A Compositional Semantics for Modelica-style Variable-structure Modeling

P. Pepper[1] A. Mehlhase[2] Ch. Höger[3] L. Scholz[4]

[1]Institut für Softwaretechnik, TU Berlin, Germany, {peter.pepper}@tu-berlin.de
[2]Institut für Softwaretechnik, TU Berlin, Germany, {a.mehlhase}@tu-berlin.de
[3]Institut für Softwaretechnik, TU Berlin, Germany, {christoph.hoeger}@tu-berlin.de
[4]Institut für Mathematik, TU Berlin, Germany, {lscholz}@math.tu-berlin.de

Abstract

Modelica traditionally has a non-compositional semantic definition, based on so-called "flattening". But in the realm of programming languages and theoretical computer science it is by now an accepted principle that semantics should be given in a compositional way. Such a semantics is given in this paper for Modelica-style languages. Moreover, the approach is also used to consider more general modeling concepts, namely so-called variable-structure systems. As an outlook we discuss the correspondence between such an idealized mathematical semantics and a more pragmatic numeric solver-oriented semantics.

Keywords Modelica, compositional semantics, structure dynamics, uncertainty.

1. Introduction

A large class of technical multi-physics systems can be modeled in languages like Matlab/Simulink/Stateflow, Ascet, Labview, Scicos, Modelica and various others. These systems are essentially based on the paradigms of control theory and differential-algebraic equations (DAEs). Of these systems, Modelica is distinguished by the fact that it integrates the control theoretical aspects with the software engineering principles of object-oriented programming. We consider this a major advantage and therefore concentrate in this paper on Modelica-style modeling.

However, there are a number of modeling concepts that are also missing from Modelica, most notably the so-called *structure dynamics*, also referred to as *variable-structure modeling*. Roughly speaking this means that during its lifetime the system passes through *modes*, in which it obeys different sets of (differential) equations. This paradigm is very helpful in the "natural" specification of many systems, but unfortunately it adds a number of severe conceptual, semantic and implementation problems to classical Modelica. Nevertheless, it has already been – at least partly – addressed in a few systems, notably Mosilab [21] or SOL [29]. Structure dynamics is also possible in languages like Hydra [24, 20, 7, 19] or frameworks like CIF (Compositional Interchange Format) [25, 26, 27]. However, we do not consider such languages or frameworks here, since they are not Modelica-like languages; rather Hydra is a Haskell-based functional language that achieves similar effects with other means (see below); the same is true for MKL [3, 4]. And CIF is a framework that shall integrate all kinds of languages and language paradigms.

We note in passing that variable-structure systems can also be treated in tools like Simulink and Dymola, but only efficiently by way of using scripting techniques and only for systems consisting of a few different modes [16].

In this paper we consider the core principles of variable-structure modeling and try to give a rigorous semantics for them. This semantic treatment bases on ideas taken from approaches such as Phase Transition Systems [14, 15] and Hybrid Automata [11, 8], the latter of which in turn are based on StateCharts [10]. Moreover, we also integrate ideas taken from ESpec [22, 12] as well as from Lustre [5], Ptolemy [23] , CIF and comparable systems. However, since all these approaches and concepts are overlapping, we will not make any attempt to attribute each of our individual design decisions to their respective predecessors.

When comparing formal approaches such as Phase Transition Systems to Modelica-style systems, one can immediately see a fundamental difference in attitude, which makes any semantic comparison almost vain from the beginning: Whereas the former bases on clean mathematical concepts of the real numbers $\mathbb{R}$ and their function space, the latter immediately focuses on numeric solvers and all problems that are coming with them. In order to bridge that gap, we envision a two-level semantics:

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

1. *Ideal semantics*. The first semantics that we give lives in the ideal mathematical world of real numbers, continuous functions, dense time and the like.
2. *Simulation semantics*. The second semantics is derived from the first one by taking issues such as discretization, approximation, rounding errors and the like into account. However, we should emphasize that this is still work in progress.

Through this separation of concerns we do not mix up different and unrelated kinds of aspects in a single monolithic description.

In the realm of programming languages and theoretical computer science it is by now a well established principle that the semantics of languages should be given in a compositional way. This means that the global semantics of the whole can be described by giving local semantics to the parts, from which the overall semantics is derived by suitable composition rules. (This is in accordance with the principle of modularization in software and systems engineering.)

$$\begin{tikzcd}
\text{system} \arrow[r, "\textit{semantic}", "\textit{interpretation}"'] \arrow[d, "\textit{flattening}"'] & \text{components} \arrow[d, "\textit{composition}"] \\
\text{flat system} \arrow[r, "\textit{semantic}", "\textit{interpretation}"'] & \text{system model}
\end{tikzcd}$$

Figure 1. Compositional semantics

Figure 1 illustrates the relationship between compositional and non-compositional semantic definitions. The standard definition of Modelica follows the lower left path: One always considers the system as a whole and flattens it into one large system of equations, which represents its semantics. However, in the realm of programming languages one rather gives individual semantic definitions for all components and then composes the overall semantics from these local definitions. This is the approach, which we will follow in this paper.

Note: In the literature one can also find attempts to give semantics to a continuous modeling language by designing it as an embedded DSL (domain-specific language) over some existing host language. For example, Hydra [20, 7, 6] uses Haskell as the encompassing host language. Also the approaches taken in CIF, which in turn bases on earlier work on Chi, or Ptolemy [23] can be subsumed under this principle. Here the host for the embedding is not a real programming language such as Haskell; rather one employs a powerful mathematical framework (such as special automata with SOS-based semantic definition) into which the modeling language is mapped.

We do not follow this idea here, since it adds considerable complexity to the understandability of the semantics. One first has to fully comprehend the host language – which is not trivial for a language like Haskell or CIF – and then one has to understand the embedding. This may be acceptable for computer scientists, who want to enter the field of continuous modeling, but it is unacceptable for engineers, who are familiar with areas like control theory and possibly can program in C or Fortran. For this audience, Haskell poses an insurmountable obstacle in practice. Therefore we prefer to give an independent semantics to modeling languages such as Modelica, which strives for simplicity and easy understandability.

