furthermore the modeler has the option to indicate suitable choices for tearing. For solving the corresponding equation system, Solsim applies a simple iterative solver.

6 Dynamic DAE Processing

Figure 9 presents the main processing scheme of the Solsim interpreter. Its centerpiece is the loop that consists in instantiation, dynamic causalization and evaluation. The evaluation of the system can be triggered by the algorithm for time integration or by the event handler. The evaluation of certain statements (e.g. an if-statement), may then involve the creation or removal of certain components and their corresponding equations. These changes need then to be dynamically handled by the processor for differential-algebraic equations.

Essentially, it is this dynamic DAE-processor (DDP) that defines Solsim's capabilities and enables the proper and efficient handling of even severe structural changes. The DDP takes the changes in the set of equations as input and generates a causality graph as output.

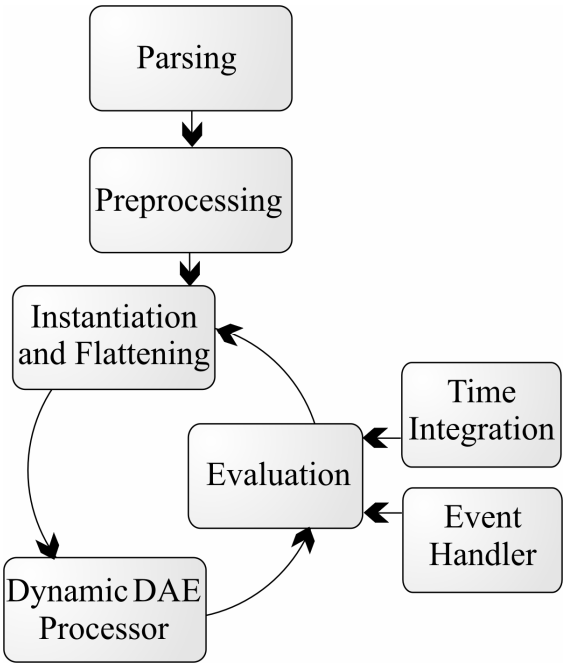


Figure 9: Processing scheme of Sol

The causality graph $G(E,V)$ is a directed acyclic graph where the vertices V correspond to the equations. The edges E are formed by those pairs of equations (s_1, s_2) where v is a variable of s_2 and determined by s_1.

Since the causality graph is an acyclic graph, it gives rise to a partial order on its vertices and can thus be used to schedule the set of causalized equations into an appropriate order for evaluation. The causality graph thereby enables the complete or partial update of a system and brings the system in a form that is suitable for numerical ODE solvers.

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. In the worst case, the exchange of a single equation will require the update of the whole system. Most changes, however, only affect parts of the system. In order to handle all these cases in an efficient manner, the dynamic DAE processor is strongly optimistic and tries to preserve the existing graph structure as much as possible.

The DDP essentially represents a set of update rules and graph-algorithms that trace each change in the set of equations and keep track of the current causalization. This has a profound impact on the handling of the tasks that have been outlined in section 5.2 to 5.4 (state selection, tearing, and differentiation).

Furthermore, the reverse counterparts of these tasks must be concerned too. The determination of a tearing variable can become obsolete and the tearing needs to be undone. The situation is similar for variables that have been selected as state variables. Also the time-derivative of a variable may not be required anymore if a change in set of equations occurs and shall therefore be eliminated.

In the DDP, the handling of all these tasks is not pursued by individual algorithms anymore. Instead, the corresponding processes are formulated as a closely interlinked set of update and downdate rules. This results in a rather complicated processing mechanism that is concerned with a good number of details. Unfortunately, this prevents any simple presentation of the DDP's functionality and hence it goes beyond the extent of this paper. For this reason, we aspire a journal publication in multiple parts and hope to publish it soon.

We can, however, outline the major principle of the DDP. In the first place, the DDP retains the causality graph as much as possible. To this end, equations remain potentially causalized, even if they lost their 'causal root'.