Part I: Ideal Semantics

2. Preliminaries: Specifications and Models

In the realm of programming languages and their semantics there is by now a well-established way of proceeding. To begin with, one has to clarify what kinds of terms are syntactically allowed. To this end we follow the principles that have been laid down very precisely in the realm of algebraic specifications. (For a comprehensive treatment see [1]; we follow here mainly the approach taken in the Espec system [22, 12].) But we have to adapt the pertinent concepts to the new situation of differential-algebraic equations.

The starting point is a *signature* Σ that determines the admitted types and operations. In our case this signature essentially consists of the standard operations of real arithmetic. (At least we focus on that part of the signature.) As a special feature the signature contains the differential operator $\frac{dx(t)}{dt}$, which we usually denote as $\dot{x}$ or x'.

Over such a signature Σ and a set X of (typed) variables one can then build the set $T(\Sigma; X)$ of well-typed *terms*. And over these terms one can then build *equations*. Due to the availability of the differential operator, these equations are actually so-called *differential-algebraic equations*, short: DAEs.

Such a signature Σ together with a set E of equations over Σ is called a *specification* (in formal logics sometimes also *theory presentation*). As is well-known from algebra and formal logics, such a specification in general has many *models*, where the notion *model* refers to a mathematical structure (usually an algebra) that is conformant to the signature and obeys the equations.[1]

Definition 2.1 (Specification, model) *Let $S = (\Sigma, E)$ be a specification. The set of* ***models*** *of this specification is denoted as Mod*$(S) = \{ A \mid A \models S \}$. $\Diamond$

Note that we include in the notation $A \models S$ not only the validity of the equations but also the conformance to the signature. If the signature is clear from the context, we also write $A \models E$.

3. Fixed-structure Systems

We first consider fixed-structure systems. This kind of system corresponds essentially to the style of models that can

[1] It is unfortunate that the word "model" occurs here in two conflicting meanings. On the one hand, there is the terminology of "continuous-system modeling" and on the other hand there is the formal-logic terminology of "models" of theories. We hope that the two usages will always be clear from the context.

be formulated in Modelica. These models are given as a hierarchy of subsystems that are ultimately based on atomic components. Since these subsystems are usually encapsulated into some suitable context, they look like components themselves.

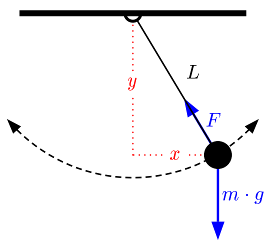
Definition 3.1 *A **system** is built up from interacting components. These **components** are in turn either* subsystems *or* atomic components. ♢

Modelica follows the approach of object-oriented languages and describes components and systems by way of *classes*. Such a class can be seen as a "blueprint", from which at runtime concrete components ("objects") are obtained by instantiation. In the terminology of algebra (see Section 2) these classes correspond to the specifications and the components to models (in the sense of formal logic). In the following we will often just speak of "components" or "systems". It will be clear from the context, whether we are talking about the classes or about the instances.

We present our semantic definition in a bottom-up fashion. That is, we start from atomic components and then compose them into larger systems.

3.1 Atomic Components

The most elementary kind of system is an *atomic component*. As is illustrated by the example in Figure 2 such an atomic component consists of parameters and constants, variables and equations. Parameters and constants are an important and convenient feature in practical applications, but they do not play an important role in the semantics. Therefore we will ignore them from now on and only concentrate on the variables.



Class $Pendulum$	
PARAM $Length\ L$	parameters
PARAM $Mass\ m$	
CONST $Acceleration\ g$	
$Length\ x$	variables V
$Length\ y$	
$Force\ f$	
$m \cdot \ddot{x} = -\frac{x}{L} \cdot f$	equations E
$m \cdot \ddot{y} = -m \cdot g - \frac{y}{L} \cdot f$	
$x^2 + y^2 = L^2$	

Figure 2. An atomic component

In general components (more precisely, the classes describing them) are embedded into the context of some encompassing system. According to the usual scoping principles of programming and modeling languages this means that the component has access to the variables of this context. We refer to these variables as *external* variables of the component. These variables are usually not mentioned explicitly in the component but determined implicitly by the scoping rules. In any case we need to split the set of variables into the two sets of "external" and "local" variables.

Definition 3.2 *An **atomic-component class** is a named triple $C = (V_e, V_l, E)$ consisting of two disjoint sets V_e and V_l of* external and local variables *and a set E of* hybrid differential-algebraic equations (DAEs). ♢

As mentioned in Section 2 such a class in general represents a whole set of possible models. (This is often referred to as "underspecification".) That is, there are many components that fulfill the equations of the class.

In our application of continuous-system modeling the variables of the class are interpreted as time-dependent functions. For example, in Figure 2 the variable x stands for a function $x(t)$ that gives the x-position of the pendulum at any given point in time t.

Definition 3.3 *Let $C = (V_e, V_l, E)$ be an atomic-component class. A **model** for this class is a structure $M = (F_e, F_l)$ consisting of two sets of time-dependent functions, which are in one-to-one correspondence to the sets V_e and V_l of variables. These functions have to fulfill the equations in E. We denote this as usual by $M \models C$. By Mod(C) we denote the* set of all models *of the class C.* ♢

Note that the functions are completely general, that is, they may be continuous or discontinuous or even discrete, that is, only defined at selected time points. In this way the set of equations E may also contain equations that determine the initial values.

In practice the one-to-one correspondence between the variables and the functions is simply established by using the same names for both, that is, e.g. x for the variable and $x(t)$ for the function. However, as we will see in a moment, there may be many instances (objects) of the same class coexisting in one system. Therefore we have many different functions that correspond to the same variable, namely one for each instance. In order to resolve these conflicts systematically we assign unique identifiers to the instances of the class (just like in Java every object of a class is identified by a unique "reference"). The function names are then annotated by these identifiers.

For example, if we have two components K_1 and K_2 as instances of a class C, and if x is a variable in C, then the two instances have functions $x_{K_1}(t)$ and $x_{K_2}(t)$. In this way an equation like $\dot{x} = 2 \cdot x$ of the class is interpreted by the two model equalities $\dot{x}_{K_1}(t) = 2 \cdot x_{K_1}(t)$ and $\dot{x}_{K_2}(t) = 2 \cdot x_{K_2}(t)$.

3.2 Composition of Components

Systems are usually built by composition of atomic components. The most elementary form of such a composition is sketched in Figure 3. We have two classes C_1 and C_2 describing components with individual variable sets V_1, V_2

and equation sets E_1, E_2. Since they coexist in the same context, they share the set V_e of external variables. As mentioned earlier, these external variables are usually not declared in the component but derived implicitly by the scoping rules.

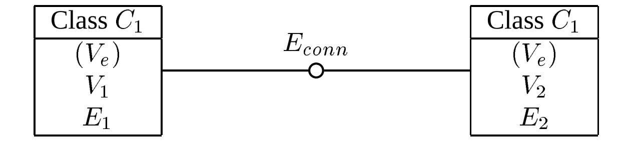


Figure 3. Composition of components

The components are linked to each other by way of so-called *connectors*. From a semantic point of view such connectors only contribute further equations, typically of the form

$$\begin{array}{rcll} C_1.e & = & C_2.e & e\text{: potential/effort} \\ C_1.f + C_2.f & = & 0 & f\text{: flow} \end{array}$$

For the semantics this means that we have to combine all models of the two classes, provided that they coincide on their common external part. This set then needs to be filtered further by the connection equations.

Definition 3.4 (Composition) *Let a system S be given, which consists of the two classes $C_1 = (V_e, V_1, E_1)$ and $C_2 = (V_e, V_2, E_2)$ (sharing the same external variables V_e) together with a set E_{conn} of* connection equations. *Then the models of the* ***combined system*** *are derived from the individual models as follows:*

$$\begin{array}{rcll} \mathit{Mod}(S) & = & (\mathit{Mod}(C_1) \otimes \mathit{Mod}(C_2)) \,|\, E_{conn} & \\ & \overset{def}{=} & \{\, M_1 \cup M_2 \,| & M_1 \in \mathit{Mod}(C_1), \\ & & & M_2 \in \mathit{Mod}(C_2), \\ & & & M_1|_{V_e} = M_2|_{V_e}, \\ & & & M_1 \cup M_2 \models E_{conn} \,\} \end{array}$$

$\Diamond$

Since we assume that the local variables are suffixed by the component names, we can form the union of the models without any danger of name clashes. (The operator $\otimes$ is a pushout in the sense of category theory; that is, we form the direct product of the two sets of models while sharing their common global parts.)

Based on this definition of the semantic meaning we also write the composition of two components in the shorthand form $(C_1 \otimes C_2) \,|\, E_{conn}$.

This construction generalizes from two to n connected components in the obvious way.

3.3 Subsystems

By connecting n components we obtain subsystems. But in general such a subsystem is embedded into some encompassing component, which contributes both variables and equations. This is illustrated in Figure 4.

Note that this construction is usually embedded into an even larger context. Therefore we have the sets of variables V_e, V_S, V_1, $\ldots$, V_n and the equations E_S, E_1, $\ldots$, E_n

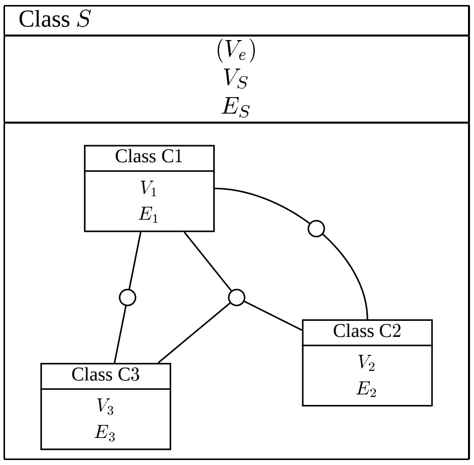


Figure 4. A subsystem

together with the various connection equations $E_{conn_1}, \ldots,$ E_{conn_k}. Note also that the subcomponents C_1, C_2 and C_3 have $V_e \cup V_S$ as their set of external variables.

Since the equations in E_S may refer to variables of the components by using prefix notations such as $\dot{x} = f(x, C_1.y, C_2.z)$, we have to respect this in the semantic definition. That is, a class reference such as $C_1.y$ refers on the model level to $y_{K_1}(t)$, where K_1 is the model instance of C_1.

Definition 3.5 *Let a system be given as described in Figure 4. Then the semantics is given by*

$$\mathit{Mod}(S) = \big(\mathit{Mod}(C_1) \otimes \cdots \otimes \mathit{Mod}(C_n)\big) \,|\, (E_S \cup E_{conn_1} \cup \cdots \cup E_{conn_k})$$

Note that $V_e \cup V_S$ is part of the global variables of each of the C_i such that these variables are taken care of by the operator $\otimes$. $\Diamond$

The construction can be applied iteratively to nested hierarchies of systems. Thus we obtain a bottom-up compositional semantics for Modelica-like languages. This semantics is equivalent to the monolithic flattening-based semantics as it has been sketched in the commuting diagram of Figure 1 in the introduction.

4. Variable-structure Systems

We want to go beyond classical Modelica and also consider variable-structure systems. Such variable-structure systems generalize the effects that can be achieved in Modelica with a combination of the if- and when-constructs or in Matlab/Simulink with the enabling blocks. Vice versa, these constructs can be semantically explained in terms of variable-structure systems. Non-standard Modelica systems such as Mosilab [21] or experimental designs such as SOL [29] already provide principal implementations of this idea.