For any new equation, the DDP attempts its integration into the existing causality graph. This may lead to premature or speculative causalizations. In consequence, residuals may yield from overdetermined equations.

Whenever a residual is generated, their correspondent sources of overdetermination are examined. Based on this analysis, an appropriate action is taken in order to eliminate the overdetermination. This action is distinct from case to case. It can represent the undoing of former causalizations or state selections but also the extraction of an algebraic loop. In this way, the DDP enables the treatment of DAEs that result from variable-structure systems in an efficient manner.

7 Conclusions

The current framework of Sol represents a feasible solution for the modeling and simulation of variable-structure systems, although being rudimentary in many aspects. The example of the trebuchet demonstrates that the object-oriented modeling paradigm of Modelica can be successfully extended to variable-structure systems of higher index. The modeling of certain subparts can be quite demanding but the resulting components are fairly generic in their usage.

The Sol language by itself is even simpler than Modelica and hence major additions to the Modelica language would not be required (like state charts as in [3]). The power and expressiveness of Sol originates from the generalizations of successful Modelica concepts and not from the introduction of new paradigms.

These generalizations though, require new methods for the processing of such a model. This is a challenging task that demands new solutions for many major stages in the classic processing scheme. The simulator Solsim meets these requirements now to a sufficient extent.

Since Solsim is an interpreter it represents computational overkill for many specific applications and thus cannot be applied yet for computationally very demanding applications. Instead, it represents a truly general framework that can be applied to a broad range of models from various domains. We think that this approach is more promising in the long term, since specializations can still be implemented when necessary.

For instance, there is a sub-class of Sol models that is decomposable into a reasonably constrained number of modes. The trebuchet belongs to this sub-class. For such models, code corresponding to each mode can be compiled in advance and then be executed. There would be no principal problem in detecting members of this sub-class and alter the translation accordingly. For other cases, a just-in-time compilation may be desired. Corresponding solutions are meanwhile developed in the framework of Hydra [1].

Both the language Sol and the corresponding software Solsim need further extensions, refinement and optimization. But most of our future effort is planned for the completion of the whole framework. The Sol project shall be made openly accessible in a well-documented state. We thereby hope to establish a promising field for future research that lets us and other researchers elaborate new modeling and processing techniques.

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] Giorgidze, G., H. Nilsson: Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In: *Proc. 7th International Modelica Conference*, Como, Italy (2009)

[2] Nikoukhah, R., S. Furic: Synchronous and asynchronous events in Modelica: proposal for an improved hybrid model. In: *Proc. 6th International Modelica Conference* (2008) Bielefeld, Germany, Vol.2, 677-690.

[3] 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.

[4] Otter, M., H. Elmqvist and S.E. Mattsson: The New Modelica MultiBody Library. In: *Proc. 3rd International Modelica Conference*, Linköping, Sweden (2003), 311-330.

[5] Otter, M., H. Elmqvist and S.E. Mattsson: Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle. In: *Proc. IEEE International Symposium on Computer Aided Control System Design*, (1999) Hawaii, 151-157.

[6] Pantelides, C.: The Consistent Initialization of Differential-Algebraic Systems. In: *SIAM J. Sci. and Stat. Comput*. (1988) Vol 9, No. 2, 213-231.

[7] Pothen, A., Chin-Ju Fan: Computing the Block Triangular Form of a Sparse Matrix. In: *ACM Transactions on Mathematical Software* (1990) Vol 16, No. 4 303-324.

[8] Tarjan, R.: Depth-first search and linear graph algorithms. In: *SIAM Journal on Computing*. (1972) Bd. 1, No. 2, 146-160.

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

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

[11] Zimmer, D. and F.E. Cellier: The Modelica Multi-bond Graph Library, *Proc. 5th International Modelica Conference*, Vienna, Austria (2006) Vol.2, 559-568.

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.