In the fixed-structure systems discussed so far we only need to consider a single time interval, namely the *life span* of (the simulation of) the system. All components are created at the beginning of this life span and the functions $x(t)$ are defined (and simulated) over this life span.

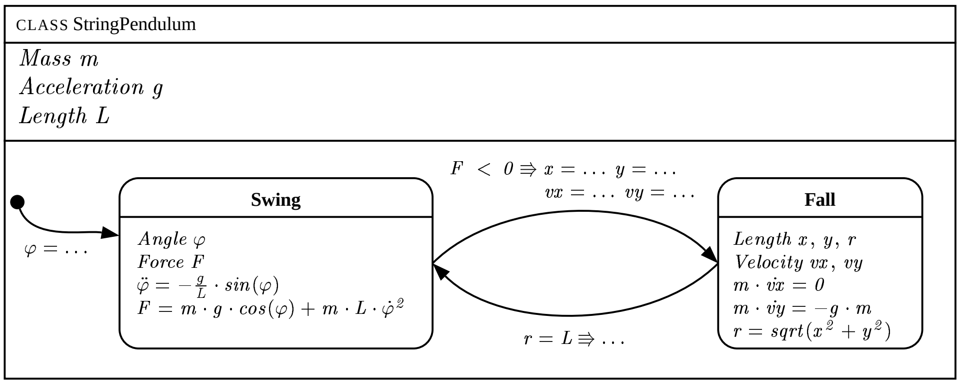

CLASS StringPendulum
$Mass\ m$
$Acceleration\ g$
$Length\ L$
Swing
$Angle\ \varphi$
$Force\ F$
$\ddot{\varphi} = -\frac{g}{L} \cdot sin(\varphi)$
$F = m \cdot g \cdot cos(\varphi) + m \cdot L \cdot \dot{\varphi}^2$
Fall
$Length\ x,\ y,\ r$
$Velocity\ vx,\ vy$
$m \cdot \dot{vx} = 0$
$m \cdot \dot{vy} = -g \cdot m$
$r = sqrt(x^2 + y^2)$
$\varphi = \dots$
$F < 0 \Rrightarrow x = \dots\ y = \dots$
$vx = \dots\ vy = \dots$
$r = L \Rrightarrow \dots$

Figure 5. A simple dynamic component

However, it will frequently be the case that a component passes through different modes, in which it exhibits different kinds of behaviors. An example is given in Figure 6 and its specification in Figure 5. Here we consider a string pendulum, which initially starts as a normal pendulum but changes to a free-falling ball as soon as the force F is less than zero.

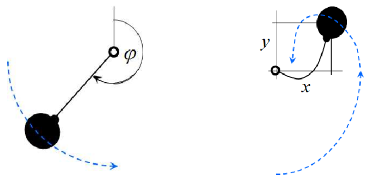


Figure 6. A simple dynamic component

For specifying such variable-structure systems and components we use a notation that is oriented at so-called *Hybrid Automata* [11] which in turn can be seen as a combination of *State Charts* [10] and *Phase Transition Systems* [14, 15]. As can be seen in Figure 5 such a component still has component-external variables and equations. (Actually, in this simple example there are only parameters and constants.) But now it also has modes with additional mode-local variables and equations. Moreover, there are guarded transitions between the modes, which also possess actions that essentially describe, how the initial values of the next mode are to be determined.

We will not indulge further into this slight generalization, since we want to consider even more general settings as described – again by an oversimplified example – in Figure 7. The specification of the two modes is sketched in Figure 8.

Here our initial system consists of two components, a car and a ball. The car exhibits the simple physics of a rolling device on an inclined plane, while the ball only contributes its mass. After hitting the block, there is only one interesting component left, namely the ball, which now follows the physical laws of a bouncing ball.

As can be seen in this example, each mode can contain whole subsystems such that the overall topology of the system under consideration may change dynamically.

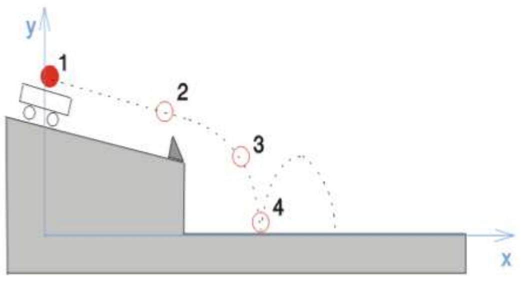


Figure 7. A simple mode-changing system

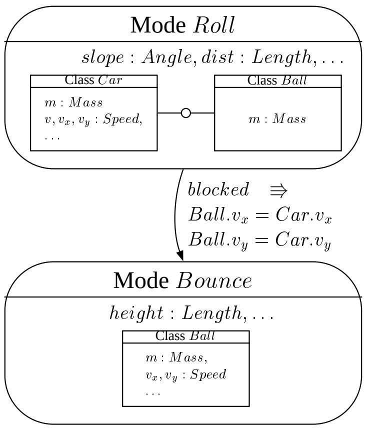


Figure 8. A simple changing topology

This principle is illustrated more abstractly in Figure 9, which shows the general setting.

A system has external variables V, equations E and components $C_1, C_2, C_3, \dots$. Moreover, it possesses modes $M_1, M_2, \dots$, which in turn have local variables, equations and subsystems. In order not to overload the pictorial illustration we have refrained in Figure 9 from drawing the connectors between the mode-local subsystems S_{M_i} and the component-subsystem S. Of course, such connectors are possible (and they are easily described in textual form).

This hierarchical structuring can be iterated over arbitrarily many levels.

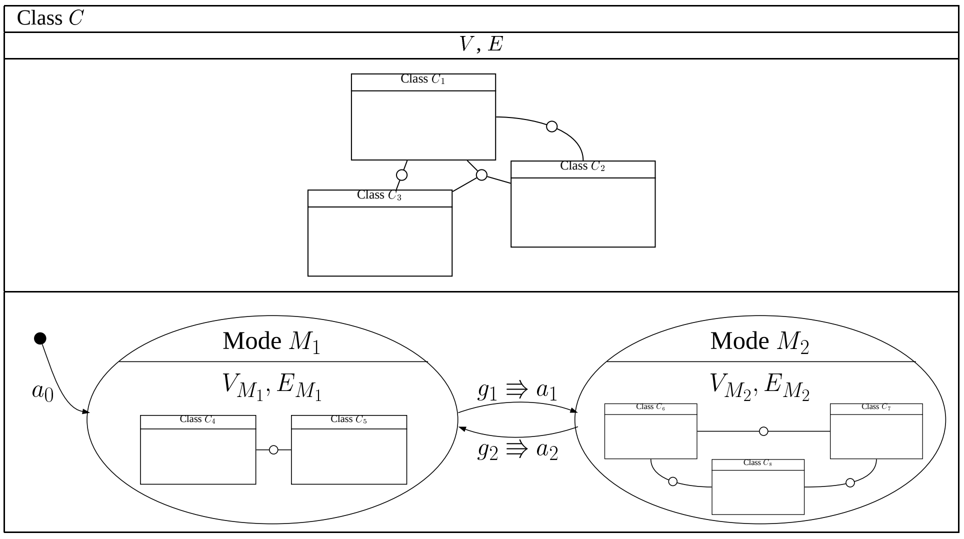


Figure 9. Variable-structure systems

Definition 4.1 (Variable-structure component) *A **variable-structure component** is a named tuple $C = (V, E, S, D)$, where V and E are sets of component-external variables and equations, S is a subsystem, that is, a set of interconnected subcomponents, and D is the **dynamics**, that is, a set of modes and transitions. (The modes and transitions will be defined more rigorously in the next sections.) Each mode $M \in D$ in general comprises mode-local variables V_M and equations E_M as well as a mode-local subsystem S_M of components. The dynamics has an **entry point** t^α and an **exit point** t^ω.* ◊

In the remainder of this paper we will make this informal definition more precise by giving rigorous semantics to its various features.

5. Modes and Time

The situation in a real-world system can be roughly sketched as in Figure 10. The modes (more precisely: instances of modes) follow each other along the time line. The component-external variables correspond to functions that live across the modes; an example is illustrated in Figure 10 by the thick line $f(t)$. The mode-local variables correspond to functions that live only during their mode; examples are given in the figure by the thin lines $g(t)$ and $h(t)$.

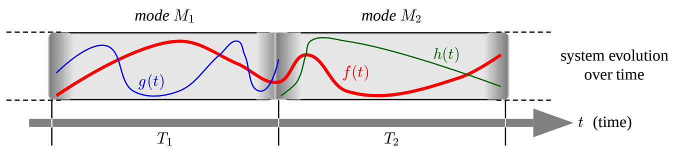


Figure 10. Real-world semantics

Note that we do not put any additional constraints on the functions; that is, within its realm each of the functions may be continuous, possess discontinuities or may even be discrete. In other words, we consider modes to be a *design concept*, not a technical feature that shall handle complications such as discontinuities.

Each mode – more precisely: each *instance* M_i of a mode – is associated to a unique interval $T_i = [t_i^\alpha, t_i^\omega)$ of the time line. For reasons that will be discussed in a moment we use half-open time intervals.

Note: In practical technical systems the transition between modes usually takes some time, in which the system is non-observable. A typical point in case is a diode, for which the switching from *on* to *off* is a very short but continuous process. In the idealized mathematical treatment such transition periods are often modeled as instantaneous transitions (which makes them often discontinuous). It is debatable, whether one might even allow whole transition intervals instead of transition points, or whether one should model such situations by transition modes. Both variations would be mathematically feasible; but for reasons of simplicity we opted for the design with left-closed intervals.

As a final preparatory remark we point out that a fixed-structure system can be considered as a borderline case of a variable-structure system with exactly one mode that spans the whole life time of the system.

Definition 5.1 (Modes) *Let a class $C = (V, E, S, D)$ be given as sketched in Figure 9. The instantiation of this specification (at runtime) leads to the creation of a new component K, which is a model of C. This component has a lifetime $T_K = [t^\alpha, t^\omega)$. The start time t^α is the moment of the creation of the component, the end time t^ω is determined by rules that will be discussed later (e.g. caused by events). We require $t^\alpha < t^\omega$, i.e. we exclude components with zero life time.*

The fixed-structure part $C_{fix} = (V, E, C_1, \ldots C_n)$ of the class is semantically defined as in Definition 3.5. Note that the subcomponents also have the life span T_K.

*The semantics of the system during a **mode** M is derived from the fixed-part semantics by composing it with the mode-local system S_M, that is, $C_{fix} \otimes S_M \mid E_{conn}$ as specified in Definition 3.4. Here E_{conn} are the additional connection equations that link the subsystem of the mode to the fixed-part subsystem.*

The life span of the mode M is determined by rules that will be explained in connection with transitions in the next section. $\Diamond$

Note that this semantic definition entails a subtle aspect. The fixed part $C_{fix} = (V, E, C_1, \ldots C_n)$ of the class in general has a whole set *Mod*(C_{fix}) of models. Since we allow the individual modes to impose further restrictions on the fixed-part variables in $V, V_1, \ldots, V_n$, the behavior of the corresponding functions varies from mode to mode.

Example: Let $x \in V$ be a component variable (without a restricting equation) and let x_1 and x_2 be two local variables of the modes M_1 and M_2, respectively. Let the following equations be given:

$$\begin{array}{llll} M_1: & x_1 = 1 & M_2: & x_2 = 2 \\ & x = x_1 & & x = x_2 \end{array}$$

Then $x(t)$ is a function, which is constant but different in M_1 and M_2 and has a discontinuous jump between the two modes.

6. Transitions and Events

Next we consider the actual mode transitions as illustrated in the examples of Figure 6 and Figure 7 or more schematically in Figure 9. The principles of such a mode transition are illustrated in Figure 11.

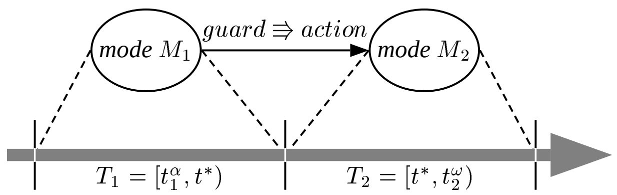

mode M_1
$guard \Rrightarrow action$
mode M_2
$T_1 = [t_1^\alpha, t^*)$
$T_2 = [t^*, t_2^\omega)$

Figure 11. Semantics of transitions

During the execution of the model each mode instance is associated to a time interval. In the situation of Figure 11 we have the two intervals $T_1 = [t_1^\alpha, t^*)$ and $T_2 = [t^*, t_2^\omega)$. The start point of T_1 and the end point of T_2 are of no interest here. The focus of our attention is the point t^*, which represents the transition between (the instances of) the two modes.

Definition 6.1 (Transition point) *Let two modes M_1 and M_2 with a transition $(guard \Rrightarrow action)$ be given as sketched in Figure 11. During runtime the transition point t^* between (instances of) these two modes is defined as follows: t^* is the smallest time instance greater than the entry point t_1^α of mode M_1 such that $guard(t^*) = true$ and $guard(\tau) = false$ for all τ with $t_1^\alpha < \tau < t^*$. Then, $T_1 = [t_1^\alpha, t^*)$ and the entry point of mode M_2 is given by $t_2^\alpha = t^*$.* $\Diamond$

Since modes shall not degenerate to zero length, we have the condition $t_1^\alpha < t^* < t_2^\omega$.

[18] provides a nice discussion of many aspects of such transitions from the point of view of physics. The design with half-open intervals ensures a well-defined solution $x(t)$ for all times t. Nevertheless, there are cases where the left-closedness may have to be relaxed (see also the discussion in [18]). However, these are special cases that certainly have to be dealt with by the solvers, but not an issue of ideal semantics.

Remark 1: The case of the immediate re-firing in the case of self-loops can be repaired by introducing the concept of *superdense time* [15]. (A typical example for such a self-loop is provided by the "bouncing ball".) Here the time is not only $\mathbb{R}$ but the product $\mathbb{R} \times \mathbb{N}$, where the second component could be interpreted as the number of the "clock" that is responsible for the event. Then each occurrence of the event $y = 0$ in the bouncing-ball example could be associated to another clock. Even though this idea of superdense time may turn out helpful in some situations, we currently refrain from introducing it and try to work with our simpler model.

Therefore we would rather consider the immediate refiring as a modeling error (analogously to infinite while loops in programming). For instance, in the bouncing-ball example the guard should not only be $y = 0$ but $y = 0 \wedge vy < 0$. The transition action (see below) has to set $vy_{new} = -vy_{old}$. When the velocity is zero, the mode should be exited.

Remark 2: There is the possibility of *conflicting guards*. That is, two guards g_1 and g_2 may fire at the same instant t^*. There are several options for treating this situation. One could let the system choose nondeterministically between the transitions. One could avoid this nondeterminism by enforcing priorities between the transitions (like in Matlab/Simulink/Stateflow). Or one could require the disjointness of the guards and report an error, when two of them fire simultaneously. This is a topic for language design. Since each of these options is compatible with our semantic concepts, we can ignore the issue here. We note in passing that there are attempts to utilize the idea of superdense time also for this issue.

So far we have only considered the guards that fire the transitions and thus determine the transition point t^*. We still need to define the meaning of the *action* part of the transition. The purpose of these actions is to provide the *initial values* for the next mode. There are various syntactic means for achieving this effect.

Example: Consider again the bouncing-ball example. When the ball hits the ground, the vertical velocity vy is reversed and deminished by some constant factor c. This could be written in a Pascal-like programming style as $vy := -c * vy$. Or we could use an equation-style notation like $vy_{new} = -c * vy_{old}$, which would necessitate notations for "old" and "new" (which is implicitly contained in the program-style notation by the occurrence of vy on the left or right side of the assignment). Again, the notation is an issue of language design and thus does not concern us here.

But we need to give a semantic meaning to x_{old} and x_{new} for all variables x occurring in M_1 and/or M_2.

Definition 6.2 (Transition action) *Consider the scenario of Definition 6.1. The* ***action*** *part of the transition describes a computation that uses (implicitly or explicitly) values x_{old} and x_{new} for certain variables x. Depending on the application, we can define $x_{old} = x(t^*)$ or*

$x_{old} = \lim_{h\to 0} x(t^* - h)$. *$x_{new}$ is computed as a function of x_{old}. Then $x = x_{new}$ is used as the initial-value equation for the differential-algebraic system in mode M_2.* $\Diamond$

This completes the compositional definition of the "ideal" semantics of Modelica-style variable-structure systems.

Part II Simulation Semantics

7. Numerical Integration and Solvers

The ideal semantics lives in the realm of the real numbers $\mathbb{R}$ with infinite precision and infinitely accurate computations. But in reality we have to make do with the limitations of computers, where one struggles with floating-point arithmetic and its rounding errors, with solver techniques realized in packages like DASSL [2] or RADAU5 [9], with causalization and index reduction and so forth. This leads to a different kind of semantics that we baptize "solver semantics" or "simulation semantics".

The reason for dealing with this question is the current situation in modeling languages such as Matlab/Simulink or Modelica. The specification documents of these languages often switch back and forth between concepts that we refer to as "ideal semantics" and the difficulties introduced by the computer-related limitations. Hence it is not always clear, which aspects describe fundamental semantic concepts and which aspects are actually due to shortcomings of certain solver methods. Sometimes one even gets the impression that certain statements actually address specific features of certain compilers or compilation techniques.

7.1 Solver Issues

Solver semantics differs from the ideal semantics primarily in two respects: discretization and finite precision.

1. *Discretization*. Functions over the continuous time domain $\mathbb{R}$ are replaced by functions over discrete time points t_i, where $t_{i+1} = t_i + h_i$ for some discretization step size h_i, starting at the initial time point t_0; the step size may be uniform or varying.

   It should be noted that the principle of discretization is still compatible with ideal real-number arithmetic over $\mathbb{R}$. In the Haskell-oriented literature this is usually modeled by *streams* of sample values $f(t_i)$. For example, in [28] it is shown that this approximation is – under certain constraints – faithful in the limit, as the step size goes to zero. (This is not really surprising in the light of century-old work by mathematicians like Cauchy and others, since their techniques are essentially translated here into Haskell-speak.)

2. *Precision*. The real numbers $\mathbb{R}$ are replaced by machine numbers of limited size, usually `float` or at best `double`, on some processors even by fixed-point emulations of floating-point numbers. This limited precision combined with the need for not only finite but actually efficient computation leads to several kinds of errors:

   - The use of limited machine numbers leads to the well-known *rounding errors*.
   - The numerical solution $\tilde{x}$ obtained by the solver is only an approximation of the solution $x(t)$ at discrete time points. This leads to *discretization errors*.
   - Computations such as integration or Newton iteration for finding zero values are only executed with certain (fixed or variable) step sizes and terminated after a finite number of steps. This also leads to *discretization and approximation errors*.
   - Parameters and input values generally come with a certain error. For example, neither π nor the gravity g have their exact values. This in addition leads to *modeling errors*.
   - Event detection for variable-structure systems usually comprises interpolation of the discrete solution $\tilde{x}$ between mesh points t_i and t_{i+1}, and a root finding procedure. Again, this introduces an error.

We cannot do away with the intrinsic difficulties of Numerics. In particular when dealing with differential-algebraic systems there are a number of additional difficulties that have to be dealt with, e.g. the problem of order reduction of numerical methods and drift-off of the numerical solution due to high index problems or the problem of inconsistencies of initial values. A number of elaborate methods, including numerical integration, index reduction and consistent initialization have been developed over the last decades to deal with these problems, see [2, 9, 13].

The purpose of (this section of) our paper is not to indulge into these numeric issues. Rather we accept their effects as a given fact and study the impacts that this observation has for our semantic considerations.

We should point out quite clearly that this is all work in progress and that the following is but a sketch of a research direction.

7.2 Uncertainty

In order to get a grasp on these difficulties we employ the notion of *uncertainty*. That is, all values are considered to be "uncertain". This applies to all kinds of values that are computed in the simulation, also including the time points t^*. For example, if we compute the so-called zero crossing in the bouncing-ball example, we obtain the point t^* in time, at which the guard $y = 0$ becomes true and fires. However, this is only so in the ideal semantics. In the solver semantics we have an uncertain value $\tilde{y}$ which leads to an uncertain time point $\tilde{t}^*$.

As mentioned before, we are not interested in the Numerics behind this uncertainty. Uncertain values may be represented as simple intervals of real numbers or they may be represented as intervals with a, say, Gaussian distribution. One may even use ideas of fuzzy logic for such a representation. The challenge in Numerical Mathematics is to come up with algorithms and methods that allow us to infer the uncertainty of the result of a computation from the inputs of the computation.

In the following we presume the existence of such a notion of uncertainty. Then we can derive the solver semantics along the same lines as the ideal semantics, but now using the uncertain values $\tilde{x}$ and their computations in the place of the real values x, i.e., $\tilde{x} = x \pm \omega$ for a small uncertainty ω.

Then, the solver semantics are defined by the signature $\tilde{\Sigma}$ containing float or double types and floating point operations (introducing rounding errors) and the set of variables $\tilde{X}$ with corresponding uncertainties contained in the set $\tilde{W}$. Let $\tilde{E}$ be the corresponding set of equations, then for a specification $\tilde{S} = (\tilde{\Sigma}, \tilde{E})$ the set of models is denoted by *Mod*$(\tilde{S}) = \{ \tilde{A} \mid \tilde{A} \models \tilde{S} \}$ and a component is defined by $\tilde{C} = (\tilde{V}, \tilde{W}, \tilde{E})$. Composition of components and of subsystems can be defined in an analogous way as in Section 3. The same holds true for the definition of variable-structure components.

7.3 Semantics and Uncertainty

If we analyze the definitions in the previous sections, it is easily seen that the difference between the ideal and the solver semantics only plays a role at a few points. To begin with, all composition operators are not affected by the differences in the two semantics. What needs to be considered are the following issues:

- The fulfillment of equations, that is $(\tilde{A} \models \tilde{E})$ in the structure $\tilde{A}$ is influenced by uncertainty. The meaning of an equation $\tilde{x} = \tilde{y}$ as compared to $x = y$ is uncertain again. To analyze such an uncertain equation mathematically interval arithmetic can be employed. Since all our constructions are defined relative to the relation "is-a-model-of" (short: "$\models$"), there is no fundamental problem here.
- The action parts of the mode transitions are analogous to the above equations, since we essentially need to solve an uncertain equation of the form $\tilde{x}_{new} = f(\tilde{x}_{old})$.
- The most severe problem concerns the *guards*. Here we run into problems, since our definition of the transition point t^* now has to be interpreted with uncertainty, i.e., $\tilde{t^*}$ is defined by $\tilde{g}(\tilde{t^*}) = true$ and $\tilde{g}(\tau) = false$ for all τ with $\tilde{t_1^\alpha} < \tau < \tilde{t^*}$. Since both, the guard function $\tilde{g}$ and $\tilde{t^*}$ are blurred, the whole interval $\tilde{T_1} = [\tilde{t_1^\alpha}, \tilde{t^*})$ is blurred. This could have major effects. For example, in the case of two very close events the uncertainty could mean that the later event actually fires before the earlier one. Or two events that in reality happen simultaneously are considered as being separate due to uncertainty effects. As can be seen in the bouncing-ball example the events will (towards the end of the simulation) be so close together that – due to uncertainty – they can no longer be distinguished. (This leads in many animated simulations to the funny effect that in the end the ball breaks through the surface and travels towards the center of the earth.)

The question, if the solver semantics are faithful, i.e. converge to the ideal semantics, when the discretization step size tends to zero, strongly depends on the given model and on the employed numerical solver. Statements under which conditions a specific solver semantics converges to its ideal semantics have to be derived for individual cases.

The most challenging problems occur in variable-structure systems: besides the problem of robust treatment in the case of blurring event times one has to ensure that each instance of a mode lives long enough since the event time $\tilde{t}^*$ is determined as zero crossing in the discretization interval $[t_i, t_{i+1}]$. Concerning DAEs, in variable-structure systems not only the initial values have to be ensured to be consistent with the algebraic constraints, but also consistency of each reinitialization after events has to be guaranteed. We will not go further into the details of these problems in this paper. For the analysis and numerical treatment of hybrid DAEs we refer to [17].

These problems do exist and there is no general mechanism to avoid them. As a matter of fact, most of these effects need an application-dependent individual treatment. What we suggest is that in the semantic specification of modeling languages these solver-related effects are clearly separated from the other semantic concepts. Moreover, it should be discussed what kinds of language constructs could be added such that the modeler has the capability to describe these uncertainty effects appropriately.

8. Conclusion

We have presented a compositional semantics for essential parts of Modelica-style modeling languages. Such a compositional semantics is a mandatory prerequisite for a clean design of conceptual ideas such as variable-structure systems or compilation paradigms such as separate compilation.

Moreover, it is important to provide a clean separation of the basic modeling principles of a language from the effects that are caused by the limitations of numerics. Clearly, a deeper analysis of the latter issue is a field for extensive research in the realm of Numerical Analysis. This separation is strongly motivated by the following consideration (as was pointed out explicitly by one of the reviewers): The possibility to handle structural changes in a model and the ease of doing this depends on the computational framework that is used. By separating the semantics of the model from that of the computational framework, different frameworks can be applied to the same model, thus allowing one to realize simulators for a larger set of structural changes in the future.

References

[1] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[2] Kathryn E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, volume 14 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 1996.

[3] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis,

Linköping University, 2010.

[4] David Broman and Peter Fritzson. Higher-Order Acausal Models. *Simulation News Europe*, 19(1):5–16, 2009.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[6] George Giorgidze and Henrik Nilsson. Embedding a Functional Hybrid Modelling language in Haskell. In *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Languages, Hatfield, England*, volume 5836 of *Lecture Notes in Computer Science*. Springer, 2008.

[7] George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10)*, pages 19–34, 2010.

[8] Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In *Proceedings of the First InternationalWorkshop on Hybrid and Real-time Systems (HART 97)*, Lecture Notes in Computer Science 1201, pages 331–345, 1997.

[9] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, second edition, 1996.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[11] Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.

[12] Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304 USA. *Specware System and documentation*, 2003. http://www.specware.org/.

[13] Peter Kunkel and Volker Mehrmann. *Differential-Algebraic Equations — Analysis and Numerical Solution*. EMS Publishing House, Zürich, Switzerland, 2006.

[14] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *REX Workshop*, pages 447–484, 1991.

[15] Zohar Manna and Amir Pnueli. Verifying hybrid systems. In *Hybrid Systems*, pages 4–35, 1992.

[16] Alexandra Mehlhase. Varying the level of detail during simulation. In *to appear in Proc. ASIM 2011*, 2011.

[17] Volker Mehrmann and Lena Wunderlich. Hybrid systems of differential-algebraic equations – analysis and numerical solution. *Journal of Process Control*, 19:1218–1228, 2009.

[18] Pieter J. Mosterman. Hybrid dynamic systems: mode transition behavior in hybrid dynamic systems. In Chick S, P. J. Sanchez, D. Ferrin, and D. J. Morrice, editors, *Proc. 2003 Winter Simulation Conference*, pages 623–631, 2003.

[19] Henrik Nilsson and George Giorgidze. Exploiting structural dynamism in functional hybrid modelling for simulation of ideal diodes. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic*. Czech Technical University Publishing House, 2010.

[20] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of 5th Int. Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2003.

[21] Christoph Nytsch-Geusen, Andre Nordwig, Thilo Ernst, Peter Schwarz, Matthias Vetter, Christoph Wittwer, Andreas Holm, Jürgen Leopold, Gerhardt Schmidt, Ulrich Doll, and Alexander Mattes. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, 2005.

[22] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Evolving specification engineering. In *AMAST*, pages 299–314, 2008.

[23] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[24] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change: Towards optimisation of FRP by reasoning about change. *Journal of Higher-Order and Symbolic Computation (HOSC)*, 24(1), 2011.

[25] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Foundations of an interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio Butazzo, editors, *10th International Workshop on Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600. Springer, 2007.

[26] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Revised hybrid system interchange format. Technical Report HYCON Deliverable D3.6.3, HYCON NoE, 2007.

[27] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Concrete syntax and semantics of the compositional interchange format for hybrid systems. In *Proceedings of the 17th IFAC World Congress (IFAC'08) July 11-16, 2008, Seoul, Korea*, 2008.

[28] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI 2000: Symposium on Programming Language Design and Implementation*, pages 242–252, 2000.

[29] Dirk Zimmer. *Equation-Based Modeling of Variable Structure Systems*. PhD thesis, ETH Zürich, 2010.